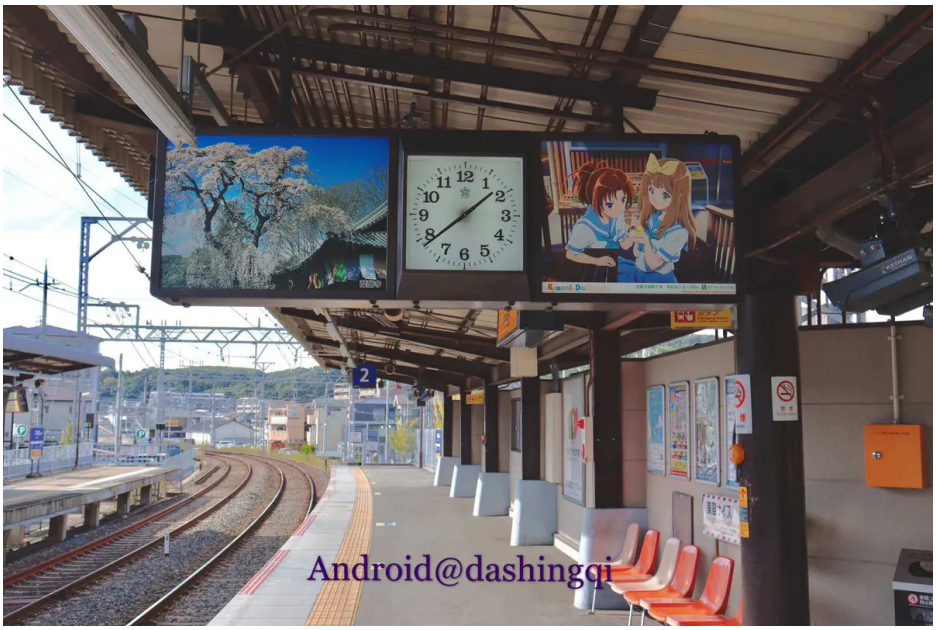
Kotlin协程 ----- suspendCoroutine和suspendCancellableCoroutine的使用

dashingqi 关注
1 2021.06.26 18:23:52 字数 647 阅读 3,726



Android_Banner.jpg

简介

- suspendCoroutine 的使用
- suspendCancellableCoroutine的使用
- Retrofit是如何支持协程的

suspendCoroutine 的使用

这里我们将使用suspendCoroutine将单一方法的接口方法改造成具有返回值的方法

单一方法的回调

声明一个单一方法的接口

```
/**
 * @author : zhangqi
 * @time : 6/22/21
 * desc : 单一方法接口
 */
interface SingleMethodCallback {

    fun onCallBack(value: String)
}

```

接着模拟一个耗时的操作，当操作完毕我们把结果回调给实现类

```
  /**
   * 模拟一个耗时操作
   */
private fun runTask(callback: SingleMethodCallback) {
```

```
        thread {
            Thread.sleep(1000)
            callback.onCallBack("result")
        }
    }
```

最后我们调用runTask方法，传入SingleMethodCallback的实现

```
private fun runTaskDefault() {
        runTask(object : SingleMethodCallback {
            override fun onCallBack(value: String) {
                TODO("Not yet implemented")
            }
        })
    }
```

接着我们使用Kotlin协程提供的 *suspendCoroutine* 让runTaskDefault具有返回值；

改造一下runTaskDefault ---> runTaskWithSuspend

```
 suspend fun runTaskWithSuspend(): String {
        // suspendCoroutine是一个挂起函数
        return suspendCoroutine { continuation ->
            runTask(object : SingleMethodCallback {
                override fun onCallBack(value: String) {
                    continuation.resume(value)
                }
            })
        }
    }
```

这里 suspendCoroutine是一个挂起函数，挂起函数只能在协程或者其他挂起函数中被调用，同时我们在回调中将结果值传入到Coutination的resume方法中；

经过我们上述的操作将回调方法具有返回值了；

suspendCancellableCoroutine 的使用

Success And Failure 类别的接口

声明 success and failure 类型的接口

```
/**
 * @author : zhangqi
 * @time : 6/22/21
 * desc :
 */
interface ICallBack {
    fun onSuccess(data: String)
    fun onFailure(t: Throwable)
}
```

同样我们模拟一个耗时操作，在获取结果的时候 调用 onSuccess()将结果回调给实现，出现错误调用onFailure将错误交给实现处理

```
 /**
   * 模拟一个耗时操作
   */
 private fun request(callback: ICallBack) {
   thread {
     try {
       callback.onSuccess("success")
     } catch (e: Exception) {
       callback.onFailure(e)
     }
   }
 }
```

最后我们调用requet方法，传入接口的实现，

```
private fun requestDefault() {
  request(object : ICallBack {
    override fun onSuccess(data: String) {
      // doSomething
    }

    override fun onFailure(t: Throwable) {
      // handle Exception
    }

  })
}
```

同样我们使用Kotlin协程提供的挂起函数将 requestDefault()改造成 具有返回值的函数 requestWithSuspend()

只不过我们这里使用了 *suspendCancellablkeCoroutine* ,代码上见吧！

```
 private suspend fun requestWithSuspend(): String {
        return suspendCancellableCoroutine { cancellableContinuation->
            request(object : ICallBack {
                override fun onSuccess(data: String) {
                    cancellableContinuation.resume(data)
                }

                override fun onFailure(t: Throwable) {
                    cancellableContinuation.resumeWithException(t)
                }
            })
        }
    }
```

suspendCancellableCoroutine 是一个挂起函数，我们将requestWithSuspend声明称挂起函数

在onSucess()中我们我们调用CancellableContinue # resume 方法将结果返回，在onFailure调用CancellableContinuation # resumeWithException 将异常传入进去；

调用requestWithSuspend()

```
private fun runRequestSuspend() {
  try {
    viewModelScope.launch {
      val value = requestWithSuspend()
    }
  } catch (e: Exception) {
    e.printStackTrace()
  }
}
```

在ViewModel中Kotlin协程提供了 viewModelScope 来开启一个协程，改协程是具有声明周期的与当前ViewModel保持一致；

这里我们使用了try{}catch 将我们开启的协程处理了下，调用成功获取到value值，出现错误我们在catch块中除了一下；

以上就是 我们两种日常遇见频率较高的情况进行的改造（回调方法具有返回值）

Retrofit是如何支持协程的

Retrofit是在2.6版本开始支持，我们先对比下使用协程前后的区别

使用协前

```
/**
  * 发现页面的数据
```

```
  */
@GET("/api/v7/index/tab/discovery")
fun getDiscoveryData(): Call<OpenEyeResponse>

// 在ViewModel中调用
 /**
   * 没有使用协程做网络请求
   */
    fun getDiscoverData() {
      WidgetService.openEyeInstance.getDiscoveryData().enqueue(object : Callback<OpenE
        override fun onResponse(call: Call<OpenEyeResponse>, response: Response<OpenEy
          var body = response.body()
        }

        override fun onFailure(call: Call<OpenEyeResponse>, t: Throwable) {
        }
      })
    }
```

接着我们看下使用协程后

使用协程后

```
/**
  * 通过协程做本次请求
  * @return OpenEyeResponse
  */
@GET("/api/v7/index/tab/discovery")
suspend fun getDiscoveryDataCoroutine(): OpenEyeResponse

  /**
   * 使用协程做的请求
   */
fun getDiscoverDataWithCoroutine() {
  try {
    viewModelScope.launch {
      var discoveryDataCoroutine = WidgetService.openEyeInstance.getDiscoveryDataCorou
    }
  } catch (e: Exception) {
  }
}
```

可以看见，在接口类中声明的方法声明为挂起函数，同时我们可以将我们想要的数据结构直接返回不用Call包一层；

Retrofit支持协程

Retrofit # HttpServiceMethod

```
 okhttp3.Call.Factory callFactory = retrofit.callFactory;
    if (!isKotlinSuspendFunction) {
      return new CallAdapted<>(requestFactory, callFactory, responseConverter, callAda
      // 当是直接返回数据结构走这里
    } else if (continuationWantsResponse) {
      //noinspection unchecked Kotlin compiler guarantees ReturnT to be Object.
      return (HttpServiceMethod<ResponseT, ReturnT>)
            // 执行了 SuspendForResponse
          new SuspendForResponse<>(
              requestFactory,
              callFactory,
              responseConverter,
              (CallAdapter<ResponseT, Call<ResponseT>>) callAdapter);
    } else {
      //noinspection unchecked Kotlin compiler guarantees ReturnT to be Object.
      return (HttpServiceMethod<ResponseT, ReturnT>)
          new SuspendForBody<>(
              requestFactory,
              callFactory,
              responseConverter,
              (CallAdapter<ResponseT, Call<ResponseT>>) callAdapter,
              continuationBodyNullable);
    }
```

SuspendForResponse ---> KotlinExtensions.awaitResponse

```
SuspendForResponse(
        RequestFactory requestFactory,
        okhttp3.Call.Factory callFactory,
        Converter<ResponseBody, ResponseT> responseConverter,
        CallAdapter<ResponseT, Call<ResponseT>> callAdapter) {
      super(requestFactory, callFactory, responseConverter);
      this.callAdapter = callAdapter;
    }

    @Override
    protected Object adapt(Call<ResponseT> call, Object[] args) {
      call = callAdapter.adapt(call);
      //noinspection unchecked Checked by reflection inside RequestFactory.
      Continuation<Response<ResponseT>> continuation =
          (Continuation<Response<ResponseT>>) args[args.length - 1];

      // See SuspendForBody for explanation about this try/catch.
      try {
        // 在这里直接调用了 KotlinExtensions.awaitResponse
        return KotlinExtensions.awaitResponse(call, continuation);
      } catch (Exception e) {
        return KotlinExtensions.suspendAndThrow(e, continuation);
      }
    }
```

KotlinExtensions.awaitResponse

```
suspend fun <T> Call<T>.awaitResponse(): Response<T> {
  // 在这里使用了suspendCancellableCoroutine
  return suspendCancellableCoroutine { continuation ->
     // 当我们开启的协程开启了之后，会回调到这个方法
     // 取消当前的请求
    continuation.invokeOnCancellation {
      cancel()
    }
    enqueue(object : Callback<T> {
      override fun onResponse(call: Call<T>, response: Response<T>) {
        // 当成功拿到response之后 将response返回
        continuation.resume(response)
      }

      override fun onFailure(call: Call<T>, t: Throwable) {
        // 失败的话 直接将异常抛出
        continuation.resumeWithException(t)
      }
    })
  }
}
```