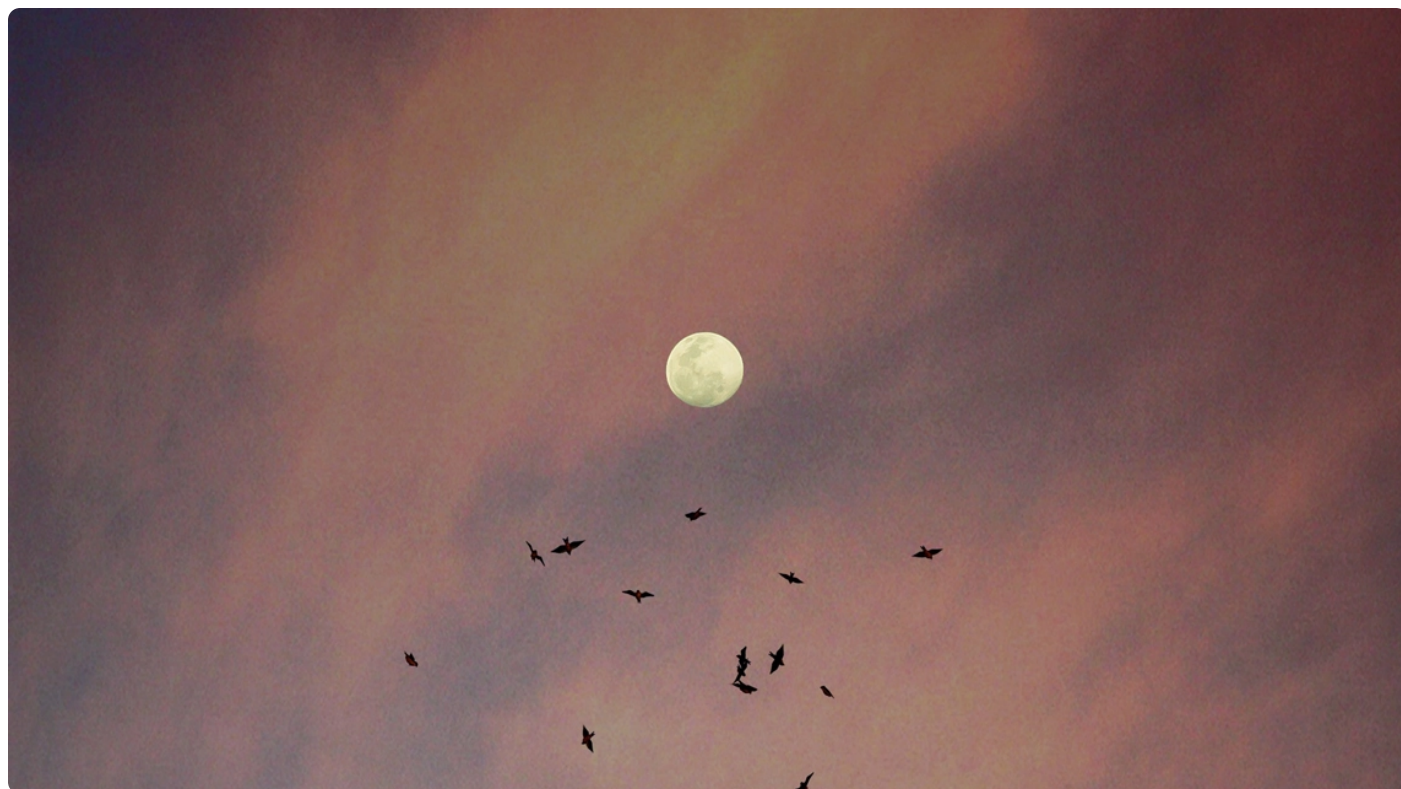


18 | 实战：让KtHttp支持挂起函数

2022-02-23 朱涛

《朱涛·Kotlin编程第一课》

课程介绍 >



讲述：朱涛

时长 18:08 大小 16.61M



你好，我是朱涛。今天这节实战课，我们接着前面 [第 12 讲](#) 里实现的网络请求框架，来进一步完善这个 KtHttp，让它支持挂起函数。

在上一次实战课当中，我们已经开发出了两个版本的 KtHttp，1.0 版本的是基于命令式风格的，2.0 版本的是基于函数式风格的。其中 2.0 版本的代码风格，跟我们平时工作写的代码风格很不一样，之前我也说了，这主要是因为业界对 Kotlin 函数式编程接纳度并不高，所以这节课的代码，我们将基于 1.0 版本的代码继续改造。这样，也能让课程的内容更接地气一些，甚至你都可以借鉴今天写代码的思路，复用到实际的 Android 或者后端开发中去。

跟往常一样，这节课的代码还是会分为两个版本：

- 3.0 版本，在之前 1.0 版本的基础上，扩展出**异步请求**的能力。
- 4.0 版本，进一步扩展异步请求的能力，让它**支持挂起函数**。

好，接下来就正式开始吧！

3.0 版本：支持异步（Call）

有了上一次实战课的基础，这节课就会轻松一些了。关于动态代理、注解、反射之类的知识不会牵涉太多，我们今天主要把精力都集中在协程上来。不过，在正式开始写协程代码之前，我们需要先让 KtHttp 支持异步请求，也就是 Callback 请求。

这是为什么呢？别忘了 [第 15 讲](#) 的内容：**挂起函数本质就是 Callback！** 所以，为了让 KtHttp 支持挂起函数，我们可以采用迂回的策略，让它先支持 Callback。在之前 1.0、2.0 版本的代码中，KtHttp 是只支持同步请求的，你可能对异步同步还有些懵，我带你来看个例子吧。

首先，这个是同步代码：

 复制代码

```
1 fun main() {
2     // 同步代码
3     val api: ApiService = KtHttpV1.create(ApiService::class.java)
4     val data: RepoList = api.repos(lang = "Kotlin", since = "weekly")
5     println(data)
6 }
```

可以看到，在 main 函数当中，我们调用了 KtHttp 1.0 的代码，其中 3 行代码的运行顺序是 1、2、3，这就是典型的同步代码。它的另一个特点就是：所有代码都会在一个线程中执行，因此这样的代码如果运行在 Android、Swing 之类的 UI 编程平台上，是会导致主线程卡死的。

那么，异步代码又是长什么样的呢？

 复制代码

```
1 private fun testAsync() {
2     // 异步代码
3     KtHttpV3.create(ApiServiceV3::class.java).repos(
4         lang = "Kotlin",
5         since = "weekly"
6     ).call(object : Callback<RepoList> {
7         override fun onSuccess(data: RepoList) {
8             println(data)
9         }
10    })
11 }
```

```

11         override fun onFail(throwable: Throwable) {
12             println(throwable)
13         }
14     })
15 }

```

上面的 `testAsync()` 方法当中的代码，就是典型的异步代码，它跟同步代码最大的差异就是，有了一个 **Callback**，而且代码不再是按照顺序执行的了。你可以参考下面这个动图：

异步Callback

```

private fun testAsync() {
    KtHttpV3.create(ApiServiceV3::class.java).repos(
        lang = "Kotlin",
        since = "weekly"
    ).call(object : Callback<RepoList> {
        override fun onSuccess(data: RepoList) {
            println(data)
        }

        override fun onFail(throwable: Throwable) {
            println(throwable)
        }
    })
}

```

所以，在 3.0 版本的开发中，我们就是要实现类似上面 `testAsync()` 的请求方式。为此，我们首先需要创建一个 **Callback 接口**，在这个 **Callback** 当中，我们可以拿到 **API** 请求的结果。

 复制代码

```

1 interface Callback<T: Any> {
2     fun onSuccess(data: T)
3     fun onFail(throwable: Throwable)
4 }

```

在 **Callback** 这个接口里，有一个泛型参数 **T**，还有两个回调，分别是 `onSuccess` 代表接口请求成功、`onFail` 代表接口请求失败。需要特别注意的是，这里我们运用了 [空安全思维](#) 当中的泛型边界“**T: Any**”，这样一来，我们就可以保证 **T** 类型一定是非空的。

除此之外，我们还需要一个 **KtCall** 类，它的作用是承载 **Callback**，或者说，它是用来调用 **Callback** 的。

 复制代码

```
1 class KtCall<T: Any>(  
2     private val call: Call,  
3     private val gson: Gson,  
4     private val type: Type  
5 ) {  
6     fun call(callback: Callback<T>): Call {  
7         // TODO  
8     }  
9 }
```

KtCall 这个类仍然使用了泛型边界“**T: Any**”，另外，它还有几个关键的成员分别是：**OkHttp** 的 **Call** 对象、JSON 解析的 **Gson** 对象，以及反射类型 **Type**。然后还有一个 **call()** 方法，它接收的是前面我们定义的 **Callback** 对象，返回的是 **OkHttp** 的 **Call** 对象。所以总的来说，**call()** 方法当中的逻辑会分为三个步骤。

 复制代码

```
1 class KtCall<T: Any>(  
2     private val call: Call,  
3     private val gson: Gson,  
4     private val type: Type  
5 ) {  
6     fun call(callback: Callback<T>): Call {  
7         // 步骤1, 使用call请求API  
8         // 步骤2, 根据请求结果, 调用callback.onSuccess()或者是callback.onFail()  
9         // 步骤3, 返回OkHttp的Call对象  
10    }  
11 }
```

我们一步步来分析这三个步骤：

- 步骤 1，使用 **OkHttp** 的 **call** 对象请求 **API**，这里需要注意的是，为了将请求任务派发到异步线程，我们需要使用 **OkHttp** 的异步请求方法 **enqueue()**。
- 步骤 2，根据请求结果，调用 **callback.onSuccess()** 或者是 **callback.onFail()**。如果请求成功了，我们在调用 **onSuccess()** 之前，还需要用 **Gson** 将请求结果进行解析，然后才返回。

- 步骤 3，返回 OkHttpClient 的 Call 对象。

接下来，我们看看具体代码是怎么样的：

 复制代码

```
1 class KtCall<T: Any>(  
2     private val call: Call,  
3     private val gson: Gson,  
4     private val type: Type  
5 ) {  
6     fun call(callback: Callback<T>): Call {  
7         call.enqueue(object : okhttp3.Callback {  
8             override fun onFailure(call: Call, e: IOException) {  
9                 callback.onFail(e)  
10            }  
11  
12            override fun onResponse(call: Call, response: Response) {  
13                try { // ①  
14                    val t = gson.fromJson<T>(response.body?.string(), type)  
15                    callback.onSuccess(t)  
16                } catch (e: Exception) {  
17                    callback.onFail(e)  
18                }  
19            }  
20        })  
21        return call  
22    }  
23 }
```

经过前面的解释，这段代码就很好理解了，唯一需要注意的是注释①处，由于 API 返回的结果并不可靠，即使请求成功了，其中的 JSON 数据也不一定合法，所以这里我们一般还需要进行额外的判断。在实际的商业项目当中，我们可能还需要根据当中的状态码，进行进一步区分和封装，这里为了便于理解，我就简单处理了。

那么在实现了 KtCall 以后，我们就只差 ApiService 这个接口了，这里我们定义 ApiServiceV3，以作区分。

 复制代码

```
1 interface ApiServiceV3 {  
2     @GET("/repo")  
3     fun repos(  
4         @Field("lang") lang: String,  
5         @Field("since") since: String
```

```
6 ): KtCall<RepoList> // ①
7 }
```

我们需要格外留意以上代码中的注释①，这其实就是 **3.0 和 1.0 之间的最大区别**。由于 `repo()` 方法的返回值类型是 `KtCall`，为了支持这种写法，我们的 `invoke` 方法就需要跟着做一些小的改动：

 复制代码

```
1 // 这里也同样使用了泛型边界
2 private fun <T: Any> invoke(path: String, method: Method, args: Array<Any>): An
3     if (method.parameterAnnotations.size != args.size) return null
4
5     var url = path
6     val parameterAnnotations = method.parameterAnnotations
7     for (i in parameterAnnotations.indices) {
8         for (parameterAnnotation in parameterAnnotations[i]) {
9             if (parameterAnnotation is Field) {
10                 val key = parameterAnnotation.value
11                 val value = args[i].toString()
12                 if (!url.contains("?")) {
13                     url += "?$key=$value"
14                 } else {
15                     url += "&$key=$value"
16                 }
17             }
18         }
19     }
20 }
21
22 val request = Request.Builder()
23     .url(url)
24     .build()
25
26 val call = okHttpClient.newCall(request)
27 val genericReturnType = getTypeArgument(method)
28
29 // 变化在这里
30 return KtCall<T>(call, gson, genericReturnType)
31 }
32
33 // 拿到 KtCall<RepoList> 当中的 RepoList类型
34 private fun getTypeArgument(method: Method) =
35     (method.genericReturnType as ParameterizedType).actualTypeArguments[0]
```

在上面的代码中，大部分代码和 1.0 版本的一样的，只是在最后封装了一个 `KtCall` 对象，直接返回。所以在后续调用它的时候，我们就可以这么写了：`ktCall.call()`。

```
1 private fun testAsync() {
2     // 创建api对象
3     val api: ApiServiceV3 = KtHttpV3.create(ApiServiceV3::class.java)
4
5     // 获取ktCall
6     val ktCall: KtCall<RepoList> = api.repos(
7         lang = "Kotlin",
8         since = "weekly"
9     )
10
11    // 发起call异步请求
12    ktCall.call(object : Callback<RepoList> {
13        override fun onSuccess(data: RepoList) {
14            println(data)
15        }
16
17        override fun onFail(throwable: Throwable) {
18            println(throwable)
19        }
20    })
21 }
```

以上代码很好理解，我们一步步创建 API 对象、ktCall 对象，最后发起请求。不过，在工作中一般是不会这么写代码的，因为创建太多一次性临时对象了。我们完全可以用**链式调用**的方式来做：

```
1 private fun testAsync() {
2     KtHttpV3.create(ApiServiceV3::class.java)
3     .repos(
4         lang = "Kotlin",
5         since = "weekly"
6     ).call(object : Callback<RepoList> {
7         override fun onSuccess(data: RepoList) {
8             println(data)
9         }
10
11         override fun onFail(throwable: Throwable) {
12             println(throwable)
13         }
14     })
15 }
```


如果你没有很多编程经验，那你可能会对这种方式不太适应，但在实际写代码的过程中，你会发现这种模式写起来会比上一种舒服很多，因为**你再也不用为临时变量取名字伤脑筋了**。

总的来说，到这里的话，我们的异步请求接口就已经完成了。而且，由于我们的实际请求已经通过 **OkHttp** 派发（enqueue）到统一的线程池当中去了，并不会阻塞主线程，所以这样的代码模式执行在 **Android**、**Swing** 之类的 **UI** 编程平台，也不会引起 **UI** 界面卡死的问题。

那么，**3.0** 版本是不是到这里就结束了呢？其实并没有，因为我们还有一种情况没有考虑。我们来看看下面这段代码示例：

 复制代码

```
1 interface ApiServiceV3 {
2     @GET("/repo")
3     fun repos(
4         @Field("lang") lang: String,
5         @Field("since") since: String
6     ): KtCall<RepoList>
7
8     @GET("/repo")
9     fun reposSync(
10        @Field("lang") lang: String,
11        @Field("since") since: String
12    ): RepoList // 注意这里
13 }
14
15 private fun testSync() {
16     val api: ApiServiceV3 = KtHttpV3.create(ApiServiceV3::class.java)
17     val data: RepoList = api.reposSync(lang = "Kotlin", since = "weekly")
18     println(data)
19 }
```

请留意注释的地方，**reposSync()** 的返回值类型是 **RepoList**，而不是 **KtCall** 类型，这其实是我们 **1.0** 版本的写法。看到这，你是不是发现问题了？虽然 **KtHttp** 支持了异步请求，但原本的同步请求反而不支持了。

所以，为了让 **KtHttp** 同时支持两种请求方式，我们只需要增加一个 **if 判断** 即可：

 复制代码

```
1 private fun <T: Any> invoke(path: String, method: Method, args: Array<Any>): An
2     // 省略其他代码
3
```



```

4     return if (isKtCallReturn(method)) {
5         val genericReturnType = getTypeArgument(method)
6         KtCall<T>(call, gson, genericReturnType)
7     } else {
8         // 注意这里
9         val response = okHttpClient.newCall(request).execute()
10
11         val genericReturnType = method.genericReturnType
12         val json = response.body?.string()
13         gson.fromJson<Any?>(json, genericReturnType)
14     }
15 }
16
17 // 判断当前接口的返回值类型是不是KtCall
18 private fun isKtCallReturn(method: Method) =
19     getRawType(method.genericReturnType) == KtCall::class.java

```

在上面的代码中，我们定义了一个方法 `isKtCallReturn()`，它的作用是判断当前接口方法的返回值类型是不是 `KtCall`，如果是的话，我们就认为它是一个异步接口，这时候返回 `KtCall` 对象；如果不是，我们就认为它是同步接口。这样我们只需要将 1.0 的逻辑挪到 `else` 分支，就可以实现兼容了。

那么到这里，我们 3.0 版本的开发就算是完成了。接下来，我们进入 4.0 版本的开发。

4.0 版本：支持挂起函数

终于来到协程实战的部分了。在日常的开发工作当中，你也许经常会面临这样的问题：虽然很想用 Kotlin 的协程来简化异步开发，但公司的底层框架全部都是 `Callback` 写的，根本不支持挂起函数，我一个上层的业务开发工程师，能有什么办法呢？

其实，我们当前的 `KtHttp` 就面临着类似的问题：3.0 版本只支持 `Callback` 异步调用，现在我们想要扩展出挂起函数的功能。这其实就是大部分 Kotlin 开发者会遇到的场景。

就我这几年架构迁移的实践经验来看，针对这个问题，我们主要有两种解法：

- 第一种解法，不改动 SDK 内部的实现，直接在 SDK 的基础上扩展出协程的能力。
- 第二种解法，改动 SDK 内部，让 SDK 直接支持挂起函数。

下面我们先来看看第一种解法。至于第二种解法，其实还可以细分出好几种思路，由于它涉及到挂起函数更底层的一些知识，具体方案我会在源码篇的第 27 讲介绍。

解法一：扩展 KtCall

这种方式有一个优势，那就是我们不需要改动 3.0 版本的任何代码。这种场景在工作中也是十分常见的，比如说，项目中用到的 SDK 是开源的，或者 SDK 是公司其他部门开发的，我们无法改动 SDK。

具体的做法，就是为 KtCall 这个类扩展出一个挂起函数。

 复制代码

```
1  /*
2  注意这里          函数名称
3      ↓              ↓          */
4  suspend fun <T: Any> KtCall<T>.await(): T = TODO()
```

在上面的代码中，我们定义了一个扩展函数 `await()`。首先，它是一个挂起函数，其次，它的扩展接收者类型是 `KtCall`，其中带着一个泛型 `T`，挂起函数的返回值也是泛型 `T`。

而由于它是一个挂起函数，所以，我们的代码就可以换成这样的方式来写了。

 复制代码

```
1  fun main() = runBlocking {
2      val ktCall = KtHttpV3.create(ApiServiceV3::class.java)
3      .repos(lang = "Kotlin", since = "weekly")
4
5      val result = ktCall.await() // 调用挂起函数
6      println(result)
7  }
```

那么，现在我们就只剩下一个问题了：**`await()` 具体该如何实现？**

在这里，我们需要用到 Kotlin 官方提供的一个顶层函数：`suspendCoroutine{}`，它的函数签名是这样的：

 复制代码

```
1  public suspend inline fun <T> suspendCoroutine(crossinline block: (Continuation
2      // 省略细节
3  }
```

从它的函数签名，我们可以发现，它是一个挂起函数，也是一个高阶函数，参数类型是“(Continuation) -> Unit”，如果你还记得第 15 讲当中的内容，你应该就已经发现了，**它其实就等价于挂起函数类型！**

所以，我们可以使用 `suspendCoroutine{}` 来实现 `await()` 方法：

 复制代码

```
1  /*
2  注意这里
3      ↓
4  suspend fun <T: Any> KtCall<T>.await(): T = suspendCoroutine{
5      continuation ->
6          //      ↑
7          // 注意这里
8  }
```

如果你仔细分析这段代码的话，会发现 `suspendCoroutine{}` 的作用，其实就是**将挂起函数当中的 continuation 暴露出来**。

那么，`suspendCoroutine{}` 当中的代码具体该怎么写呢？答案应该也很明显了，当然是要用这个被暴露出来的 `continuation` 来做文章啦！

这里我们再来回顾一下 `Continuation` 这个接口：

 复制代码

```
1  public interface Continuation<in T> {
2      public val context: CoroutineContext
3      // 关键在于这个方法
4      public fun resumeWith(result: Result<T>)
5  }
```

通过定义可以看到，整个 `Continuation` 只有一个方法，那就是 `resumeWith()`，根据它的名字我们就可以推测出，它是用于“恢复”的，参数类型是 `Result`。所以很明显，这就是一个带有泛型的“结果”，它的作用就是承载协程执行的结果。

所以，综合来看，我们就可以进一步写出这样的代码了：

```

1 suspend fun <T: Any> KtCall<T>.await(): T =
2     suspendCoroutine { continuation ->
3         call(object : Callback<T> {
4             override fun onSuccess(data: T) {
5                 continuation.resumeWith(Result.success(data))
6             }
7
8             override fun onFail(throwable: Throwable) {
9                 continuation.resumeWith(Result.failure(throwable))
10            }
11        })
12    }

```

以上代码也很容易理解，当网络请求执行成功以后，我们就调用 `resumeWith()`，同时传入 `Result.success(data)`；如果请求失败，我们就传入 `Result.failure(throwable)`，将对应的异常信息传进去。

不过，也许你会觉得创建 `Result` 的写法太繁琐了，没关系，你可以借助 Kotlin 官方提供的扩展函数提升代码可读性。

```

1 suspend fun <T : Any> KtCall<T>.await(): T =
2     suspendCoroutine { continuation ->
3         call(object : Callback<T> {
4             override fun onSuccess(data: T) {
5                 continuation.resume(data)
6             }
7
8             override fun onFail(throwable: Throwable) {
9                 continuation.resumeWithException(throwable)
10            }
11        })
12    }

```

到目前为止，`await()` 这个扩展函数其实就已经实现了。这时候，如果我们在协程当中调用 `await()` 方法的话，代码是可以正常执行的。不过，这种做法其实还有一点瑕疵，那就是**不支持取消**。

让我们来写一个简单的例子：

```

1 fun main() = runBlocking {
2     val start = System.currentTimeMillis()
3     val deferred = async {
4         KtHttpV3.create(ApiServiceV3::class.java)
5             .repos(lang = "Kotlin", since = "weekly")
6             .await()
7     }
8
9     deferred.invokeOnCompletion {
10         println("invokeOnCompletion!")
11     }
12     delay(50L)
13
14     deferred.cancel()
15     println("Time cancel: ${System.currentTimeMillis() - start}")
16
17     try {
18         println(deferred.await())
19     } catch (e: Exception) {
20         println("Time exception: ${System.currentTimeMillis() - start}")
21         println("Catch exception:$e")
22     } finally {
23         println("Time total: ${System.currentTimeMillis() - start}")
24     }
25 }
26
27 suspend fun <T : Any> KtCall<T>.await(): T =
28     suspendCoroutine { continuation ->
29         call(object : Callback<T> {
30             override fun onSuccess(data: T) {
31                 println("Request success!") // ①
32                 continuation.resume(data)
33             }
34
35             override fun onFail(throwable: Throwable) {
36                 println("Request fail!: $throwable")
37                 continuation.resumeWithException(throwable)
38             }
39         })
40     }
41
42 /*
43 输出结果:
44 Time cancel: 536 // ②
45 Request success! // ③
46 invokeOnCompletion!
47 Time exception: 3612 // ④
48 Catch exception:kotlinx.coroutines.JobCancellationException: DeferredCoroutine
49 Time total: 3612
50 */

```

在 `main` 函数当中，我们在 `async` 里调用了挂起函数，接着 `50ms` 过去后，我们就去尝试取消协程。这段代码中一共有三处地方需要注意，我们来分析一下：

- 结合注释①、③一起分析，我们发现，即使调用了 `deferred.cancel()`，网络请求仍然会继续执行。根据“Catch exception:”输出的异常信息，我们也发现，当 `deferred` 被取消以后我们还去调用 `await()` 的时候，会抛出异常。
- 对比注释②、④，我们还能发现，`deferred.await()` 虽然会抛出异常，但是它却耗时 `3000ms`。虽然 `deferred` 被取消了，但是当我们调用 `await()` 的时候，它并不会马上就抛出异常，而是会等到内部的网络请求执行结束以后，才抛出异常，在此之前都会被挂起。

综上所述，当我们使用 `suspendCoroutine{}` 来实现挂起函数的时候，默认情况下是不支持取消的。那么，具体该怎么做呢？其实也很简单，就是使用 Kotlin 官方提供的另一个 API：**`suspendCancellableCoroutine{}`**。

 复制代码

```
1 suspend fun <T : Any> KtCall<T>.await(): T =
2 //          变化1
3 //          ↓
4     suspendCancellableCoroutine { continuation ->
5         val call = call(object : Callback<T> {
6             override fun onSuccess(data: T) {
7                 println("Request success!")
8                 continuation.resume(data)
9             }
10
11             override fun onFail(throwable: Throwable) {
12                 println("Request fail!: $throwable")
13                 continuation.resumeWithException(throwable)
14             }
15         })
16
17 //          变化2
18 //          ↓
19     continuation.invokeOnCancellation {
20         println("Call cancelled!")
21         call.cancel()
22     }
23 }
```

当我们使用 `suspendCancellableCoroutine{}` 的时候，可以往 `continuation` 对象上面设置一个监听：`invokeOnCancellation{}`，它代表当前的协程被取消了，这时候，我们只需要将 `OkHttp`

的 call 取消即可。

这样一来，main() 函数就能保持不变，得到的输出结果却大不相同。

 复制代码

```
1  /*
2  suspendCoroutine结果:
3
4  Time cancel: 536
5  Request success!
6  invokeOnCompletion!
7  Time exception: 3612 // ①
8  Catch exception:kotlinx.coroutines.JobCancellationException: DeferredCoroutine
9  Time total: 3612
10 */
11
12 /*
13 suspendCancellableCoroutine结果:
14
15 Call cancelled!
16 Time cancel: 464
17 invokeOnCompletion!
18 Time exception: 466 // ②
19 Catch exception:kotlinx.coroutines.JobCancellationException: DeferredCoroutine
20 Time total: 466
21 Request fail!: java.io.IOException: Canceled // ③
22 */
```

对比注释①、②，可以发现，后者是会立即响应协程取消事件的，所以当代码执行到 **deferred.await()** 的时候，会立即抛出异常，而不会挂起很长时间。另外，通过注释③这里的结果，我们也可以发现，OkHttp 的网络请求确实被取消了。

所以，我们可以得出一个结论，使用 **suspendCancellableCoroutine{}**，我们可以避免不必要的挂起，比如例子中的 **deferred.await()**；另外也可以节省计算机资源，因为这样可以避免不必要的协程任务，比如这里被成功取消的网络请求。

到这里，我们的解法一就已经完成了。这种方式并没有改动 KtHttp 的源代码，而是以扩展函数来实现的。所以，从严格意义上来讲，KtHttp 4.0 版本并没有开发完毕，等到第 27 讲我们深入理解了挂起函数的底层原理后，我们再来完成解法二的代码。

小结

这节课，我们在 KtHttp 1.0 版本的基础上，扩展出了异步请求的功能，完成了 3.0 版本的开发；接着，我们又在 3.0 版本的基础上，让 KtHttp 支持了挂起函数，这里我们是用的外部扩展的思路，并没有碰 KtHttp 内部的代码。

这里主要涉及以下几个知识点：

- 在 3.0 版本开发中，我们运用了泛型边界“T: Any”，落实对泛型的非空限制，同时通过封装 KtCall，为下一个版本打下了基础。
- 接着，在 4.0 版本中，我们借助扩展函数的特性，为 KtCall 扩展了 await() 方法。
- 在实现 await() 的过程中，我们使用了两个协程 API，分别是 suspendCoroutine{}、suspendCancellableCoroutine{}，在 Kotlin 协程当中，我们**永远都要优先使用后者**。
- suspendCancellableCoroutine{} 主要有两大优势：第一，它可以避免不必要的挂起，提升运行效率；第二，它可以避免不必要的资源浪费，改善软件的综合指标。

思考题

你能分析出下面的代码执行结果吗？为什么会是这样的结果？它能给你带来什么启发？欢迎给我留言，也欢迎你把今天的内容分享给更多的朋友。

 复制代码

```
1 fun main() = runBlocking {
2     val start = System.currentTimeMillis()
3     val deferred = async {
4         KtHttpV3.create(ApiServiceV3::class.java)
5             .repos(lang = "Kotlin", since = "weekly")
6             .await()
7     }
8
9     deferred.invokeOnCompletion {
10         println("invokeOnCompletion!")
11     }
12     delay(50L)
13
14     deferred.cancel()
15     println("Time cancel: ${System.currentTimeMillis() - start}")
16
17     try {
18         println(deferred.await())
19     } catch (e: Exception) {
20         println("Time exception: ${System.currentTimeMillis() - start}")
21         println("Catch exception:$e")
22     }
```


```

22     } finally {
23         println("Time total: ${System.currentTimeMillis() - start}")
24     }
25 }
26
27 suspend fun <T : Any> KtCall<T>.await(): T =
28     suspendCancellableCoroutine { continuation ->
29         val call = call(object : Callback<T> {
30             override fun onSuccess(data: T) {
31                 println("Request success!")
32                 continuation.resume(data)
33             }
34
35             override fun onFail(throwable: Throwable) {
36                 println("Request fail!: $throwable")
37                 continuation.resumeWithException(throwable)
38             }
39         })
40
41 // 注意这里
42 //         continuation.invokeOnCancellation {
43 //             println("Call cancelled!")
44 //             call.cancel()
45 //         }
46     }

```

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 2  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 17 | Context: 万物皆为Context?

下一篇 期中考试 | 用Kotlin实现图片处理程序

精选留言 (6)

 写留言



迹、卜懂表

Kthttp系列实战 像是简易版的retrofit2 对学习 retrofit的源码有很大帮助

作者回复: 是的, Retrofit2也是练手的绝佳案例。



神秘嘉Bin

2022-03-02

思考题:

(1) 执行async可认为一瞬间就到了suspendCancellableCoroutine的await扩展方法, 即协程被挂起。

(2) 执行deferred.cancel(), 可以使得挂起函数立刻返回并抛出协程cancel异常

(3) 协程取消了, 但网络请求还是发出去了, (因为网络请求有自己的线程) 也会回来, 调用continuation.resume, 发现协程被取消了, 抛出协程已经被取消的异常

(4.1) 网络IO比deferred.await()早, 那么deferred.await()会拿到异常, 并catch

(4.2) 网络IO比deferred.await()晚, 那么deferred.await()会立刻返回, 没有异常

以上都是我猜的, 没有实际运行 --

作者回复: 分析很好。其实, 能够直接分析出协程代码的执行流程, 并且说出具体的原因, 这也是很重要的一种能力。

共 2 条评论 >



better

2022-02-24

第一, 它可以避免不必要的挂起, 提升运行效率; 请问老师, 这一条指的是?

思考题:

网络请求还是会执行, 第一点避免了, 但是二点没有避免。

作者回复: 是的。



Allen

2022-02-23

问题二: 像 suspendCoroutine 这一类系统所提供的挂起函数底层到底实现了什么, 才使得其具有挂起的功能? 是内部自己实现了 Callback 吗? 为啥我们自己实现的 suspend 函数必须调用系统提供的挂起函数才能生效?

作者回复: 这个问题有点深，在这里说不清，等到源码篇以后，你自己就懂啦~



Allen

2022-02-23

涛哥，问两个问题哈。如果上面例子中的网络请求是运行在当前线程，是不是这里的挂起实际上也没有什么用，因为其还是会阻塞当前线程（像下面的代码一样）？

```
suspend fun testSuspendFunc() {  
    suspendCancellableCoroutine<Unit> {  
        // stimulate the network request  
        Thread.sleep(5000)  
        it.resumeWith(Result.success(Unit))  
    }  
}
```

作者回复: 是的。



Allen

2022-02-23

思考题的执行结果和 `suspendCoroutine` 的执行结果是一样的。取消了监听 `invokeOnCancellation` 的方法后，`suspendCancellableCoroutine` 和 `suspendCoroutine` 本质上是一回事。

作者回复: 你实际运行然后仔细对比看看，是不是一点差别都没有？

共 3 条评论 >

