【带着问题学】协程到底是怎么切换线程的?

程序员江同学 2021-07-04 8,951 阅读9分钟 关注

「本文已参与好文召集令活动，点击查看：后端、大前端双赛道投稿，2万元奖池等你挑战！」

前言

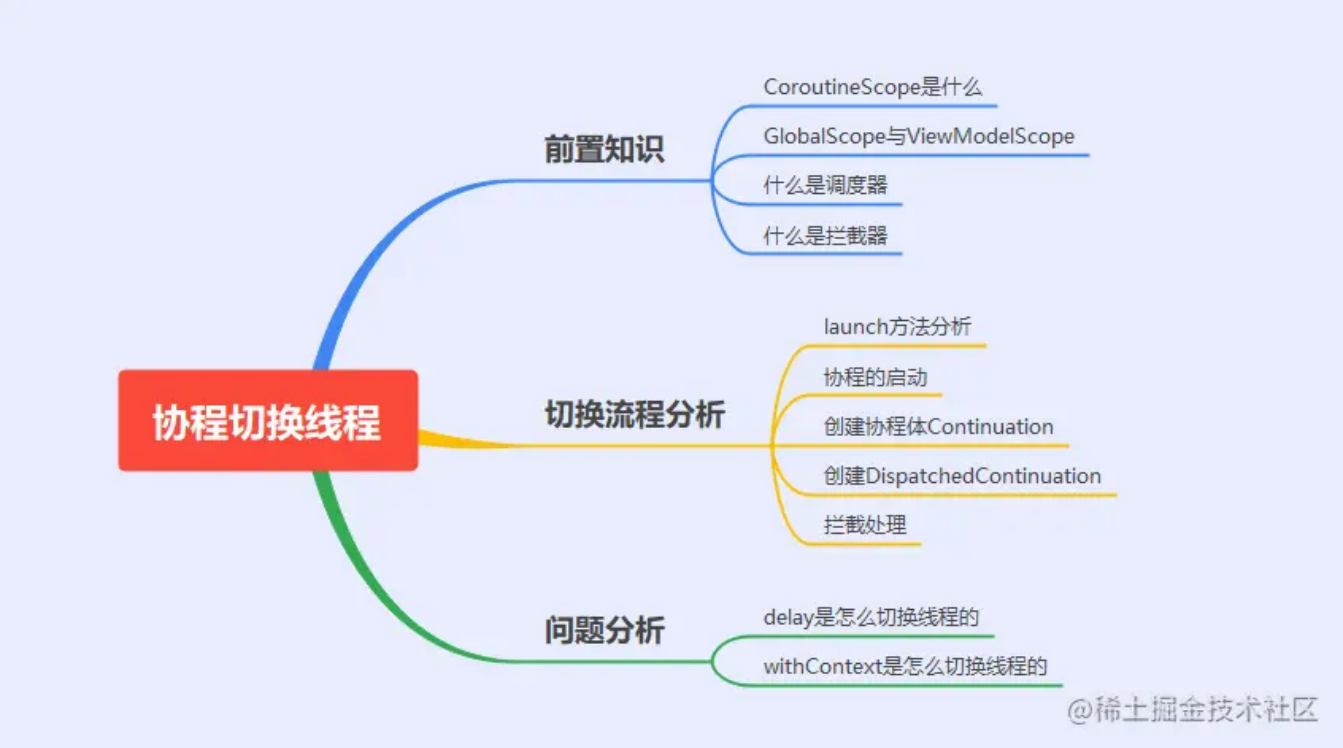
之前对协程做了一个简单的介绍，回答了**协程到底是什么**的问题,感兴趣的同学可以了解下：【带着问题学】协程到底是什么?

通过上文，我们了解了以下内容

1. `kotlin` 协程本质上对线程池的封装
2. `kotlin` 协程可以用同步方式写异步代码，自动实现对线程切换的管理

这就引出了本文的主要内容， `kotlin` 协程到底是怎么切换线程的?

具体内容如下:



1. 前置知识

`CoroutineScope` 即协程运行的作用域,它的源码很简单

```kotlin
public interface CoroutineScope {
    public val coroutineContext: CoroutineContext
}
```

可以看出 `CoroutineScope` 的代码很简单，主要作用是提供 `CoroutineContext` ，协程运行的上下文

我们常见的实现有 `GlobalScope` , `LifecycleScope` , `ViewModelScope` 等

1.2 `GlobalScope` 与 `ViewModelScope` 有什么区别?

```kotlin
public object GlobalScope : CoroutineScope {
    /**
     * 返回 [EmptyCoroutineContext].
     */
    override val coroutineContext: CoroutineContext
        get() = EmptyCoroutineContext
}

public val ViewModel.viewModelScope: CoroutineScope
    get() {
        val scope: CoroutineScope? = this.getTag(JOB_KEY)
        if (scope != null) {
            return scope
        }
        return setTagIfAbsent(
            JOB_KEY,
            CloseableCoroutineScope(SupervisorJob() + Dispatchers.Main.immediate)
        )
    }
```

两者的代码都挺简单，从上面可以看出

1. `GlobalScope` 返回的为 `CoroutineContext` 的空实现
2. `ViewModelScope` 则往 `CoroutineContext` 中添加了 `Job` 与 `Dispatcher`

```
fun testOne(){
        GlobalScope.launch {
          print("1:" + Thread.currentThread().name)
          delay(1000)
          print("2:" + Thread.currentThread().name)
      }
}
//打印结果为：DefaultDispatcher-worker-1
  fun testTwo(){
      viewModelScope.launch {
          print("1:" + Thread.currentThread().name)
          delay(1000)
          print("2:" + Thread.currentThread().name)
      }
  }
  //打印结果为：main
```

上面两种 `Scope` 启动协程后，打印当前线程名是不同的，一个是线程池中的一个线程，一个则是主线程

这是因为 `ViewModelScope` 在 `CoroutineContext` 中添加了 `Dispatchers.Main.immediate` 的原因

我们可以得出结论：协程就是通过 `Dispatchers` 调度器来控制线程切换的

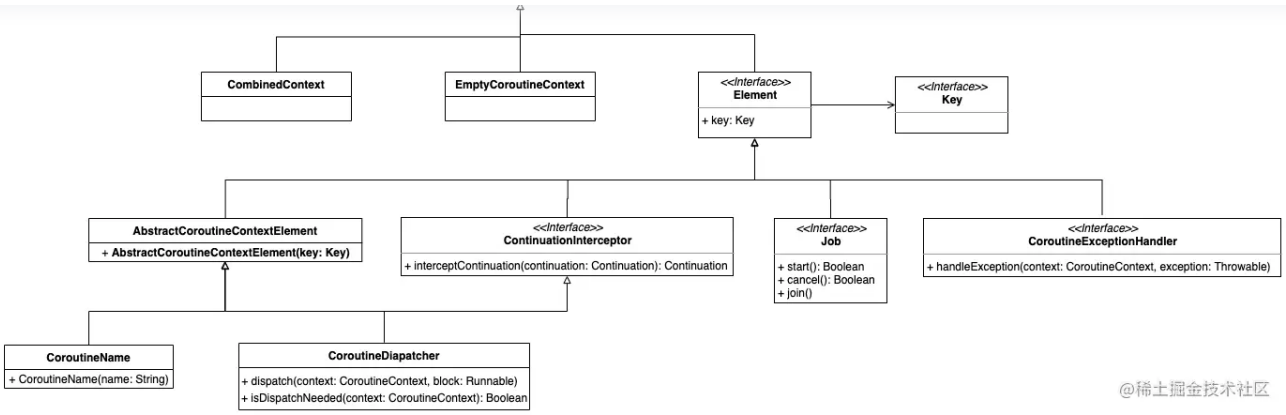
1.3 什么是调度器?

从使用上来讲，调度器就是我们使用的 `Dispatchers.Main` , `Dispatchers.Default` ， `Dispatcher.IO` 等

从作用上来讲，调度器的作用是控制协程运行的线程

从结构上来讲， `Dispatchers` 的父类是 `ContinuationInterceptor` ,然后再继承于 `CoroutineContext`

它们的类结构关系如下：



这也是为什么 `Dispatchers` 能加入到 `CoroutineContext` 中的原因,并且支持 `+` 操作符来完成增加

1.4 什么是拦截器

从命名上很容易看出， `ContinuationInterceptor` 即协程拦截器，先看一下接口

kotlin 复制代码

```kotlin
interface ContinuationInterceptor : CoroutineContext.Element {
    // ContinuationInterceptor 在 CoroutineContext 中的 Key
    companion object Key : CoroutineContext.Key<ContinuationInterceptor>
    /**
     * 拦截 continuation
     */
    fun <T> interceptContinuation(continuation: Continuation<T>): Continuation<T>

    //...
}
```

从上面可以提炼出两个信息

1.拦截器的 `Key` 是单例的，因此当你添加多个拦截器时，生效的只会有一个

2.我们都知道， `Continuation` 在调用其 `Continuation#resumeWith()` 方法，会执行其 `suspend` 修饰的函数的代码块，如果我们提前拦截到，是不是可以做点其他事情？这就是调度器切换线程的原理

上面我们已经介绍了是通过 `Dispatchers` 指定协程运行的线程，通过 `interceptContinuation` 在协程恢复前进行拦截，从而切换线程

带着这些前置知识，我们一起来看下协程启动的具体流程，明确下协程切换线程源码具体实现

2. 协程线程切换源码分析

我们首先看一下协程是怎样启动的，传入了什么参数

```kotlin
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job {
    val newContext = newCoroutineContext(context)
    val coroutine = if (start.isLazy)
        LazyStandaloneCoroutine(newContext, block) else
        StandaloneCoroutine(newContext, active = true)
    coroutine.start(start, coroutine, block)
    return coroutine
}
```

总共有3个参数：

1.传入的协程上下文

2. `CoroutinStart` 启动器，是个枚举类，定义了不同的启动方法，默认是 `CoroutineStart.DEFAULT`

3. `block` 就是我们传入的协程体，真正要执行的代码

这段代码主要做了两件事:

1.组合新的 `CoroutineContext`

2.再创建一个 Continuation

2.1.1 组合新的 `CoroutineContext`

```kotlin
public actual fun CoroutineScope.newCoroutineContext(context: CoroutineContext): CoroutineContex
    val combined = coroutineContext + context
    val debug = if (DEBUG) combined + CoroutineId(COROUTINE_ID.incrementAndGet()) else combined
    return if (combined !== Dispatchers.Default && combined[ContinuationInterceptor] == null)
        debug + Dispatchers.Default else debug
}
```

2.如果 `combined` 中没有拦截器，会传入一个默认的拦截器，即 `Dispatchers.Default` ,这也解释了为什么我们没有传入拦截器时会有一个默认切换线程的效果

2.1.2 创建一个 `Continuation`

```kotlin
val coroutine = if (start.isLazy)
        LazyStandaloneCoroutine(newContext, block) else
        StandaloneCoroutine(newContext, active = true)
    coroutine.start(start, coroutine, block)
```

默认情况下，我们会创建一个 `StandloneCoroutine`

值得注意的是，这个 `coroutine` 其实是我们协程体的 `complete` ，即成功后的回调，而不是协程体本身

然后调用 `coroutine.start` ，这表明协程开始启动了

2.2 协程的启动

```kotlin
public fun <R> start(start: CoroutineStart, receiver: R, block: suspend R.() -> T) {
    initParentJob()
    start(block, receiver, this)
}
```

接着调用 `CoroutineStart` 的 `start` 来启动协程，默认情况下调用的是 `CoroutineStart.Default`

经过层层调用，最后到达了：

```kotlin
internal fun <R, T> (suspend (R) -> T).startCoroutineCancellable(receiver: R, completion: Contin
    runSafely(completion) {
        // 外面再包一层 Coroutine
        createCoroutineUnintercepted(receiver, completion)
            // 如果需要，做拦截处理
            .intercepted()
            // 调用 resumeWith 方法
```

这里就是协程启动的核心代码，虽然比较短，却包括3个步骤：

1.创建协程体 `Continuation`

2.创建拦截 `Continuation`,即 `DispatchedContinuation`

3.执行 `DispatchedContinuation.resumeWith` 方法

2.3 创建协程体 `Continuation`

调用 `createCoroutineUnintercepted`,会把我们的协程体即 `suspend block` 转换成 `Continuation`,它是 `SuspendLambda`，继承自 `ContinuationImpl`

`createCoroutineUnintercepted` 方法在源码中找不到具体实现，不过如果你把协程体代码反编译后就可以看到真正的实现

详情可见：字节码反编译

2.4 创建 `DispatchedContinuation`

kotlin 复制代码

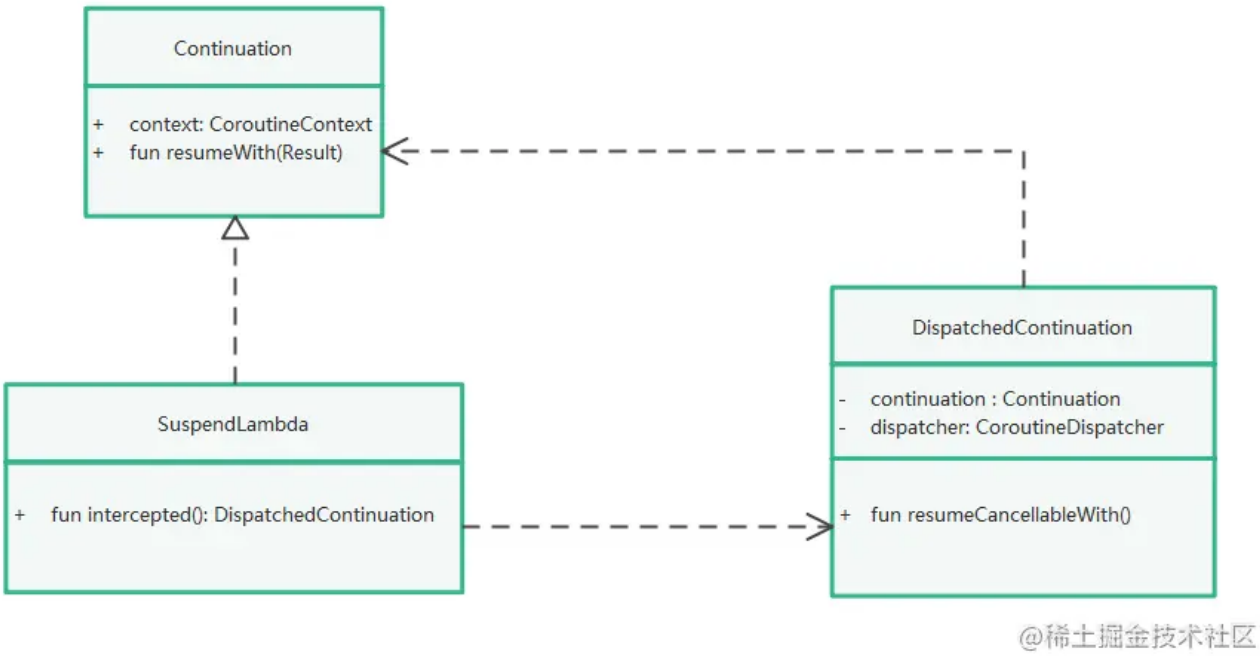
```kotlin
public actual fun <T> Continuation<T>.intercepted(): Continuation<T> =
    (this as? ContinuationImpl)?.intercepted() ?: this

//ContinuationImpl
public fun intercepted(): Continuation<Any?> =
        intercepted
            ?: (context[ContinuationInterceptor]?.interceptContinuation(this) ?: this)
                .also { intercepted = it }

//CoroutineDispatcher
public final override fun <T> interceptContinuation(continuation: Continuation<T>): Continuation
      DispatchedContinuation(this, continuation)
```

从上可以提炼出以下信息

1. `interepted` 是个扩展方法，最后会调用到 `ContinuationImpl.intercepted` 方法

2.在 `intercepted` 会利用 `CoroutineContext`，获取当前的拦截器

4.我们将协程体的 `Continuation` 传入 `DispatchedContinuation` ,这里其实用到了 `装饰器模式` ，实现功能的增强



这里其实很明显了，通过 `DispatchedContinuation` 装饰原有协程，在 `DispatchedContinuation` 里通过调度器处理线程切换，不影响原有逻辑，实现功能的增强

2.5 拦截处理

kotin 复制代码

```
//DispatchedContinuation
inline fun resumeCancellableWith(
    result: Result<T>,
    noinline onCancellation: ((cause: Throwable) -> Unit)?
) {
    val state = result.toState(onCancellation)
    if (dispatcher.isDispatchNeeded(context)) {
        _state = state
        resumeMode = MODE_CANCELLABLE
        dispatcher.dispatch(context, this)
    } else {
        executeUnconfined(state, MODE_CANCELLABLE) {
            if (!resumeCancelled(state)) {
                resumeUndispatchedWith(result)
            }
        }
```

上面说到了启动时会调用 `DispatchedContinuation` 的 `resumeCancellableWith` 方法

这里面做的事也很简单：

1.如果需要切换线程，调用 `dispatcher.dispatcher` 方法，这里的 `dispatcher` 是通过 `CoroutineConext` 取出来的

2.如果不需要切换线程，直接运行原有线程即可

2.5.2 调度器的具体实现

我们首先明确下， `CoroutineDispatcher` 是通过 `CoroutineContext` 取出来的,这也是协程上下文作用的体现

`CoroutineDispater` 官方提供了四种实现：

`Dispatchers.Main` , `Dispatchers.IO` , `Dispatchers.Default` , `Dispatchers.Unconfined`

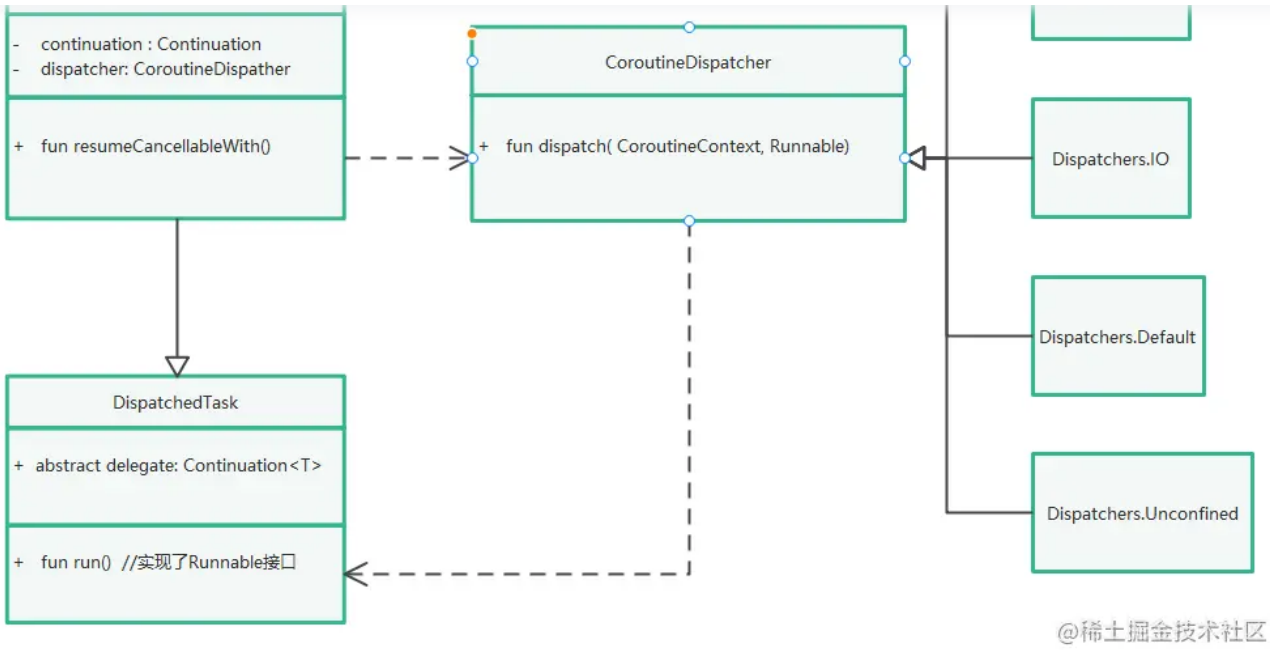
我们一起简单看下 `Dispatchers.Main` 的实现

kotlin 复制代码

```kotlin
internal class HandlerContext private constructor(
    private val handler: Handler,
    private val name: String?,
    private val invokeImmediately: Boolean
) : HandlerDispatcher(), Delay {
    public constructor(
        handler: Handler,
        name: String? = null
    ) : this(handler, name, false)

    //...

    override fun dispatch(context: CoroutineContext, block: Runnable) {
        // 利用主线程的 Handler 执行任务
        handler.post(block)
    }
}
```

可以看到，其实就是用 `handler` 切换到了主线程

如果用 `Dispatcers.IO` 也是一样的，只不过换成线程池切换了



如上所示，其实就是一个装饰模式

1.调用 `CoroutinDispatcher.dispatch` 方法切换线程

2.切换完成后调用 `DispatchedTask.run` 方法，执行真正的协程体

3 `delay` 是怎样切换线程的?

上面我们介绍了协程线程调度的基本原理与实现,下面我们来回答几个小问题

我们知道 `delay` 函数会挂起，然后等待一段时间再恢复。

可以想象，这里面应该也涉及到线程的切换，具体是怎么实现的呢?

kotlin 复制代码

```kotlin
public suspend fun delay(timeMillis: Long) {
    if (timeMillis <= 0) return // don't delay
    return suspendCancellableCoroutine sc@ { cont: CancellableContinuation<Unit> ->
        // if timeMillis == Long.MAX_VALUE then just wait forever like awaitCancellation, don't
        if (timeMillis < Long.MAX_VALUE) {
            cont.context.delay.scheduleResumeAfterDelay(timeMillis, cont)
        }
    }
}

internal val CoroutineContext.delay: Delay get() = get(ContinuationInterceptor) as? Delay ?: Def
```

我们来看一个具体实现

```kotlin
internal class HandlerContext private constructor(
    private val handler: Handler,
    private val name: String?,
    private val invokeImmediately: Boolean
) : HandlerDispatcher(), Delay {
    override fun scheduleResumeAfterDelay(timeMillis: Long, continuation: CancellableContinuatio
        // 利用主线程的 Handler 延迟执行任务，将完成的 continuation 放在任务中执行
        val block = Runnable {
            with(continuation) { resumeUndispatched(Unit) }
        }
        handler.postDelayed(block, timeMillis.coerceAtMost(MAX_DELAY))
        continuation.invokeOnCancellation { handler.removeCallbacks(block) }
    }

    //..
}
```

1.可以看出，其实也是通过 `handler.postDelayed` 实现延时效果的

2.时间到了之后，再通过 `resumeUndispatched` 方法恢复协程

3.如果我们用的是 `Dispatcher.IO` ，效果也是一样的，不同的就是延时效果是通过切换线程实现的

4. `withContext` 是怎样切换线程的?

我们在协程体内，可能通过 `withContext` 方法简单便捷的切换线程，用同步的方式写异步代码，这也是 `kotin` 协程的主要优势之一

```kotlin
    fun test(){
        viewModelScope.launch(Dispatchers.Main) {
            print("1:" + Thread.currentThread().name)
            withContext(Dispatchers.IO){
                delay(1000)
                print("2:" + Thread.currentThread().name)
            }
            print("3:" + Thread.currentThread().name)
        }
```

可以看出这段代码做了一个切换线程然后再切换回来的操作，我们可以提出两个问题

1. `withContext` 是怎样切换线程的?

2. `withContext` 内的协程体结束后，线程怎样切换回到 `Dispatchers.Main` ?

kotlin 复制代码

```kotlin
public suspend fun <T> withContext(
    context: CoroutineContext,
    block: suspend CoroutineScope.() -> T
): T {
    return suspendCoroutineUninterceptedOrReturn sc@ { uCont ->
        // 创建新的context
        val oldContext = uCont.context
        val newContext = oldContext + context
        ....
        //使用新的Dispatcher，覆盖外层
        val coroutine = DispatchedCoroutine(newContext, uCont)
        coroutine.initParentJob()
        //DispatchedCoroutine作为了complete传入
        block.startCoroutineCancellable(coroutine, coroutine)
        coroutine.getResult()
    }
}

private class DispatchedCoroutine<in T>(
    context: CoroutineContext,
    uCont: Continuation<T>
) : ScopeCoroutine<T>(context, uCont) {
  //在complete时会会回调
    override fun afterCompletion(state: Any?) {
        afterResume(state)
    }

    override fun afterResume(state: Any?) {
        //uCont就是父协程，context仍是老版context,因此可以切换回原来的线程上
        uCont.intercepted().resumeCancellableWith(recoverResult(state, uCont))
    }
}
```

这段代码其实也很简单，可以提炼出以下信息

1. `withContext` 其实就是一层 `Api` 封装，最后调用到了 `startCoroutineCancellable` ,这就跟 `launch` 后面的流程一样了，我们就不继续跟了

2.传入的 `context` 会覆盖外层的拦截器并生成一个 `newContext` ，因此可以实现线程的切换

4. `DispatchedCoroutine` 中传入的 `uCont` 是父协程，它的拦截器仍是外层的拦截器，因此会切换回原来的线程中

总结

本文主要回答了 `kotlin` 协程到底是怎么切换线程的这个问题，并对源码进行了分析

简单来讲主要包括以下步骤：

1.向 `CoroutineContext` 添加 `Dispatcher` ，指定运行的协程

2.在启动时将 `suspend block` 创建成 `Continuation` ,并调用 `intercepted` 生成 `DispatchedContinuation`

3. `DispatchedContinuation` 就是对原有协程的装饰，在这里调用 `Dispatcher` 完成线程切换任务后，`resume` 被装饰的协程，就会执行协程体内的代码了

其实 `kotlin` 协程就是用装饰器模式实现线程切换的

看起来似乎有不少代码，但是真正的思路其实还是挺简单的，这大概就是设计模式的作用吧

如果本文对你有所帮助，欢迎点赞收藏~

参考资料

深入浅出kotlin协程

破解 Kotlin 协程(3) - 协程调度篇

标签： Kotlin Android

本文收录于以下专栏

kotlin协程从入门到精通 专栏目录

kotlin协程系列文章，带您深入了解kotlin协程的基本原理，使用与进阶...

51 订阅 · 3 篇文章

订阅

上一篇 【带着问题学】协程到底是什么?

登录 / 注册 即可发布评论！

爱海贼的小码农 LV.3 Android @无业游民

希望作者有空的花可以补一期，协程是怎么做线程的恢复的。因为我觉得kotlin的协程，更重要的一部分是做恢复的动作。🙏

1年前 1 1 ···

小鱼人爱编程：juejin.cn

1年前 1 回复 ···

coding_dyp

写的很不错，可以学习下👍

2年前 点赞 评论 ···

树洞robter 自动匿名机器人#树洞一下#

写的不错 关注了兄弟

2年前 点赞 1 ···

程序员江同学 作者：谢谢认可~

2年前 1 回复 ···

查看全部 9 条评论 >

目录 收起

相关推荐

【JVM入门食用指南-04】Android虚拟机&类加载机制
1.7k阅读 · 8点赞

Android Studio 中 CPU Profiler 系统性能分析工具的使用
4.5k阅读 · 18点赞

来讨论下 Android 面试该问什么类型的题目?
6.0k阅读 · 132点赞

Android Jetpack 开发套件 #9 食之无味！App Startup 可能比你想象中要简单
6.6k阅读 · 55点赞

Android 性能优化系列：抖音字节跳动技术团队教你Java 内存优化
685阅读 · 1点赞

精选内容

深入浅出区块链 Day5
后端要努力 · 556阅读 · 1点赞

Django 中使用 Jinja 模板引擎
K8sCat · 556阅读 · 0点赞

🔥🔥Istio：微服务开发的终极利器，你还在为繁琐的通信和部署流程烦恼吗?
努力的小雨 · 663阅读 · 4点赞

LeetCode第83题删除排序链表中的重复元素
服务端技术栈 · 583阅读 · 2点赞

为你推荐

MVVM 进阶版：MVI 架构了解一下~
程序员江同学 | 2年前 | 49k | 358 | 115 Android 架构

Google 推荐使用 MVI 架构？卷起来了~
程序员江同学 | 1年前 | 43k | 285 | 57 Android 架构

三年经验Android开发面经总结
程序员江同学 | 3年前 | 33k | 526 | 91 面试

【Gradle7.0】依赖统一管理的全新方式，了解一下~
程序员江同学 | 2年前 | 39k | 240 | 35 Android gradle

官方推荐 Flow 取代 LiveData,有必要吗?
程序员江同学 | 2年前 | 28k | 325 | 50 Androi... Kotlin

MVI 架构封装：快速优雅地实现网络请求
程序员江同学 | 1年前 | 24k | 113 | 51 Android 架构

Compose + MVI + Navigation 快速实现 wanAndroid 客户端
程序员江同学 | 1年前 | 17k | 126 | 26 Android 开源

相比 XML，Compose 性能到底怎么样?
程序员江同学 | 2年前 | 16k | 92 | 25 Android

【带着问题学】android事件分发8连问
程序员江同学 | 2年前 | 16k | 96 | 14 源码 Android

【带着问题学】协程到底是什么?
程序员江同学 | 2年前 | 13k | 189 | 38 Kotlin Android

【知识点】OkHttp 原理 8 连问

程序员江同学 | 2年前 | 13k 171 8 Android 源码

落地 Kotlin 代码规范，DeteKt 了解一下~

程序员江同学 | 1年前 | 12k 169 16 Android Kotlin

MVI 架构更佳实践：支持 LiveData 属性监听

程序员江同学 | 2年前 | 13k 85 32 Android 架构

【带着问题学】Glide做了哪些优化?

程序员江同学 | 2年前 | 11k 144 31 性能优化