

17 | Context: 万物皆为Context?

2022-02-21 朱涛

《朱涛 · Kotlin编程第一课》

课程介绍 >



讲述：朱涛

时长 18:52 大小 17.29M



你好，我是朱涛。今天我们来学习 Kotlin 协程的 Context。

协程的 Context，在 Kotlin 当中有一个具体的名字，叫做 CoroutineContext。它是我们理解 Kotlin 协程非常关键的一环。

从概念上讲，CoroutineContext 很容易理解，它只是个 [上下文](#) 而已，实际开发中它最常见的用处就是切换线程池。不过，CoroutineContext 背后的代码设计其实比较复杂，如果不能深入理解它的设计思想，那我们在后面阅读协程源码，并进一步建立复杂并发结构的时候，都将会困难重重。

所以这节课，我将会从应用的角度出发，带你了解 CoroutineContext 的使用场景，并会对照源码带你理解它的设计思路。另外，知识点之间的串联也是很重要的，所以我还会带你分析它跟我们前面学的 Job、Deferred、launch、async 有什么联系，让你能真正理解和掌握协程的上下文，并建立一个**基于 CoroutineContext 的协程知识体系**。

Context 的应用

前面说过，`CoroutineContext` 就是协程的上下文。你在前面的第 14~16 讲里其实就已经见过它了。在 [第 14 讲](#) 我介绍 `launch` 源码的时候，`CoroutineContext` 其实就是函数的第一个参数：

 复制代码

```
1 // 代码段1
2
3 public fun CoroutineScope.launch(
4     //             这里
5     //             ↓
6     context: CoroutineContext = EmptyCoroutineContext,
7     start: CoroutineStart = CoroutineStart.DEFAULT,
8     block: suspend CoroutineScope.() -> Unit
9 ): Job {}
```

这里我先说一下，之前我们在调用 `launch` 的时候，都没有传 `context` 这个参数，因此它会使用默认值 `EmptyCoroutineContext`，顾名思义，这就是一个空的上下文对象。而如果我们想要指定 `launch` 工作的线程池的话，就需要自己传 `context` 这个参数了。

另外，在 [第 15 讲](#) 里，我们在挂起函数 `getUserInfo()` 当中，也用到了 `withContext()` 这个函数，当时我们传入的是“`Dispatchers.IO`”，这就是 Kotlin 官方提供的一个 `CoroutineContext` 对象。让我们来回顾一下：

 复制代码

```
1 // 代码段2
2
3 fun main() = runBlocking {
4     val user = getUserInfo()
5     logX(user)
6 }
7
8 suspend fun getUserInfo(): String {
9     logX("Before IO Context.")
10    withContext(Dispatchers.IO) {
11        logX("In IO Context.")
12        delay(1000L)
13    }
14    logX("After IO Context.")
15    return "BoyCoder"
16 }
17
```

```

18  /*
19  输出结果：
20  =====
21  Before IO Context.
22  Thread:main @coroutine#1
23  =====
24  =====
25  In IO Context.
26  Thread:DefaultDispatcher-worker-1 @coroutine#1
27  =====
28  =====
29  After IO Context.
30  Thread:main @coroutine#1
31  =====
32  =====
33  BoyCoder
34  Thread:main @coroutine#1
35  =====
36  */

```

可以看到，当我们在 `withContext()` 这里指定线程池以后，**Lambda** 当中的代码就会被分发到 **DefaultDispatcher** 线程池中去执行，而它外部的所有代码仍然还是运行在 **main** 之上。

其实，**Kotlin** 官方还提供了挂起函数版本的 `main()` 函数，所以我们的代码也可以改成这样：

 复制代码

```

1  // 代码段3
2
3  suspend fun main() {
4      val user = getUserInfo()
5      logX(user)
6  }

```

不过，你要注意的是：挂起函数版本的 `main()` 的底层做了很多封装，虽然它可以帮我们省去写 `runBlocking` 的麻烦，但不利于我们学习阶段的探索和研究。因此，后续的 **Demo** 我们仍然以 `runBlocking` 为主，你只需要知道 **Kotlin** 有这么一个东西，等到你深入理解协程以后，就可以直接用“`suspend main()`”写 **Demo** 了。

我们说回 `runBlocking` 这个函数，第 14 讲里我们介绍过，它的第一个参数也是 **CoroutineContext**，所以，我们也可以传入一个 **Dispatcher** 对象作为参数：

```

1  // 代码段4
2
3  //                                变化在这里
4  //                                ↓
5  fun main() = runBlocking(Dispatchers.IO) {
6      val user = getUserInfo()
7      logX(user)
8  }
9
10 /*
11  输出结果:
12  =====
13  Before IO Context.
14  Thread:DefaultDispatcher-worker-1 @coroutine#1
15  =====
16  =====
17  In IO Context.
18  Thread:DefaultDispatcher-worker-1 @coroutine#1
19  =====
20  =====
21  After IO Context.
22  Thread:DefaultDispatcher-worker-1 @coroutine#1
23  =====
24  =====
25  BoyCoder
26  Thread:DefaultDispatcher-worker-1 @coroutine#1
27  =====
28  */

```

这时候，我们会发现，所有的代码都运行在 `DefaultDispatcher` 这个线程池当中了。而 Kotlin 官方除了提供了 `Dispatchers.IO` 以外，还提供了 `Dispatchers.Main`、`Dispatchers.Unconfined`、`Dispatchers.Default` 这几种内置 `Dispatcher`。我来分别给你介绍一下：

- **Dispatchers.Main**，它只在 UI 编程平台才有意义，在 Android、Swing 之类的平台上，一般只有 Main 线程才能用于 UI 绘制。这个 `Dispatcher` 在普通的 JVM 工程当中，是无法直接使用的。
- **Dispatchers.Unconfined**，代表无所谓，当前协程可能运行在任意线程之上。
- **Dispatchers.Default**，它是用于 CPU 密集型任务的线程池。一般来说，它内部的线程个数是与机器 CPU 核心数量保持一致的，不过它有一个最小限制 2。
- **Dispatchers.IO**，它是用于 IO 密集型任务的线程池。它内部的线程数量一般会更多一些（比如 64 个），具体线程的数量我们可以通过参数来配置：

需要特别注意的是，Dispatchers.IO 底层是可能复用 Dispatchers.Default 当中的线程的。如果你足够细心的话，会发现前面我们用的都是 Dispatchers.IO，但实际运行的线程却是 DefaultDispatcher 这个线程池。

为了让这个问题更加清晰，我们可以把上面的例子再改一下：

[复制代码](#)

```

1 // 代码段5
2
3 //                                变化在这里
4 //                                ↓
5 fun main() = runBlocking(Default) {
6     val user = getUserInfo()
7     logX(user)
8 }
9
10 /*
11 输出结果：
12 =====
13 Before IO Context.
14 Thread:DefaultDispatcher-worker-1 @coroutine#1
15 =====
16 =====
17 In IO Context.
18 Thread:DefaultDispatcher-worker-2 @coroutine#1
19 =====
20 =====
21 After IO Context.
22 Thread:DefaultDispatcher-worker-2 @coroutine#1
23 =====
24 =====
25 BoyCoder
26 Thread:DefaultDispatcher-worker-2 @coroutine#1
27 =====
28 */

```

当 Dispatchers.Default 线程池当中有富余线程的时候，它是可以被 IO 线程池复用的。可以看到，后面三个结果的输出都是在同一个线程之上的，这就是因为 Dispatchers.Default 被 Dispatchers.IO 复用线程导致的。如果我们换成自定义的 Dispatcher，结果就会不一样了。

[复制代码](#)

```

1 // 代码段6
2
3
4 val mySingleDispatcher = Executors.newSingleThreadExecutor {
5     Thread(it, "MySingleThread").apply { isDaemon = true }
6 }.asCoroutineDispatcher()
7
8 //                      变化在这里
9 //                      ↓
10 fun main() = runBlocking(mySingleDispatcher) {
11     val user = getUserInfo()
12     logX(user)
13 }
14
15 public fun ExecutorService.asCoroutineDispatcher(): ExecutorCoroutineDispatcher
16     ExecutorCoroutineDispatcherImpl(this)
17
18 /*
19 输出结果:
20 =====
21 Before IO Context.
22 Thread:MySingleThread @coroutine#1
23 =====
24 =====
25 In IO Context.
26 Thread:DefaultDispatcher-worker-1 @coroutine#1
27 =====
28 =====
29 After IO Context.
30 Thread:MySingleThread @coroutine#1
31 =====
32 =====
33 BoyCoder
34 Thread:MySingleThread @coroutine#1
35 =====
36 */

```

在上面的代码中，我们是通过 `asCoroutineDispatcher()` 这个扩展函数，创建了一个 `Dispatcher`。从这里我们也能看到，`Dispatcher` 的本质仍然还是线程。这也再次验证了我们 [之前的说法](#)：协程运行在线程之上。

然后在这里，当我们为 `runBlocking` 传入自定义的 `mySingleDispatcher` 以后，程序运行的结果就不一样了，由于它底层并没有复用线程，因此只有“In IO Context”是运行在 `DefaultDispatcher` 这个线程池的，其他代码都运行在 `mySingleDispatcher` 之上。

另外，前面提到的 **`Dispatchers.Unconfined`**，我们也要额外注意。还记得之前学习 `launch` 的时候，我们遇到的例子吗？请问下面 4 行代码，它们的执行顺序是怎样的？


```

1 // 代码段7
2
3 fun main() = runBlocking {
4     logX("Before launch.") // 1
5     launch {
6         logX("In launch.") // 2
7         delay(1000L)
8         logX("End launch.") // 3
9     }
10    logX("After launch") // 4
11 }

```

如果你理解了第 14 讲的内容，那你一定能分析出它们的运行顺序应该是：1、4、2、3。

但你要注意，同样的代码模式在特殊的环境下，结果可能会不一样。比如在 Android 平台，或者是如果我们指定了 `Dispatchers.Unconfined` 这个特殊的 `Dispatcher`，它的这种行为模式也会被打破。比如像这样：

```

1 // 代码段8
2
3 fun main() = runBlocking {
4     logX("Before launch.") // 1
5     //          变化在这里
6     //          ↓
7     launch(Dispatchers.Unconfined) {
8         logX("In launch.") // 2
9         delay(1000L)
10        logX("End launch.") // 3
11    }
12    logX("After launch") // 4
13 }
14
15 /*
16 输出结果：
17 =====
18 Before launch.
19 Thread:main @coroutine#1
20 =====
21 =====
22 In launch.
23 Thread:main @coroutine#2
24 =====
25 =====
26 After launch

```

```
27 Thread:main @Coroutine#1
28 =====
29 =====
30 End launch.
31 Thread:kotlinx.coroutines.DefaultExecutor @coroutine#2
32 =====
33 */
```

以上代码的运行顺序就变成了：1、2、4、3。这一点，就再一次说明了 Kotlin 协程的难学。传了一个不同的参数进来，整个代码的执行顺序都变了，这谁不头疼呢？最要命的是，Dispatchers.Unconfined 设计的本意，也并不是用来改变代码执行顺序的。

请你留意“End launch”运行的线程“DefaultExecutor”，是不是觉得很乱？其实 Unconfined 代表的意思就是，**当前协程可能运行在任何线程之上，不作强制要求。**

由此可见，Dispatchers.Unconfined 其实是很危险的。所以，**我们不应该随意使用 Dispatchers.Unconfined。**

好，现在我们也了解了 CoroutineContext 的常见应用场景。不过，我们还没解释这节课的标题，什么是“万物皆为 Context”？

万物皆有 Context

所谓的“万物皆为 Context”，当然是一种夸张的说法，我们换成“万物皆有 Context”可能更加准确。

在 Kotlin 协程当中，但凡是重要的概念，都或多或少跟 CoroutineContext 有关系：Job、Dispatcher、CoroutineExceptionHandler、CoroutineScope，甚至挂起函数，它们都跟 CoroutineContext 有着密切的联系。甚至，它们之中的 Job、Dispatcher、CoroutineExceptionHandler 本身，就是 Context。

我这么一股脑地告诉你，你肯定觉得晕乎乎，所以下面我们就一个个来看。

CoroutineScope

在学习 launch 的时候，我提到过如果要调用 launch，就必须先有“协程作用域”，也就是 CoroutineScope。


```

1 // 代码段9
2
3 //          注意这里
4 //          ↓
5 public fun CoroutineScope.launch(
6     context: CoroutineContext = EmptyCoroutineContext,
7     start: CoroutineStart = CoroutineStart.DEFAULT,
8     block: suspend CoroutineScope.() -> Unit
9 ): Job {}
10
11 // CoroutineScope 源码
12 public interface CoroutineScope {
13     public val coroutineContext: CoroutineContext
14 }

```

如果你去看 `CoroutineScope` 的源码，你会发现，它其实就是一个简单的接口，而这个接口只有唯一的成员，就是 `CoroutineContext`。所以，`CoroutineScope` 只是对 `CoroutineContext` 做了一层封装而已，它的核心能力其实都来自于 `CoroutineContext`。

而 `CoroutineScope` 最大的作用，就是可以方便我们批量控制协程。

```

1 // 代码段10
2
3 fun main() = runBlocking {
4     // 仅用于测试，生成环境不要使用这么简易的CoroutineScope
5     val scope = CoroutineScope(Job())
6
7     scope.launch {
8         logX("First start!")
9         delay(1000L)
10        logX("First end!") // 不会执行
11    }
12
13    scope.launch {
14        logX("Second start!")
15        delay(1000L)
16        logX("Second end!") // 不会执行
17    }
18
19    scope.launch {
20        logX("Third start!")
21        delay(1000L)
22        logX("Third end!") // 不会执行
23    }
24

```

```

25     delay(500L)
26
27     scope.cancel()
28
29     delay(1000L)
30 }
31
32 /*
33 输出结果:
34 =====
35 First start!
36 Thread:DefaultDispatcher-worker-1 @coroutine#2
37 =====
38 =====
39 Third start!
40 Thread:DefaultDispatcher-worker-3 @coroutine#4
41 =====
42 =====
43 Second start!
44 Thread:DefaultDispatcher-worker-2 @coroutine#3
45 =====
46 */

```

在上面的代码中，我们自己创建了一个简单的 `CoroutineScope`，接着，我们使用这个 `scope` 连续创建了三个协程，在 500 毫秒以后，我们就调用了 `scope.cancel()`，这样一来，代码中每个协程的“end”日志就不会输出了。

这同样体现了协程**结构化并发**的理念，相同的功能，我们借助 `Job` 也同样可以实现。关于 `CoroutineScope` 更多的底层细节，我们会在源码篇的时候深入学习。

那么接下来，我们就看看 `Job` 跟 `CoroutineContext` 的关系。

Job 和 Dispatcher

如果说 `CoroutineScope` 是封装了 `CoroutineContext`，那么 `Job` 就是一个真正的 `CoroutineContext` 了。

 复制代码

```

1 // 代码段11
2
3 public interface Job : CoroutineContext.Element {}
4
5 public interface CoroutineContext {
6     public interface Element : CoroutineContext {}

```

```
7 }
```

上面这段代码很有意思，`Job` 继承自 `CoroutineContext.Element`，而 `CoroutineContext.Element` 仍然继承自 `CoroutineContext`，这就意味着 `Job` 是间接继承自 `CoroutineContext` 的。所以说，`Job` 确实是一个真正的 `CoroutineContext`。

所以，我们写这样的代码也完全没问题：

[📄 复制代码](#)

```
1 // 代码段12
2
3 fun main() = runBlocking {
4     val job: CoroutineContext = Job()
5 }
```

不过，更有趣的是 `CoroutineContext` 本身的接口设计。

[📄 复制代码](#)

```
1 // 代码段13
2
3 public interface CoroutineContext {
4
5     public operator fun <E : Element> get(key: Key<E>): E?
6
7     public operator fun plus(context: CoroutineContext): CoroutineContext {}
8
9     public fun minusKey(key: Key<*>): CoroutineContext
10
11     public fun <R> fold(initial: R, operation: (R, Element) -> R): R
12
13     public interface Key<E : Element>
14 }
```

从上面代码中的 `get()`、`plus()`、`minusKey()`、`fold()` 这几个方法，我们可以看到 `CoroutineContext` 的接口设计，就跟集合 API 一样。准确来说，它的 API 设计和 `Map` 十分类似。

Map 接口

```
public interface Map<K,V> {  
    V get(Object key);  
  
    V put(K key, V value);  
  
    V remove(Object key);  
  
    void forEach(){...}  
}
```

CoroutineContext 接口

```
interface CoroutineContext {  
    operator fun <E : Element> get(key: Key<E>): E?  
  
    operator fun plus(c: CoroutineContext) {...}  
  
    fun minusKey(k: Key<*>): CoroutineContext  
  
    fun <R> fold(i: R, o: (R, Element) -> R): R  
}
```

所以，我们完全可以**把 CoroutineContext 当作 Map 来用**。

 复制代码

```
1 // 代码段14  
2  
3 @OptIn(ExperimentalStdlibApi::class)  
4 fun main() = runBlocking {  
5     // 注意这里  
6     val scope = CoroutineScope(Job() + mySingleDispatcher)  
7  
8     scope.launch {  
9         // 注意这里  
10         logX(coroutineContext[CoroutineDispatcher] == mySingleDispatcher)  
11         delay(1000L)  
12         logX("First end!") // 不会执行  
13     }  
14  
15     delay(500L)  
16     scope.cancel()  
17     delay(1000L)  
18 }  
19 /*  
20 输出结果:  
21 =====  
22 true  
23 Thread:MySingleThread @coroutine#2  
24 =====  
25 */
```

在上面的代码中，我们使用了“Job() + mySingleDispatcher”这样的方式创建 CoroutineScope，代码之所以这么写，是因为 CoroutineContext 的 plus() 进行了**操作符重**

载。

 复制代码

```
1 // 代码段15
2
3 //      操作符重载
4 //      ↓
5 public operator fun <E : Element> plus(key: Key<E>): E?
```

你注意这里代码中的 **operator** 关键字，如果少了它，我们就得换一种方式了：
`mySingleDispatcher.plus(Job())`。因为，当我们用 **operator** 修饰 `plus()` 方法以后，就可以用
“+”来重载这个方法，类似的，`List` 和 `Map` 都支持这样的写法：`list3 = list1+list2`、`map3 =
map1 + map2`，这代表集合之间的合并。

另外，我们还使用了“`coroutineContext[CoroutineDispatcher]`”这样的方式，访问当前协程所对
应的 `Dispatcher`。这也是因为 `CoroutineContext` 的 `get()`，支持了**操作符重载**。

 复制代码

```
1 // 代码段16
2
3 //      操作符重载
4 //      ↓
5 public operator fun <E : Element> get(key: Key<E>): E?
```

实际上，在 `Kotlin` 当中很多集合也是支持 `get()` 方法重载的，比如 `List`、`Map`，我们都可以使
用这样的语法：`list[0]`、`map[key]`，以数组下标的方式来访问集合元素。

还记得我们在 [第 1 讲](#)提到的“集合与数组的访问方式一致”这个知识点吗？现在我们知道了，
这都要归功于操作符重载。实际上，`Kotlin` 官方的源代码当中大量使用了操作符重载来简化代
码逻辑，而 `CoroutineContext` 就是一个最典型的例子。

如果你足够细心的话，这时候你应该也发现了：`Dispatcher` 本身也是 `CoroutineContext`，不然
它怎么可以实现“`Job() + mySingleDispatcher`”这样的写法呢？最重要的是，当我们以这样的方
式创建出 `scope` 以后，后续创建的协程就全部都运行在 `mySingleDispatcher` 这个线程之上
了。

那么，**Dispatcher** 到底是如何跟 **CoroutineContext** 建立关系的呢？让我们来看看它的源码吧。

 复制代码

```
1 // 代码段17
2
3 public actual object Dispatchers {
4
5     public actual val Default: CoroutineDispatcher = DefaultScheduler
6
7     public actual val Main: MainCoroutineDispatcher get() = MainDispatcherLoade
8
9     public actual val Unconfined: CoroutineDispatcher = kotlinx.coroutines.Uncco
10
11     public val IO: CoroutineDispatcher = DefaultIoScheduler
12
13     public fun shutdown() {    }
14 }
15
16 public abstract class CoroutineDispatcher :
17     AbstractCoroutineContextElement(ContinuationInterceptor), ContinuationInter
18
19 public interface ContinuationInterceptor : CoroutineContext.Element {}
```

可以看到，Dispatchers 其实是一个 object 单例，它的内部成员的类型是 CoroutineDispatcher，而它又是继承自 ContinuationInterceptor，这个类则是实现了 CoroutineContext.Element 接口。由此可见，Dispatcher 确实就是 CoroutineContext。

其他 CoroutineContext

除了上面几个重要的 CoroutineContext 之外，协程其实还有一些上下文是我们还没提到的。比如 **CoroutineName**，当我们创建协程的时候，可以传入指定的名称。比如：

 复制代码

```
1 // 代码段18
2
3 @OptIn(ExperimentalStdlibApi::class)
4 fun main() = runBlocking {
5     val scope = CoroutineScope(Job() + mySingleDispatcher)
6     // 注意这里
7     scope.launch(CoroutineName("MyFirstCoroutine!")) {
8         logX(coroutineContext[CoroutineDispatcher] == mySingleDispatcher)
9         delay(1000L)
10        logX("First end!")
11    }
```

```

11     }
12
13     delay(500L)
14     scope.cancel()
15     delay(1000L)
16 }
17
18 /*
19 输出结果:
20
21 =====
22 true
23 Thread:MySingleThread @MyFirstCoroutine!#2    // 注意这里
24 =====
25 */

```

在上面的代码中，我们调用 `launch` 的时候，传入了“`CoroutineName("MyFirstCoroutine!")`”作为协程的名字。在后面输出的结果中，我们得到了“`@MyFirstCoroutine!#2`”这样的输出。由此可见，其中的数字“2”，其实是一个自增的唯一 ID。

`CoroutineContext` 当中，还有一个重要成员是 **`CoroutineExceptionHandler`**，它主要负责处理协程当中的异常。

 复制代码

```

1 // 代码段19
2
3 public interface CoroutineExceptionHandler : CoroutineContext.Element {
4
5     public companion object Key : CoroutineContext.Key<CoroutineExceptionHandle
6
7     public fun handleException(context: CoroutineContext, exception: Throwable)
8 }

```

可以看到，`CoroutineExceptionHandler` 的接口定义其实很简单，我们基本上一眼就能看懂。`CoroutineExceptionHandler` 真正重要的，其实只有 `handleException()` 这个方法，如果我们要自定义异常处理器，我们就只需要实现该方法即可。

 复制代码

```

1 // 代码段20
2
3 // 这里使用了挂起函数版本的main()
4 suspend fun main() {
5     val myExceptionHandler = CoroutineExceptionHandler { _, throwable ->

```



```

6         println("Catch exception: $throwable")
7     }
8     val scope = CoroutineScope(Job() + mySingleDispatcher)
9
10    val job = scope.launch(myExceptionHandler) {
11        val s: String? = null
12        s!!.length // 空指针异常
13    }
14
15    job.join()
16 }
17 /*
18 输出结果:
19 Catch exception: java.lang.NullPointerException
20 */

```

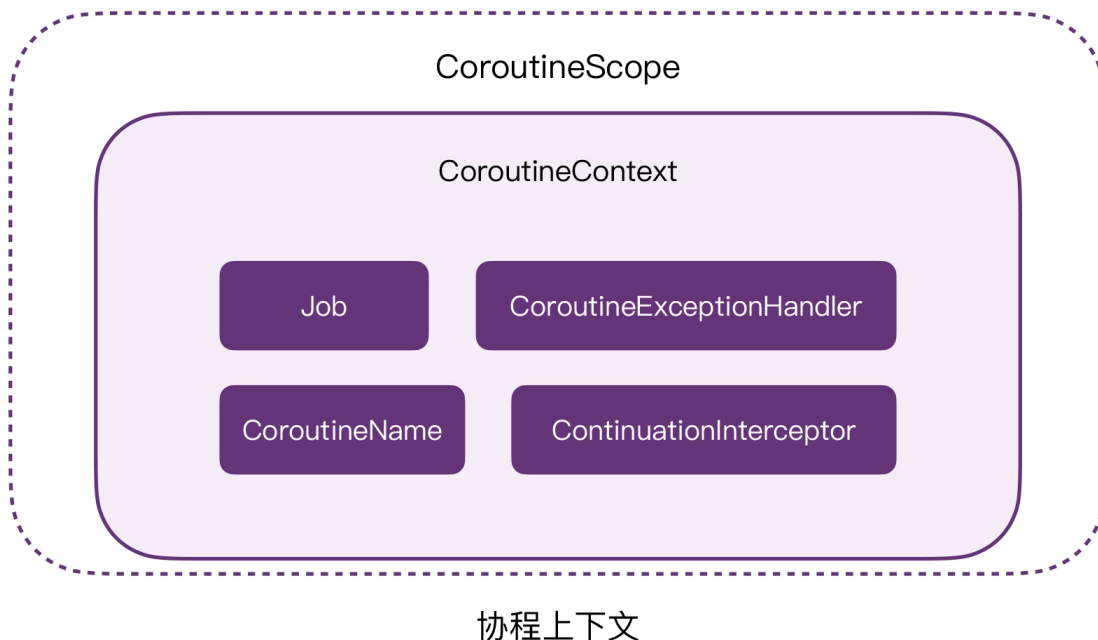
不过，虽然 `CoroutineExceptionHandler` 的用法看起来很简单，但当它跟协程“结构化并发”理念相结合以后，内部的异常处理逻辑是很复杂的。关于协程异常处理的机制，我们会在第 23 讲详细介绍。

小结

这节课的内容到这里就结束了，我们来总结一下吧。

- `CoroutineContext`，是 Kotlin 协程当中非常关键的一个概念。它本身是一个接口，但它的接口设计与 `Map` 的 API 极为相似，我们在使用的过程中，也可以把它当作 **Map 来用**。
- 协程里很多重要的类，它们本身都是 `CoroutineContext`。比如 `Job`、`Deferred`、`Dispatcher`、`ContinuationInterceptor`、`CoroutineName`、`CoroutineExceptionHandler`，它们都继承自 `CoroutineContext` 这个接口。也正因为它们都继承了 `CoroutineContext` 接口，所以我们可以通过**操作符重载**的方式，写出更加灵活的代码，比如“`Job() + mySingleDispatcher+CoroutineName("MyFirstCoroutine!")`”。
- 协程当中的 `CoroutineScope`，本质上也是 `CoroutineContext` 的一层**简单封装**。
- 另外，协程里极其重要的“挂起函数”，它与 `CoroutineContext` 之间也有着非常紧密的联系。

另外我也画了一张结构图，来描述 `CoroutineContext` 元素之间的关系，方便你建立完整的知识体系。



所以总的来说，我们前面学习的 Job、Dispatcher、CoroutineName，它们本质上只是 CoroutineContext 这个集合当中的一种数据类型，只是恰好 Kotlin 官方让它们都继承了 CoroutineContext 这个接口。而 CoroutineScope 则是对 CoroutineContext 的进一步封装，它的核心能力，全部都是源自于 CoroutineContext。

思考题

课程里，我提到了“挂起函数”与 CoroutineContext 也有着紧密的联系，请问，你能找到具体的证据吗？或者，你觉得下面的代码能成功运行吗？为什么？


复制代码


```
1 // 代码段21
2
3 import kotlinx.coroutines.*
4 import kotlin.coroutines.coroutineContext
5
6 //          挂起函数能可以访问协程上下文吗？
7 //          ↓
8 suspend fun testContext() = coroutineContext
9
10 fun main() = runBlocking {
11     println(testContext())
12 }
```

欢迎在留言区分享你的答案，也欢迎你把今天的内容分享给更多的朋友。

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 16 | Job：协程也有生命周期吗？

下一篇 18 | 实战：让KtHttp支持挂起函数

精选留言 (11)

 写留言



夜班同志

2022-02-22

挂起函数的Continuation就有CoroutineContext

作者回复: 没错~



 2



白泽、

2022-03-12

如果为协程作用域创建时传入多个CoroutineContext，比如 Job() + Dispatcher.IO + Dispatcher.Main，那么携程最终会在哪个线程池中执行呢

作者回复: Dispatcher之间的组合其实并没有意义，你可以将其理解为后者替换前者。不过，在大部分情况下，IDE都会直接报错并告诉你：“Dispatcher之间的组合没有意义”。

...

```
fun main() {  
    // 报错
```

```
val scope = CoroutineScope(Dispatchers.IO + Dispatchers.Main)

}
```



1



神秘嘉Bin

2022-03-02

suspend方法需要在协程中执行，协程又一定有上下文，所以可以访问的到哈~ 也就是在suspend方法中可以访问当前协程上下文，并且拿到一些有用的信息

作者回复: 很到位~



1



白乾涛

2022-02-26

- 1、思考题中的方法为什么要加 suspend，加不加有什么区别吗？
- 2、为什么代码打印的都是 EmptyCoroutineContext，且没有 name？

```
import kotlinx.coroutines.*
```

```
import kotlinx.coroutines.GlobalScope.coroutineContext
```

```
fun main() = runBlocking {
    println(1) // 1 - EmptyCoroutineContext - null
    CoroutineScope(Dispatchers.IO + Job() + CoroutineName("bqt")).launch {
        println(2) // 2 - EmptyCoroutineContext - null
    }
    delay(100L)
}
```

```
suspend fun println(text: Any) = println("$text - $coroutineContext - ${coroutineContext[CoroutineName]?.name}")
```

作者回复: import kotlin.coroutines.coroutineContext

导包的时候，不要弄错了。

共 2 条评论 >



1



WWWarmFly

2022-03-20

请教老师，

Dispatcher 内部成员的类型是CoroutineContext，这里怎么推出

Dispatcher 确实就是 CoroutineContext

作者回复: 这样的关系: Dispatcher - CoroutineContext.Element - CoroutineContext



Shanks-王冲

2022-03-17

Kotlin1.6源码package kotlin.coroutines中找到了这个，public suspend inline val coroutineContext: CoroutineContext，成员定义成suspend了，我不知道该怎么解释，贴出试试

作者回复: 没错，方向是对的，这就是我希望你们去看的coroutineContext变量，你可以再想想它的作用吗？

PS: 具体答案我会在第27讲里给出的。



Renext

2022-03-08

代码段6报错: Cannot access 'ExecutorCoroutineDispatcherImpl': it is private in file

作者回复: ExecutorService.asCoroutineDispatcher()是Kotlin的源码哈，不需要你写进工程里的。



神秘嘉Bin

2022-03-02

如果你理解了第 14 讲的内容，那你一定能分析出它们的运行顺序应该是：1、4、2、3。

也有可能是1、2、4、3吧？ 这个得看CPU的调度了，也有可能子协程的2线运行吧？

作者回复: ``

```
fun main() = runBlocking {  
    logX("Before launch.") // 1  
    launch {
```

```
logX("In launch.") // 2
delay(1000L)
logX("End launch.") // 3
}
logX("After launch") // 4
}
...
```

如果只针对这个案例，由于这里不涉及到多线程，所有协程都会运行在main之上，所以，我们基本上可以认为代码的执行顺序是这样的：1、4、2、3。

但如果涉及到多线程，则可能由于主线程繁忙，coroutine2在子线程先运行，而输出：1、2、4、3。

另外，如果我们配置其他的启动模式，或者其他的Context，则可能出现其他的代码运行顺序。

共 2 条评论 >



7Promise

2022-02-21

思考题代码可以运行。coroutineContext方法是返回当前的CoroutineContext，因为runBlocking是CorouScope，CorouScope具有成员CoroutineContext，所以coroutineContext方法可以返回runBlocking的CoroutineContext。

作者回复: 嗯，没错。



Allen

2022-02-21

代码是可以运行的，coroutineContext 的作用是获取当前运行作用域所对应协程的上下文信息。

这里打印出来的信息就是 runBlocking 所运行的协程所对应上下文的信息。

```
[CoroutineId(1), "coroutine#1":BlockingCoroutine{Active}@759ebb3d, BlockingEventLoop@484b61fc]
```

作者回复: 是的。

共 4 条评论 >





面无表情的生鱼片

2022-02-21

请教老师，经常看到 `Job() + Dispatcher`，这么做是什么原因呢

作者回复: 操作符重载你肯定能理解了对吧？至于 `Job() + Dispatcher` 的含义，其实就是同时指定：`parentJob`，还有线程池。

共 2 条评论 >

