# L5-协程

本质上，协程是轻量级的线程。

在协程部分，需要我们整理常用协程函数速查表，和常用协程操作符速查表。

漫画理解协程：

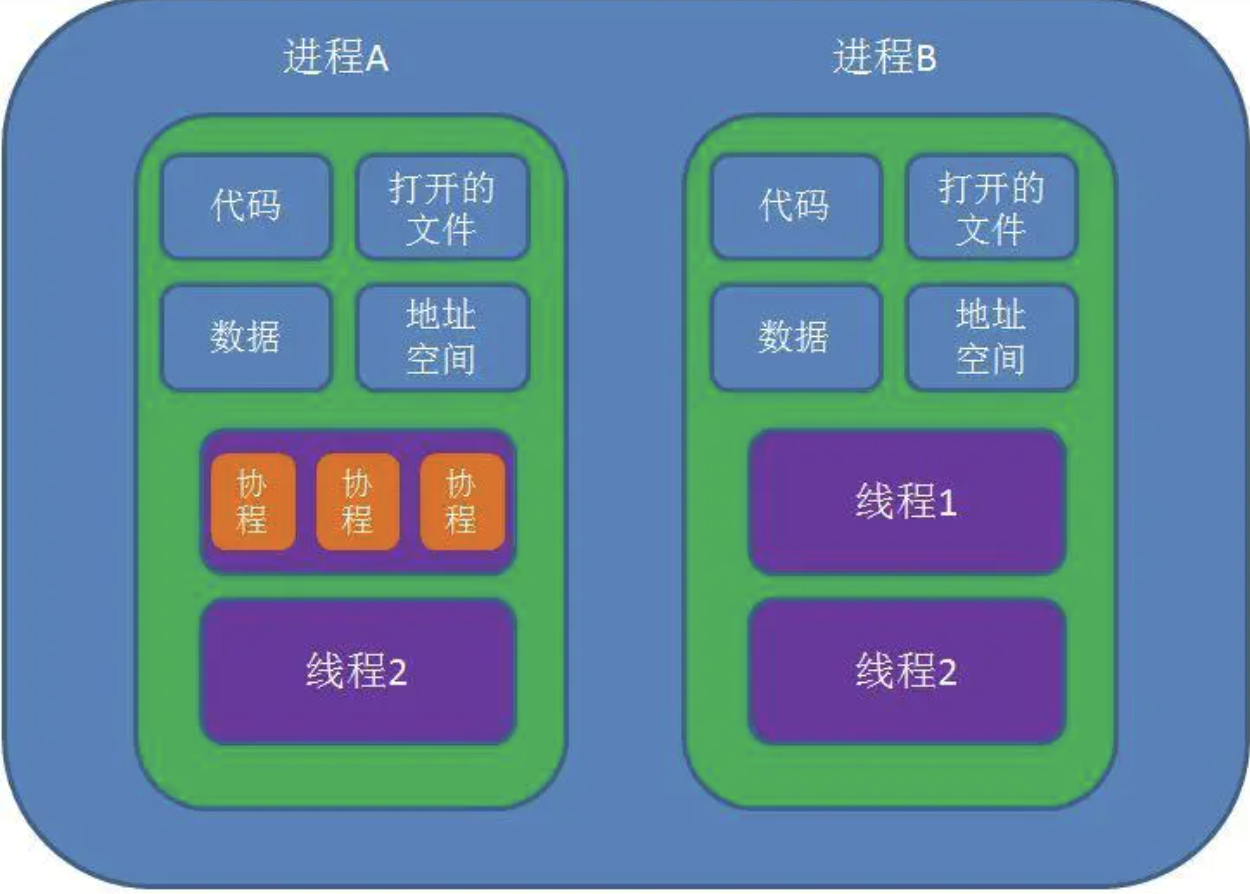
https://www.sohu.com/a/236536167\_684445

先对协程进行概念性的理解。直接先说区别，Coroutine是编译器级的，Process和Thread是操作系统级的。

线程之间的切换需要深入到内核级别，因此线程的切换代价比较大，表现在：  
\* 线程对象的创建和初始化  
\* 线程上下文切换  
\* 线程状态的切换由系统内核完成  
\* 对变量的操作需要加锁

因为在同一个线程里，协程之间的切换不涉及线程上下文的切换和线程状态的改变，不存在资源、数据并发，所以不用加锁，只需要判断状态就OK，所以执行效率比多线程高很多

· 协程是非阻塞式的(也有阻塞API)，一个协程在进入阻塞后不会阻塞当前线程，当前线程会去执行其他协程任务



理性看待协程，虽然从协程的定义来看非常适合Android开发，因为协程是在线程当中的，又不阻塞线程，那么我们就可以在主线程中启动协程，当协程返回结果时直接刷新ui。

如果这样实现远比我们使用第三方框架然后切换线程要方便的多。

如果将程序分为IO密集型应用和CPU密集型应用，二者的发展历程大致如下：

IO密集型应用: 多进程->多线程->事件驱动->协程  
CPU密集型应用:多进程->多线程

协程主要是让原来要使用“异步+回调方式”写出来的复杂代码, 简化成可以用看似同步的方式写出来（对线程的操作进一步抽象）。这样我们就可以按串行的思维模型去组织原本分散在不同上下文中的代码逻辑，而不需要去处理复杂的状态同步问题。

#### 3.6.0 准备工作

### 搭建协程代码工程

首先，我们来新建一个Kotlin Gradle工程。生成标准gradle工程后，在配置文件build.gradle中，配置kotlinx-coroutines-core依赖：

添加 dependencies :

1. compile 'org.jetbrains.kotlinx:kotlinx-coroutines-core:0.16'

kotlinx-coroutines还提供了下面的模块：

compile group: 'org.jetbrains.kotlinx', name: 'kotlinx-coroutines-jdk8', version: '0.16'

compile group: 'org.jetbrains.kotlinx', name: 'kotlinx-coroutines-nio', version: '0.16'

compile group: 'org.jetbrains.kotlinx', name: 'kotlinx-coroutines-reactive', version: '0.16'

我们使用Kotlin最新的1.1.3-2 版本:

1. buildscript {
2. ext.kotlin\_version = '1.1.3-2'
3. ...
4. dependencies {
5. classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin\_version"
6. }
7. }

其中，kotlin-gradle-plugin是Kotlin集成Gradle的插件。

另外，配置一下JCenter 的仓库:

1. repositories {
2. jcenter()
3. }

### 9.1.2 简单协程示例

下面我们先来看一个简单的协程示例。

运行下面的代码：

1. fun firstCoroutineDemo0() {
2. launch(CommonPool) {
3. delay(3000L, TimeUnit.MILLISECONDS)
4. println("Hello,")
5. }
6. println("World!")
7. Thread.sleep(5000L)
8. }

#### 3.6.1桥接阻塞与非阻塞的世界

Kotlin中提供了runBlocking函数来实现类似主协程的功能：

runBlocking函数来实现类似主协程的功能。 runBlocking函数不是用来当作普通协程函数使用的，它的设计主要是用来桥接普通阻塞代码和挂起风格的（suspending style）的非阻塞代码的

[runBlocking](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 与 [coroutineScope](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 可能看起来很类似，因为它们都会等待其协程体以及所有子协程结束。 主要区别在于，[runBlocking](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 方法会阻塞当前线程来等待， 而 [coroutineScope](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 只是挂起，会释放底层线程用于其他用途。 由于存在这点差异，[runBlocking](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 是常规函数，而 [coroutineScope](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 是挂起函数。

#### 3.6.2 取消与超时

在Java中有两类线程：用户线程 (User Thread)、守护线程 (Daemon Thread)。

所谓守护线程，是指在程序运行的时候在后台提供一种通用服务的线程，比如垃圾回收线程就是一个很称职的守护者，并且这种线程并不属于程序中不可或缺的部分。因此，当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程。

我们可以看出，活动的协程不会使进程保持活动状态。它们的行为就像守护程序线程。

Join()方法的介绍

kotlin 协程的所有suspend 函数都是可以取消的。我们可以通过job的isActive状态来判断协程的状态，或者检查是否有抛出 CancellationException 时取消。

协程正工作在循环计算中，并且不检查协程当前的状态, 那么调用cancel来取消协程将无法停止协程的运行,我们可以看出，即使我们调用了cancel函数，当前的job状态isAlive是false了，但是协程的代码依然一直在运行，并没有停止。

### 计算代码协程的有效取消

#### 方法一： 显式检查取消状态isActive

此部分可以先介绍 try finally 语法 和 isActive 字段。此部分可以从后往前说，否则会产生疑惑。结合文件操作的案例解释较好。

注意展示cancel 与 cancelAndJoin的区别。

#### 3.6.3 组合挂起函数

此部分可以先介绍measureTimeMillis 方法，避免大家产生理解上的障碍。

在这部分，首先要介绍 suspend 方法和普通方法的区别。

之后介绍同步执行和异步执行（async）。最后介绍一下取消就可以了。

代码块中的delay(3000L, TimeUnit.MILLISECONDS)函数，是一个用suspend关键字修饰的函数，我们称之为挂起函数。挂起函数只能从协程代码内部调用，普通的非协程的代码不能调用。

此处注意介绍 suspend,async（await）;在完成两个方法的介绍后，介绍launch与async的区别。以及返回对象job和De

#### 3.6.4 协程上下文与调度器

此部分较为重要，需要介绍清楚上下文的概念。

协程上下文包含一个 协程调度器 （参见 [CoroutineDispatcher](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-dispatcher/index.html)）它确定了相关的协程在哪个线程或哪些线程上执行。

协程调度器主要是控制我们的协程在哪个线程中运行，那么当我们的协程在非ui线程运行的时候，回调ui线程依然要做线程通信。

带有协程上下文参数的方法，均可以指定协程上下文。

关于非受限调度器和受限调度器这里面要展示Demo体现出，非受限调度器，耗时任务执行前后都在统一线程中，而受限的默认调度器耗时任务执行前后不在统一线程中。

协程的调试正常使用示例展示即可。

在不同线程间跳转，讲完之后要提醒大家：注意，在这个例子中，当我们不再需要某个在 [newSingleThreadContext](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/new-single-thread-context.html) 中创建的线程的时候， 它使用了 Kotlin 标准库中的 use 函数来释放该线程。

#### 3.6.5 子协程父协程与作用域

子协程

当一个协程被其它协程在 [CoroutineScope](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html) 中启动的时候， 它将通过 [CoroutineScope.coroutineContext](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/coroutine-context.html) 来承袭上下文，并且这个新协程的 [Job](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html) 将会成为父协程作业的 子 作业。**当一个父协程被取消的时候，所有它的子协程也会被递归的取消**。**然而，当使用 [GlobalScope](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-global-scope/index.html) 来启动一个协程时，则新协程的作业没有父作业。 因此它与这个启动的作用域无关且独立运作。**

**父协程**

一个父协程总是等待所有的子协程执行结束。父协程并不显式的跟踪所有子协程的启动，并且不必使用 [Job.join](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/join.html) 在最后的时候等待它们。

协程命名-此处可以展示协程的常用方法。也可介绍下运算符重载的概念。

### 3.7 异步流

挂起函数可以异步的返回单个值，但是该如何异步返回多个计算好的值呢？这正是 Kotlin 流（Flow）的用武之地。

注意使用 [Flow](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow/index.html) 的代码与先前示例的下述区别：

* 名为 [flow](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow/index.html) 的 [Flow](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flow.html) 类型构建器函数。
* flow { ... } 构建块中的代码可以挂起。
* 函数 simple 不再标有 suspend 修饰符。
* **流使用 [emit](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow-collector/emit.html) 函数 发射 值。**
* **流使用 [collect](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/collect.html) 函数 收集 值。**

流式操作，式kotlin中较为优雅的处理复杂遍历的方式。

如我们对一组数据进行遍历，可以看成数据是一个一个像流一样被处理，那么一般我们要做的就是看看是否符合要求（判断逻辑）使用map；对符合要求的数据再进行处理.collect逻辑。

flowOn 操作符 结束后，涉及到生产者和消费者速度协调处理的问题，kotlin封装了我们常见的业务逻辑，其实都是有迹可循的，我们可以结合实际用例，方便大家对这些操作服的理解。

缓冲(可以理解为下载功能)

合并conflate(可以理解为视频流抛弃)

collectLatest 输入都会进入到处理函数，但是新的输入会导致旧的未完成的输入结束云南行，并开始新的输入的运行。停止处理当前的任务 立即开始新任务的处理

合并 可根据 代码实现后的命令行输出 比较zip和combine的区别。

简单来说zip是等两个结果都符合的时候再继续进行合并处理。

而combine类似与缓存了结果，当有新结果时和当前的老结果进行合并处理。

展平流 一个流程的结束是下一个流程的开始。

一个流的结束可以看成是一个流返回返回结果了，另外一个流接收到此返回结果进行他的流程处理。当然他也可能是一个流。这种反而更贴近现实的业务模型。

### 3.8通道-生产者消费者-管道

#### **3.8.1通道**

延迟对象提供了一种在协程之间传输单个值的方法。而通道（Channel）提供了一种传输数据流的方法。通道是使用 SendChannel 和使用 ReceiveChannel 之间的非阻塞通信。

**通道跟阻塞队列一个关键的区别是：通道有挂起的操作, 而不是阻塞的, 同时它可以关闭。**

**这里可以先说明通道的模型是生产者与消费者模型。**

**通道是较为容易理解的。可以一句话概括为，缓存为一的非阻塞生产者消费者模型。**

关闭-和队列不同，**一个通道可以通过被关闭来表明没有更多的元素将会进入通道**。 **在接收者中可以定期的使用 for 循环来从通道中接收元素**。

#### f3.8.2构建通道生产者

使用协程生成元素序列的模式非常常见。这是在并发代码中经常有的生产者-消费者模式。

#### 3.8.3 管道

管道（Pipeline）是一种模式, **我们可以用一个协程生产无限序列**。

可以理解为，通道是封装好入口和出口的一个异步队列。

而管道是我们可以串联起来进行数据流转的一个工具。

此部分目前示例代码资料较少，介绍即可。

这里面有一个核心逻辑即管道串联后，最后一个管道调用receive的时候整条管道会从管道入口开始接收一遍。

这也就是给出的例子中，参与到循环的数字，逐步递增的原因。

#### 3.8.4 扇出与扇入

扇出与扇入的意思是一个管道可以介入多个输出，一个管道同时可以作为多个输入。

### 3.9协程方法速查表

|  |  |  |
| --- | --- | --- |
| 协程方法名 | 是否阻塞 | 描述 |
| launch | 否 | launch函数它以非阻塞（non-blocking）当前线程的方式，启动一个新的协程后台任务，并返回一个Job类型的对象作为当前协程的引用。 |
| runBlocking | 否 | runBlocking函数来实现类似主协程的功能。 runBlocking函数不是用来当作普通协程函数使用的，它的设计主要是用来桥接普通阻塞代码和挂起风格的（suspending style）的非阻塞代码的 |
| withTimeout | 否 | 设置超时的函数，方法体里面实现耗时操作。 |
| Suspend 修饰方法 | 是 | 这个执行可以看成线性执行。 |
| measureTimeMillis | 是 | 和内容有关，包裹代码块。计算方法体内整体的运行时间。需要方法体内运行结束。 |
| async | 否 | 方法体，将同步方法变成异步执行.  通过返回对象的await() 等待执行完毕 拿到运行结果 |

|  |  |
| --- | --- |
| 参数 | 说明 |
| context | 协程上下文 |
| Start | 协程启动选项 |
| Block | 协程真正要执行的代码块，必须是suspend修饰的刮起函数 |
| Job(返回类型) | 这个launch函数返回一个Job类型，Job是协程创建的后台任务的概念，它持有该协程的引用。Job接口实际上继承自CoroutineContext类型。一个Job有如下三种状态  截屏2021-01-23 下午4.37.14 |
| CommonPool：共享线程池 | CommonPool是代表共享线程池，它的主要作用是用来调度计算密集型任务的协程的执行。这个CommonPool是代表共享线程池，它的主要作用是用来调度计算密集型任务的协程的执行。它的实现使用的是java.util.concurrent包下面的API。它首先尝试创建一个java.util.concurrent.ForkJoinPool（ForkJoinPool是一个可以执行ForkJoinTask的ExcuteService，它采用了work-stealing模式：所有在池中的线程尝试去执行其他线程创建的子任务，这样很少有线程处于空闲状态，更加高效）；如果不可用，就使用java.util.concurrent.Executors来创建一个普通的线程池：Executors.newFixedThreadPool。 |
| Deferred | public interface Deferred<out T> : Job  截屏2021-01-27 下午2.57.11  截屏2021-01-27 下午2.57.23 |

## 3.10 协程与线程比较

直接先说区别，**协程是编译器级的，而线程是操作系统级的**。

协程通常是由编译器来实现的机制。线程看起来也在语言层次，但是内在原理却是操作系统先有这个东西，然后通过一定的API暴露给用户使用，两者在这里有不同。

协程就是用户空间下的线程。用协程来做的东西，用线程或进程通常也是一样可以做的，但往往多了许多加锁和通信的操作。

**线程是抢占式，而协程是非抢占式的，所以需要用户自己释放使用权来切换到其他协程，因此同一时间其实只有一个协程拥有运行权，相当于单线程的能力。**

协程并不是取代线程, 而是抽象于线程之上, 线程是被分割的CPU资源, 协程是组织好的代码流程, 协程需要线程来承载运行, 线程是协程的资源, 但协程不会直接使用线程, 协程直接利用的是执行器(Interceptor), 执行器可以关联任意线程或线程池, 可以使当前线程, UI线程, 或新建新程.。

线程是协程的资源。协程通过Interceptor来间接使用线程这个资源。

## 9.14 协程的好处

与多线程、多进程等并发模型不同，**协程依靠user-space调度，而线程、进程则是依靠kernel来进行调度**。**线程、进程间切换都需要从用户态进入内核态，而协程的切换完全是在用户态完成，且不像线程进行抢占式调度，协程是非抢占式的调度。**

通常多个运行在同一调度器中的协程运行在一个线程内，这也消除掉了多线程同步等带来的编程复杂性。同一时刻同一调度器中的协程只有一个会处于运行状态。

我们使用协程，程序只在用户空间内切换上下文，不再陷入内核来做线程切换，这样可以避免大量的用户空间和内核空间之间的数据拷贝，降低了CPU的消耗，从而大大减缓高并发场景时CPU瓶颈的窘境。

另外，使用协程，我们不再需要像异步编程时写那么一堆callback函数，代码结构不再支离破碎，整个代码逻辑上看上去和同步代码没什么区别，简单，易理解，优雅。

我们使用协程，我们可以很简单地实现一个可以随时中断随时恢复的函数。

**一些 API 启动长时间运行的操作(例如网络 IO、文件 IO、CPU 或 GPU 密集型任务等)，并要求调用者阻塞直到它们完成。协程提供了一种避免阻塞线程并用更廉价、更可控的操作替代线程阻塞的方法：协程挂起。**

协程通过将复杂性放入库来简化异步编程。程序的逻辑可以在协程中顺序地表达，而底层库会为我们解决其异步性。该库可以将用户代码的相关部分包装为回调、订阅相关事件、在不同线程(甚至不同机器)上调度执行，而代码则保持如同顺序执行一样简单。

### 3.11 阻塞 vs 挂起

协程可以被挂起而无需阻塞线程。而线程阻塞的代价通常是昂贵的，尤其在高负载时，阻塞其中一个会导致一些重要的任务被延迟。

另外，协程挂起几乎是无代价的。不需要上下文切换或者 OS 的任何其他干预。

最重要的是，挂起可以在很大程度上由用户来控制，我们可以决定挂起时做些，并根据需求优化、记日志、拦截处理等。

## 9.15 协程的内部机制

### 9.15.1 基本原理

协程完全通过编译技术实现(不需要来自 VM 或 OS 端的支持)，挂起机制是通过状态机来实现，其中的状态对应于挂起调用。

在挂起时，对应的协程状态与局部变量等一起被存储在编译器生成的类的字段中。在恢复该协程时，恢复局部变量并且状态机从挂起点接着后面的状态往后执行。

挂起的协程，是作为Continuation对象来存储和传递，Continuation中持有协程挂起状态与局部变量。

关于协程工作原理的更多细节可以在这个[设计文档](https://github.com/Kotlin/kotlin-coroutines/blob/master/kotlin-coroutines-informal.md)中找到。

### 9.15.2 标准 API

协程有三个主要组成部分：

* 语言支持(即如上所述的挂起功能)，
* Kotlin 标准库中的底层核心 API，
* 可以直接在用户代码中使用的高级 API。
* 底层 API：kotlin.coroutines

底层 API 相对较小，并且除了创建更高级的库之外，不应该使用它。 它由两个主要包组成：

kotlin.coroutines.experimental 带有主要类型与下述原语：

* createCoroutine()
* startCoroutine()
* suspendCoroutine()

kotlin.coroutines.experimental.intrinsics 带有甚至更底层的内在函数如 ：

* suspendCoroutineOrReturn()

大多数基于协程的应用程序级API都作为单独的库发布：kotlinx.coroutines。这个库主要包括下面几大模块：

* 使用 kotlinx-coroutines-core 的平台无关异步编程
* 基于 JDK 8 中的 CompletableFuture 的 API：kotlinx-coroutines-jdk8
* 基于 JDK 7 及更高版本 API 的非阻塞 IO(NIO)：kotlinx-coroutines-nio
* 支持 Swing (kotlinx-coroutines-swing) 和 JavaFx (kotlinx-coroutines-javafx)
* 支持 RxJava：kotlinx-coroutines-rx

这些库既作为使通用任务易用的便利的 API，也作为如何构建基于协程的库的端到端示例。关于这些 API 用法的更多细节可以参考相关文档。