再谈协程之Callback写出协程范儿

xuyisheng LV.5
2021年11月04日 13:54 · 阅读 2005

关注

协程的出现，颠覆了Java多年的编程风格，如果你是一个第三方库的作者，你可能想用Coroutines和Flow使你的基于Java回调的库变得更加Kotlin化、协程化。从另一方面来说，如果你是一个API消费者，你可能更愿意接入Coroutines风格的API，使其对Kotlin更友好，也让开发逻辑变得更加线性化。

今天来看下如何使用Coroutine和Flow简化API，以及如何使用suspendCancellableCoroutine和callbackFlow API构建你自己的协程风格适配器。

Callbacks

Callbacks是异步通信的一个非常常见的解决方案。事实上，大部分Java场景下，我们都使用了它们作为Java编程语言的解决方案。然而，Callbacks也有一些缺点：这种设计导致了嵌套的回调，最终导致了难以理解的代码，另外，异常处理也比较复杂。

在Kotlin中，你可以使用Coroutines简化调用Callbacks，但为此你需要建立自己的适配器，将旧的Callback转化为Kotlin风格的协程。

构建Adapter

在协程中，Kotlin提供了suspendCancellableCoroutine来适配One-shot回调，同时提供了callbackFlow来适配数据流场景下的回调。

下面的场景中，将用一个简单的Callbacks例子来演示下这种转换。

假设我们有一个「NetAPI.getData」的函数，返回一个Data Callback，在协程场景下，我们想让它返回一个suspend函数。

所以，我们给NetAPI设计一个拓展函数，用来返回Location的suspend函数，如下所示。

kotlin 复制代码

```kotlin
suspend fun NetAPI.awaitGetData(): Data
```

由于这是一个One-shot的异步操作，我们使用可以suspendCancellableCoroutine函数，suspendCancellableCoroutine执行作为参数传递给它的代码块，然后暂停当前Coroutine的执行，同时等待继续执行的信号。当Coroutine的Continuation对象中的resume或resumeWithException方法被调用时，Coroutine将恢复执行。

kotlin 复制代码

```kotlin
// NetAPI的拓展函数，用于返回Data
suspend fun NetAPI.awaitGetData(): Data =

    // 创建一个可以cancelled  suspendCancellableCoroutine
    suspendCancellableCoroutine<Data> { continuation ->

        val callback = object : NetCallback {
            override fun success(data: Data) {
                // Resume coroutine 同时返回Data
                continuation.resume(data)
            }

            override fun error(e: String) {
                // Resume the coroutine
                continuation.resumeWithException(e)
            }
        }
        addListener(callback)
        // 结束suspendCancellableCoroutine块的执行，直到在任一回调中调用continuation参数
    }
```

> 要注意的是：Coroutines库中也能找到suspendCancellableCoroutine的不可取消版本（即suspendCoroutine），但最好总是选择suspendCancellableCoroutine来处理Coroutine Scope的取消。

suspendCancellableCoroutine背后的原理

从内部实现来说，suspendCancellableCoroutine使用suspendCoroutineUninterceptedOrReturn来获取suspend函数中Coroutine的Continuation。这个Continuation对象被一个CancellableContinuation拦截，它可以用来控制当前Coroutine的生命周期。

在这之后，传递给suspendCancellableCoroutine的lambda将被执行，如果lambda返回一个结果，Coroutine将立即恢复，或者将被暂停，直到CancellableContinuation从lambda中手动进行恢复。

源码如下所示。

kotlin 复制代码

```kotlin
public suspend inline fun <T> suspendCancellableCoroutine(
  crossinline block: (CancellableContinuation<T>) -> Unit
): T =
  // Get the Continuation object of the coroutine that it's running this suspend functi
  suspendCoroutineUninterceptedOrReturn { uCont ->

    // Take over the control of the coroutine. The Continuation's been
    // intercepted and it follows the CancellableContinuationImpl lifecycle now
    val cancellable = CancellableContinuationImpl(uCont.intercepted(), ...)
    /* ... */

    // Call block of code with the cancellable continuation
    block(cancellable)

    // Either suspend the coroutine and wait for the Continuation to be resumed
    // manually in `block` or return a result if `block` has finished executing
    cancellable.getResult()
  }
```

如果我们想获取多个数据流（使用NetAPI.getDataList函数），我们就需要使用Flow创建一个数据流。理想的API应该是这样的。

kotlin 复制代码

```kotlin
fun NetAPI.getDataListFlow(): Flow<Data>
```

要将基于回调的流媒体API转换为Flow，我们需要使用创建Flow的callbackFlow构建器。在callbackFlow lambda中，我们处于Coroutine的上下文中，因此，可以调用suspend函数。与flow构建器不同，callbackFlow允许通过send函数从不同CoroutineContext发出值，或者通过offer函数在协程外发出值。

通常情况下，使用callbackFlow的流适配器遵循这三个通用步骤。

- 创建回调，使用offer将元素添加到流中。
- 注册该回调。
- 等待消费者取消循环程序并取消对回调的注册。

示例代码如下所示。

kotlin 复制代码

```kotlin
// 向consumer发送Data updates
fun NetAPI.getDataListFlow() = callbackFlow<Data> {
  // 当前会在一个协程作用域中创建一个新的Flow

  // 1. 创建回调，使用offer将元素添加到流中
  val callback = object : NetCallback() {
    override fun success(result: Result?) {
      result ?: return // Ignore null responses
      for (data in result.datas) {
        try {
          offer(data) // 将元素添加至flow
        } catch (t: Throwable) {
          // 异常处理
        }
      }
    }
  }

  // 2. 注册该回调，从而获取数据流
  requestDataUpdates(callback).addOnFailureListener { e ->
    close(e) // 异常时close
  }

  // 3. 等待消费者取消循环程序并取消对回调的注册，这样会suspend当前协程，直到这个flow被关闭
  awaitClose {
    // 移除监听
    removeLocationUpdates(callback)
  }
}
```

callbackFlow背后的原理

在协程内部，callbackFlow会使用channel，它在概念上与阻塞队列非常相似。channel都有容量配置，限定了可缓冲元素数的上限。

在callbackFlow中所创建channel的默认容量为64个元素，当你尝试向已经满的channel添加新元素时，send函数会将数据提供方挂起，直到新元素有空间能加入channel为止，而offer不会将相关元素添加到channel中，并会立即返回false。

awaitClose背后的原理

awaitClose的实现原理其实和suspendCancellableCoroutine是一样的，参考下下面的代码中的注释。

kotlin 复制代码

```kotlin
public suspend fun ProducerScope<*>.awaitClose(block: () -> Unit = {}) {
  ...
  try {
    // Suspend the coroutine with a cancellable continuation
    suspendCancellableCoroutine<Unit> { cont ->
      // Suspend forever and resume the coroutine successfully only
      // when the Flow/Channel is closed
      invokeOnClose { cont.resume(Unit) }
    }
  } finally {
    // Always execute caller's clean up code
    block()
  }
}
```

有啥用?

将基于回调的API转换为数据流，这玩意儿到底有什么用呢？我们拿最常用的View.setOnClickListener来看下，它既可以看作是一个One-shot的场景，也可以看作是数据流的场景。

我们先把它改写成suspendCancellableCoroutine形式，代码如下所示。

kotlin 复制代码

```kotlin
suspend fun View.awaitClick(block: () -> Unit): View = suspendCancellableCoroutine { co
    setOnClickListener { view ->
        if (view == null) {
            continuation.resumeWithException(Exception("error"))
        } else {
            block()
            continuation.resume(view)
        }
    }
}

使用：
lifecycleScope.launch {
    binding.test.awaitClick {
        Toast.makeText(this@MainActivity, "loading", Toast.LENGTH_LONG).show()
    }
}
```

嗯，有点一言难尽的感觉，就差脱裤子放屁了。我们再把它改成数据流的场景。

kotlin 复制代码

```kotlin
fun View.clickFlow(): Flow<View> {
    return callbackFlow {
        setOnClickListener {
            trySend(it) // offer函数被Deprecated了，使用trySend替代
        }
        awaitClose { setOnClickListener(null) }
    }
}

使用：
lifecycleScope.launch {
    binding.test.clickFlow().collect {
        Toast.makeText(this@MainActivity, "loading", Toast.LENGTH_LONG).show()
    }
}
```

好了，屁是完全放出来了。

可以发现，这种场景下，强行硬套这种模式，其实并没有什么卵用，反而会让别人觉得你是个智障。

那么到底什么场景需要使用呢？我们可以想想，为什么需要Callbback。

大部分Callback hell的场景，都是异步请求，也就是带阻塞的那种，或者就是数据流式的数据产出，所以这种仅仅是调用个闭包的回调，其实不能叫回调，它只是一个lambda，所以，我们再来看一个例子。

现在有一个TextView，显示来自一个Edittext的输入内容。这样一个场景就是一个明确的数据流场景，主要是利用Edittext的TextWatcher中的afterTextChanged回调，我们将它改写成Flow形式，代码如下所示。

kotlin 复制代码

```kotlin
fun EditText.afterTextChangedFlow(): Flow<Editable?> {
    return callbackFlow {
        val watcher = object : TextWatcher {
            override fun afterTextChanged(s: Editable?) {
                trySend(s)
            }

            override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, af

            override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count
        }
        addTextChangedListener(watcher)
        awaitClose { removeTextChangedListener(watcher) }
    }
}

使用:
lifecycleScope.launch {
    with(binding) {
        test.afterTextChangedFlow().collect { show.text = it }
    }
}
```

有点意思了，我没写回调，但是也拿到了数据流，嗯，其实有点「强行可以」的感觉。

但是，一旦这里变成了Flow，这就变得很有味道了，这可是Flow啊，我们可以利用Flow那么多的操作符，做很多有意思的事情了。

举个例子，我们可以对输入框做限流，这个场景很常见，例如搜索，用户输入的内容会自动搜索，但是又不能一输入内容就搜索，这样会产生大量的无效搜索内容，所以，这个场景也有个专有名词——输入框防抖。

之前在处理类似的需求时，大部分都是采用RxJava的方式，但现在，我们有了Flow，可以在满足协程范API的场景下，依然完成这个功能。

我们增加一下debounce即可。

```scss
lifecycleScope.launch {
    with(binding) {
        test.afterTextChangedFlow()
            .buffer(Channel.CONFLATED)
            .debounce(300)
            .collect {
                show.text = it
                // 来点业务处理
                viewModel.getSearchResult(it)
            }
    }
}
```

甚至你还可以增加一个背压策略，再来个debounce，在流停止后，完成数据收集。

> 当然你还可以把buffer和debounce直接写到afterTextChangedFlow返回的Flow中，作为当前场景的默认处理。

参考资料：

medium.com/androiddeve…

向大家推荐下我的网站 xuyisheng.top/ 专注 Android-Kotlin-Flutter 欢迎大家访问

分类： Android 标签： Android