

Kotlin | 关于协程异常处理，你想知道的都在这

Petterp 郭霖 2022-08-17 08:00 发表于江苏



点击上方蓝字即可关注
关注后可查看所有经典文章

/ 今日科技快讯 /

近日，特斯拉CEO埃隆·马斯克旗下太空探索技术公司SpaceX日前表示，欢迎研究人员以“非破坏性”的方式入侵其“星链”卫星网络。如果研究人员发现该服务中的某些漏洞，最多可获得高达25000美元的费用。

/ 作者简介 /

本篇文章来自Petterp的投稿，文章主要分享了关于协程异常处理的相关内容，相信会对大家有所帮助！同时也感谢作者贡献的精彩文章。

Petterp的博客地址：

<https://juejin.cn/user/3491704662136541>

/ 引言 /

关于协程的异常处理，一直以来都不是一个简单问题。因为涉及到了很多方面，包括 异常的传递，结构化并发下的异常处理，异常的传播方式，不同的Job 等，所以常常让很多(特别是刚使用协程的，也不乏老手)同学摸不着头脑。常见有如下两种处理方式：

- try catch
- CoroutineExceptionHandler

但这两种方式(特别是第二种)到底该什么时候用，用在哪里，却是一个问题？

比如虽然知道 `CoroutineExceptionHandler`，但为什么增加了却还是崩溃？到底应该加在哪里？尝试半天发现无解，最终又只能直接 `try catch`，粗暴并有效，最终遇到此类问题，直接下意识 `try` 住。

`try catch` 虽然直接，一定程度上也帮我们规避了很多使用方面的问题，但同时也埋下了很多坑，也就是说，并不是所有协程的异常都可以 `try` 住(取决于使用位置)，其也不是任何场景的最优解。

鉴于此，本篇将从头到尾，帮助你理清以下问题：

- 什么是结构化并发？
- 协程的异常传播流程与形式
- 协程的异常处理方式
- 为什么有些异常处理了却还是崩了
- `SupervisorJob` 的使用场景
- `supervisorScope` 与 `coroutineScope`
- 异常处理方式的场景推荐

本文尽可能会用大白话与你分享理解，如有遗漏或理解不当，也欢迎评论区反馈。

好了，让我们开始吧！

/ 结构化并发 /

在最开始前，我们先搞清楚什么是结构化并发，这对我们理解协程异常的传递将非常有帮助。

让我们先将思路转为日常业务开发中，比如在某某业务中，可能存在好几个需要同时处理的逻辑，比如同时请求两个网络接口，同时操作两个子任务等。我们暂且称上述学术化概念为多个并发操作。

而每个并发操作其实都是在处理一个单独的任务，这个任务中可能还存在子任务。同样对于这个子任务来说，它又是其父任务的子单元。每个任务都有自己的生命周期，子任务的生命周期会继承父任务的生命周期，比如如果父任务关闭，子任务也会被取消。而如果满足这样特性，我们就称其就是 结构化并发。

在协程中，我们常用的 `CoroutineScope`，正是基于这样的特性，即其也有自己的作用域与层级概念。

比如当我们每次调用其扩展方法 `launch()` 时，这个内部又是一个新的协程作用域，新的作用域又会与父协程保持着层级关系，当我们取消 `CoroutineScope` 时，其所有子协程也都会被关闭。

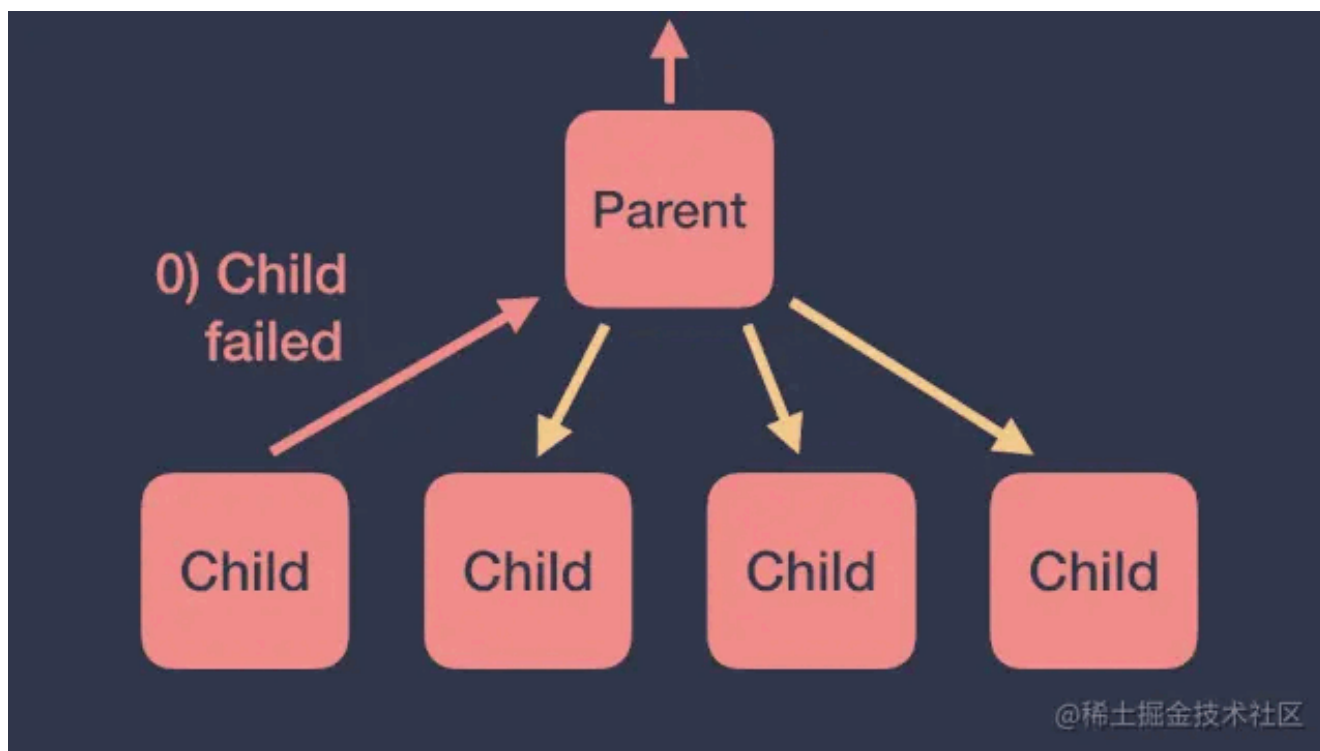
如下代码片段：

```
val scope = CoroutineScope(Job())
val jobA = scope.launch(CoroutineName("A")) {
    val jobChildA = launch(CoroutineName("child-A")) {
        delay(1000)
        println("xxx")
    }
    // jobChildA.cancel()
}
val jobB = scope.launch(CoroutineName("B")) {
    delay(500)
    println("xxx")
}
// scope.cancel()
```

我们定义了一个名为 `scope` 的作用域，其中有两个子协程 `jobA`，`B`，同时 `jobA` 又有一个子协程 `jobChildA`。如果我们要取消`jobB`，并不会影响`jobA`，其依然会继续执行；但如果我们要取消整个作用域时 `scope.cancel()`，`jobA`，`jobB` 都会被取消，相应 `jobA` 被取消时，因为其也有自己的作用域，所以 `jobChildA` 也会被取消，以此类推。而这就是协程的结构化并发特性。

/ 异常传播流程 /

默认情况下，任意一个协程发生异常时都会影响到整个协程树，而异常的传递通常是双向的，也即协程会向子协程与父协程共同传递，如下方所示：



整体流程如下：

- 先 cancel 子协程
- 取消自己
- 将异常传递给父协程
- (重复上述过程，直到根协程关闭)

举个例子，比如下面这段代码：

```
fun main() = runBlocking { this: CoroutineScope
    launch { this: CoroutineScope
        println("子协程A开始")
        delay( timeMillis: 10)
        println("子协程A抛出异常")
        throw NullPointerException()
    }
    launch { this: CoroutineScope
        println("子协程B开始")
        delay( timeMillis: 30)
        println("子协程B结束")
    }
    delay( timeMillis: 100)
    println("父协程：我还没打印呢--")
}
```

"/Applications/Android Studio.app/Contents/jre/Contents/Home/bin/java" ...

子协程A开始

子协程B开始

子协程A抛出异常

Exception in thread "main" java.lang.NullPointerException: Create breakpoint

at com.petterp.testscope.TestKt\$main\$1.invokeSuspend(Test.kt:12)

at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)

at kotlinx.coroutines.DispatchedTaskKt.resume(DispatchedTask.kt:234)

at kotlinx.coroutines.DispatchedTaskKt.dispatch(DispatchedTask.kt:166)

at kotlinx.coroutines.CancellableContinuationImpl.dispatchResume(CancellableContinuationImpl.kt:508)

at kotlinx.coroutines.CancellableContinuationImpl.resumeImpl(CancellableContinuationImpl.kt:524)

at kotlinx.coroutines.CancellableContinuationImpl.resumeImpl\$default(CancellableContinuationImpl.kt:524)

at kotlinx.coroutines.CancellableContinuationImpl.resumeUndispatched(CancellableContinuationImpl.kt:538)

at kotlinx.coroutines.EventLoopImplBase\$DelayedResumeTask.run(EventLoop.common.kt:508)

at kotlinx.coroutines.EventLoopImplBase.processNextEvent(EventLoop.common.kt:284)

at kotlinx.coroutines.BlockingCoroutine.joinBlocking(Builders.kt:85)

at kotlinx.coroutines.BuildersKt__Builders_impl__blockingCoroutineBuilder\$1.invoke(Builders.kt:103)

@稀土掘金技术社区

在上图中，我们创建了两个子协程A和B，并在A中抛出异常，查看结果如右图所示，当子协程A异常被终止时，我们的子协程B与父协程都受到影响被终止。

当然如果不想在协程异常时，同级别子协程或者父协程受到影响，此时就可以使用 `SupervisorJob`，这个我们放在下面再谈。

/ 异常传播形式 /

在协程中，异常的传播形式有两种，一种是自动传播(`launch` 或 `actor`)，一种是向用户暴露该异常(`async` 或 `produce`)，这两种的区别在于，前者的异常传递过程是层层向上传递（如果异常没有被捕获），而后者将不会向上传递，会在调用处直接暴露。

记住上述思路对我们处理协程的异常将会很有帮助。

/ 异常处理方式 /

`tryCatch`

一般而言, `tryCatch` 是我们最常见的处理异常方式，如下所示：

```
fun main() = runBlocking {
    launch {
        try {
            throw NullPointerException()
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }
    println("嘿害哈")
}
```

当异常发生时，我们底部的输出依然能正常打印，这也不难理解，就像我们在 `Android` 或者 `Java` 中的使用一样。但有些时候这种方式并不一定能有效，我们在下面中会专门提到。但大多数情况下，`tryCatch` 依然如万金油一般，稳定且可靠。

CoroutineExceptionHandler

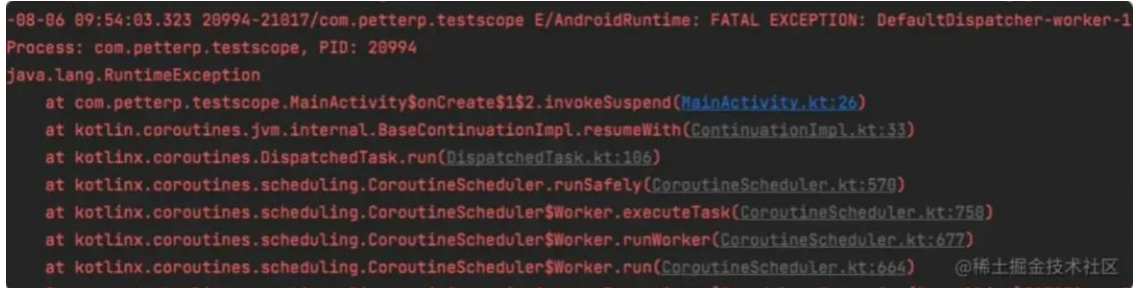
其是用于在协程中全局捕获异常行为的最后一种机制，你可以理解为类似 `Thread.uncaughtExceptionHandler` 一样。

但需要注意的是，`CoroutineExceptionHandler` 仅在未捕获的异常上调用，也即这个异常没有任何方式处理时(比如在源头`tryCatch`了)，由于协程是结构化的，当子协程发生异常时，它会优先将异常委托给父协程区处理，以此类推。直到根协程作用域或者顶级协程。因此其永远不会使用我们子协程 `CoroutineContext` 传递的

`CoroutineExceptionHandler(SupervisorJob 除外)`，对于 `async` 这种，而是直接向用户直接暴露该异常，所以我们在具体调用处直接处理就行。

如下示例所示：

```
val scope = CoroutineScope(Job())
scope.launch() {
    launch(CoroutineExceptionHandler { _, _ -> }) {
        delay(10)
        throw RuntimeException()
    }
}
```



```
-08-06 09:54:03.323 28994-21817/com.petterp.testscope E/AndroidRuntime: FATAL EXCEPTION: DefaultDispatcher-worker-1
Process: com.petterp.testscope, PID: 28994
java.lang.RuntimeException
    at com.petterp.testscope.MainActivity$onCreate$1$2.invokeSuspend(MainActivity.kt:26)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:186)
    at kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(CoroutineScheduler.kt:579)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTask(CoroutineScheduler.kt:756)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(CoroutineScheduler.kt:677)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(CoroutineScheduler.kt:664) @稀土掘金技术社区
```

不难发现异常了，原因就是我们的 `CoroutineExceptionHandler` 位置不是根协程或者 `CoroutineScope` 初始化时。

如果我们改成下述方式，就可以正常处理该异常：

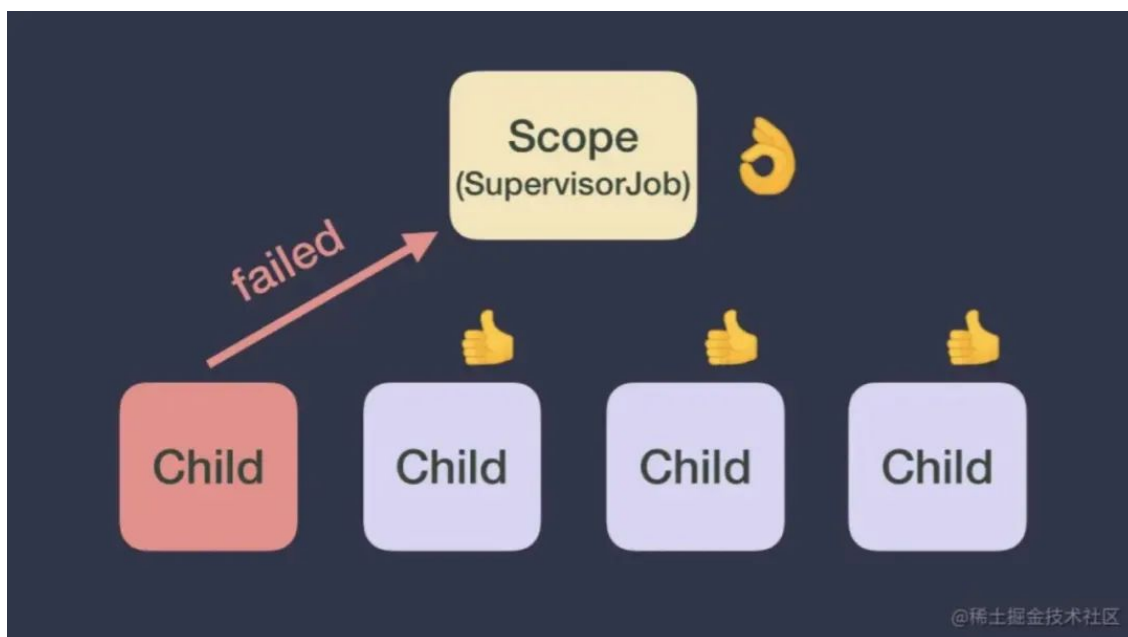
```
// 1. 初始化scope时
```

```
val scope = CoroutineScope(Job() + CoroutineExceptionHandler { _, _ -> })

// 2. 根协程
scope.launch(CoroutineExceptionHandler { _, _ -> }) { }
```

SupervisorJob

`supervisorJob` 是一个特殊的Job，其会改变异常的传递方式，当使用它时，我们子协程的失败不会影响到其他子协程与父协程，通俗点理解就是子协程会自己处理异常，并不会影响其兄弟协程或者父协程。如下图所示：



举个简单的例子：

```
val scope = CoroutineScope(SupervisorJob() + CoroutineExceptionHandler { _, _ -> })
scope.launch(CoroutineName("A")) {
    delay(10)
    throw RuntimeException()
}
scope.launch(CoroutineName("B")) {
    delay(100)
    Log.e("petterp", "正常执行,我不会收到影响")
}
```

当协程A失败时，协程B依然可以正常打印。

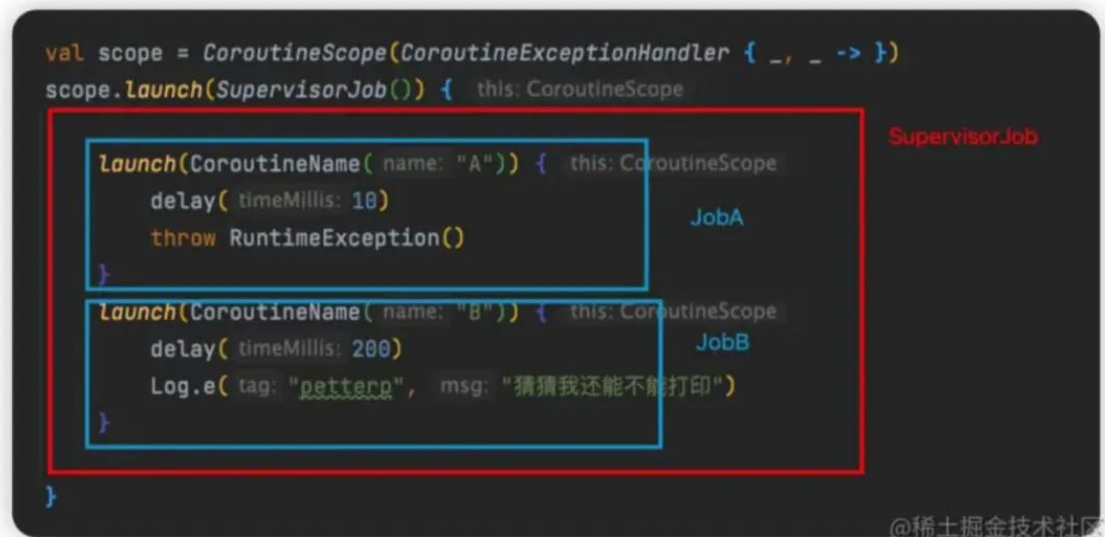
如果我们将上述的示例改一下，会发生什么情况？如下所示：

```
val scope = CoroutineScope(CoroutineExceptionHandler { _, _ -> {} })
scope.launch(SupervisorJob()) {
    launch(CoroutineName("A")) {
        delay(10)
        throw RuntimeException()
    }
    launch(CoroutineName("B")) {
        delay(100)
        Log.e("petterp", "正常执行,我不会收到影响")
    }
}
```

猜一猜B协程内部的log能否正常打印？

结果是不能

为什么？我不是已经使用了 SupervisorJob() 吗？我们用一张图来看一下：



如上图所示，我们在 scope.launch 时传递了 SupervisorJob，看着似乎没什么问题🤔，我们期望的是 SupervisorJob 也会传递到子协程。但实则不会，因为子协程在 launch 时会创建新的协程作用域，其会使用默认新的 Job 替代我们传递 SupervisorJob，所以导致我们传递的 SupervisorJob 被覆盖。所以如果我们想让子协程不影响父协程或者其他子协程，此时就必须再显示添加 SupervisorJob。

正确的打开方式如下所示：

```
scope.launch {
    launch(CoroutineName("A") + SupervisorJob()) {
        delay(10)
        throw RuntimeException()
    }
    launch(CoroutineName("B")) {
        delay(200)
        Log.e("petterp", "猜猜我还能不能打印")
    }
}
```

总结如下：

SupervisorJob 可以用来改变我们的协程异常传递方式，从而让子协程自行处理异常。但需要注意的是，因为协程具有结构化的特点，SupervisorJob 仅只能用于同一级别的子协程。如果我们在初始化 scope 时添加了 SupervisorJob，那么整个scope对应的所有根协程都将默认携带 SupervisorJob，否则就必须在 CoroutineContext 显示携带 SupervisorJob。

/ 小测试 /

try不住的异常

如下的代码，可以 try 住吗？

```
val scope = CoroutineScope(Job())
try {
    //A
    scope.launch {
        throw NullPointerException()
    }
} catch (e: Exception) {
    e.printStackTrace()
}
```

答案是不会

为什么呢？我不是已经在 A 外面try了吗？

默认情况下，如果异常没有被处理，而且顶级协程 CoroutineContext 中没有携带 CoroutineExceptionHandler，则异常会传递给默认线程的 ExceptionHandler。在 Android 中，如果没有设置 Thread.setDefaultUncaughtExceptionHandler，这个异常将立即被抛出，从而导致引发App崩溃。

我们在 launch 时，因为启动了一个新的协程作用域，而新的作用域内部已经是新的线程(可以理解为)，因为内部发生异常时因为没有被直接捕获，再加上其Job不是 SupervisorJob，所以异常将向上开始传递，因为其本身已经是根协程，此时根协程的 CoroutineContext 也没有携带 CoroutineExceptionHandler，从而导致了直接异常。

CoroutinexxHandler 不生效？

下列代码中，添加的 CoroutineExceptionHandler 会生效吗？

```
val scope = CoroutineScope(Job())
scope.launch {
    val asyncA = async(SupervisorJob()+CoroutineExceptionHandler { _, _ -> }) {
        throw RuntimeException()
    }
    val asyncB = async(SupervisorJob()+CoroutineExceptionHandler { _, _ -> }) {
        throw RuntimeException()
    }
    asyncA.await()
    asyncB.await()
}
```

答案是：不会生效

Tips：如果你不是很理解 async 的 CoroutineContext 里此时为什么要加 SupervisorJob，请看下面，会再做解释。

你可能会想，这还不简单吗，上面不是已经提过了，如果根协程或者scope中没有设置 CoroutineExceptionHandler，异常会被直接抛出，所以这里肯定异常了啊。

如果你这样想了，恭喜回答正确~ 🤞

那该怎么改一下上述示例呢？

scope 初始化时或者根协程里加上 `CoroutineExceptionHandler`，或者直接 `async` 里面 `try catch` 都可以。那还有没有其他方式呢？

此处停留10s思考，loading.....

如果你还记得我们最开始说过的异常的传播形式，就会知道，对于 `async` 这种在其异常时，其会主动向用户暴露，而不是优先向上传递。

也就是说，我们直接可以在 `await()` 时 `try Catch`。代码如下：

```
scope.launch {
    val asyncA = async(SupervisorJob()){
    val asyncB = async(SupervisorJob()){

        val resultA = kotlin.runCatching { asyncA.await() }
        val resultB = kotlin.runCatching { asyncB.await() }
    }
}
```

`runCatching` 是 `kotlin` 中对于 `tryCatch` 的一种包装，其会将结果使用 `Result` 类进行包装，从而让我们能更直观的处理结果，从而更加符合 `kotlin` 的语法习惯。

Tips

为什么上述 `async` 里要添加 `SupervisorJob()`，这里再做一个解释。

```
val scope = CoroutineScope(Job())
scope.launch {
    val asyncA = async(SupervisorJob()) { throw RuntimeException() }
    val asyncB = async xxx
}
```

因为 `async` 时内部也是新的作用域，如果 `async` 对应的是根协程，那么我们可以在 `await()` 时直接捕获异常。怎么理解呢？

如下示例：

```
val scope = CoroutineScope(Job())
// async 作为根协程
val asyncA = scope.async { throw NullPointerException() }
val asyncB = scope.async { }
scope.launch {
    // 此时可以直接tryCatch
    kotlin.runCatching {
        asyncA.await()
        asyncB.await()
    }
}
```

但如果 `async` 其对应的不是根协程(即不是 `scope`直接.`async`),则会先将异常传递给父协程，从而导致异常没有在调用处暴露，我们的`tryCatch` 自然也就无法拦截。如果此时我们为其增加 `SupervisorJob()` ,则标志着其不会主动传递异常，而是由该协程自行处理。所以我们可以调用处(`await()`) 捕获。

/ 相关扩展 /

supervisorScope

官方解释如下：

使用 `SupervisorJob` 创建一个 `CoroutineScope` 并使用此范围调用指定的挂起块。提供的作用域从外部作用域继承其`coroutineContext` , 但用 `SupervisorJob` 覆盖上下文的 `Job` 。一旦给定块及其所有子协程完成，此函数就会返回。

通俗点就是，我们帮你创建了一个 `CoroutineScope` , 初始化作用域时，使用 `SupervisorJob` 替代默认的`Job` , 然后将其的作用域扩展至外部调用。如下代码所示：

```
val scope = CoroutineScope(CoroutineExceptionHandler { _, _ -> })
```

```

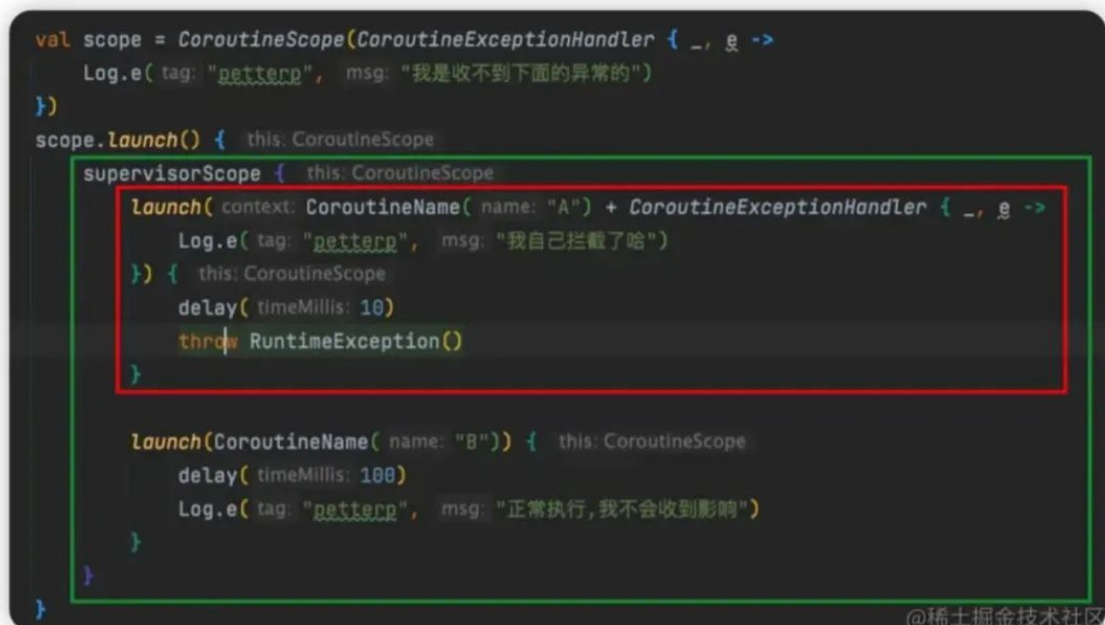
scope.launch() {
    supervisorScope {
        // launch A ❌
        launch(CoroutineName("A")) {
            delay(10)
            throw RuntimeException()
        }

        // launch B 👍
        launch(CoroutineName("B")) {
            delay(100)
            Log.e("petterp", "正常执行,我不会收到影响")
        }
    }
}

```

当 supervisorScope 里的所有子协程执行完成时，其就会正常退出作用域。

需要注意的是，supervisorScope 内部的 Job 为 SupervisorJob，所以当作用域中子协程异常时，异常不会主动层层向上传递，而是由子协程自行处理，所以意味着我们也可以为子协程增加 CoroutineExceptionHandler。如下所示：



```

val scope = CoroutineScope(CoroutineExceptionHandler { _, e ->
    Log.e( tag: "petterp", msg: "我是收不到下面的异常的")
})
scope.launch() { this: CoroutineScope
    supervisorScope { this: CoroutineScope
        launch( context: CoroutineName( name: "A") + CoroutineExceptionHandler { _, e ->
            Log.e( tag: "petterp", msg: "我自己拦截了啥")
        }) { this: CoroutineScope
            delay( timeMillis: 10)
            throw RuntimeException()
        }

        launch(CoroutineName( name: "B")) { this: CoroutineScope
            delay( timeMillis: 100)
            Log.e( tag: "petterp", msg: "正常执行,我不会收到影响")
        }
    }
}

```

@稀土掘金技术社区

当子协程异常时，因为我们使用了 supervisorScope，所以异常此时不会主动传递给外部，而是由子类自行处理。

当我们在内部 launch 子协程时，其实也就是类似 scope.launch，所以此时子协程A相也就是根协程，所以我们使用 CoroutineExceptionHandler 也可以正常拦截异常。但如果我们子协程不增加 CoroutineExceptionHandler，则此时异常会被supervisorScope 抛出，然后被外部的 CoroutineExceptionHandler 拦截(也就是初始化scope作用域时使用的ExceptionHandler)。

相应的，与 supervisorScope 相似的，还有一个 coroutineScope，下面我们也来说一下这个。

coroutineScope

其主要用于并行分解协程子任务时而使用，当其范围内任何子协程失败时，其所有的子协程也都被取消，一旦内部所有的子协程完成，其也会正常返回。

如下示例：

```
val scope = CoroutineScope(context: Job() + CoroutineExceptionHandler { _, _ ->
    Log.e(tag: "getterp", msg: "拦截异常")
})
scope.launch() { this: CoroutineScope
    coroutineScope { this: CoroutineScope
        launch(CoroutineName(name: "A")) { this: CoroutineScope
            delay(timeMillis: 10)
            throw NullPointerException()
        }
        launch(CoroutineName(name: "B")) { this: CoroutineScope
            delay(timeMillis: 1000)
            Log.e(tag: "getterp", msg: "子协程A 异常，导致我被取消了")
        }
    }
}
```

当子协程A异常未被捕获时，此时子协程B和整个协程作用域都将被异常取消，此时异常将传递到顶级 CoroutineExceptionHandler。

严格意义上来说，所有异常都可以用 tryCatch 去处理，只要我们的处理位置得当。但这并不是所有方式的最优解，特别是如果你想更优雅的处理异常时，此时就可以考虑 CoroutineExceptionHandler。下面我们通过实际需求来举例，从而体会异常处理的的一些实践。

什么时候该用 SupervisorJob，什么时候该用 Job?

引用官方的一句话就是想要避免取消操作在异常发生时被传播，记得使用 SupervisorJob，反之则使用 Job。

对于一个普通的协程，如何处理我的异常？

对于一个普通的协程，你可以在其协程作用域内使用 tryCatch(runCatching)，如果其是根协程，你也可以使用 CoroutineExceptionHandler 作为最后的拦截手段。如下所示：

```
val scope = CoroutineScope(Job())
scope.launch {
    runCatching { }
}

scope.launch(CoroutineExceptionHandler { _, throwable -> }) {
}
```

在某个子协程中，想使用 SupervisorJob 的特性去作为某个作用域去执行？

```
val scope = CoroutineScope(Job())
scope.launch(CoroutineExceptionHandler { _, _ -> }) {
    supervisorScope {
        launch(CoroutineName("A")) {
            throw NullPointerException()
        }
        launch(CoroutineName("B")) {
            delay(1000)
            Log.e("petterp", "依然会正常执行")
        }
    }
}
```

SupervisorJob+tryCatch

我们有两个接口 A，B 需要同时请求，当接口A异常时，需要不影响B接口的正常展示，当接口B异常时，此时界面展示异常信息。伪代码如下：

```
val scope = CoroutineScope(Job())
scope.launch {
    val jobA = async(SupervisorJob()) {
        throw NullPointerException()
    }
    val jobB = async(SupervisorJob()) {
        delay(100)
        1
    }
    val resultA = kotlin.runCatching { jobA.await() }
    val resultB = kotlin.runCatching { jobB.await() }
}
```

CoroutineExceptionHandler+SupervisorJob

如果你有一个顶级协程，并且需要自动捕获所有的异常，则此时可以选用上述方式，如下所示：

```
val exceptionHandler = CoroutineExceptionHandler { _, throwable ->
    Log.e("petterp", "自动捕获所有异常")
}
val ktxScope = CoroutineScope(SupervisorJob() + exceptionHandler)
```

推荐阅读：

[我的新书，《第一行代码 第3版》已出版！](#)

[模仿Android微信小程序，实现小程序独立任务视图的效果](#)

[Android 13运行时权限变更一览](#)

欢迎关注我的公众号

学习技术或投稿



长按上图，识别图中二维码即可关注

[阅读原文](#)

喜欢此内容的人还喜欢

从 0 开始学 Python 自动化测试开发（二）：环境搭建

霍格沃兹测试学院



Python 编程手册（下）

freeCodeCamp



node应用故障定位顶级技巧—动态追踪技术[Dynamic Trace]

元语言

