

# 协程简史，一文讲清楚协程的起源、发展和实现

陈蒙 郭霖 2022-10-08 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 /

北京时间10月5日下午，在瑞典首都斯德哥尔摩，瑞典皇家科学院宣布，将2022年诺贝尔化学奖授予美国化学家卡罗琳·贝尔托西、丹麦化学家摩顿·梅尔达尔和美国化学家卡尔·巴里·夏普莱斯，以表彰他们在点击化学和生物正交化学研究方面的贡献。其中，卡尔·巴里·夏普莱斯第二次获得诺贝尔化学奖。

## / 作者简介 /

大家好，国庆假期结束，我们又如期见面了。记得从今天起要连上7天班哦。

本篇文章来自陈蒙的投稿，文章主要分享了协程的起源，发展和常见实现，相信会对大家有所帮助！同时也感谢作者贡献的精彩文章。

陈蒙的博客地址：

<https://chenmeng.blog.csdn.net/?type=blog>

## / 前言 /

如果说大前端开发有什么金规铁律的话，那「不要阻塞主线程」肯定算一个。特别是面对网络请求等耗时任务时，异步编程是避免主线程卡死的常见解决方案。协程是诸多异步编程范式中的一种。相较于多线程、回调、Promise、响应式编程等其他异步编程范式，协程具有轻量级、代码可读性好等优点。当前主流编程语言要么已经支持了协程，要么正在支持的路上。本文将从起源、发展历史、常见语言实现等角度进行介绍，力求为大家展示协程的全貌。

## / 概念释义 /

### 协程定义

维基百科对协程定义的英文原文：

**Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed.**

是不是觉得晦涩难懂？每个单词都认识，但是连在一起就不知所云了。我们先往下看，后面再逐词解释。

除了维基百科，我们也许还见过别的定义，这些定义多是从某个角度描述协程特性的。

比如侧重其轻量级特性的：

**Coroutines are very light-weight threads.**

或者

**Coroutines are like very light-weight threads.**

虽只有一字之差，却是天壤之别。

侧重其暂停/恢复特性的：

**A coroutine is an instance of suspendable computation.**

侧重其表现形式的：

**A variant of functions that enables concurrency via cooperative multitasking.**

CoRoutine，其中 Co 是 Cooperative，意为平等、协作，指多段程序之间相互转移控制权；Routine 一词在高级语言中对应函数/方法等概念，即 Routine 与 Program、Subprogram、Function、Method、Subroutine、Procedure、Callable Unit 等词同义。故协程可以简称为协作的程序。

普通程序是主程序（Main）-子程序（Subroutine）的关系，一旦主程序调用（call）子程序，那么子程序独占控制权直到执行完成（return），是一种 call-return 的结构。通常，子程序一旦返回主程序，子程序的上下文就消失了，当主程序再次调用子程序时子程序需要重头开始执行一遍。

而协程不是 call-return 的结构，而是 suspend-resume 的结构，即在其中一段程序尚未执行完成时就向另一段程序移交控制权，而且前者会保存自己的上下文，以便稍后恢复现场继续执行。

从协程的调用行为上看，主程序既可以调用子程序，子程序也可以调用主程序，是一个双向的调用关系，并不是普通程序那种单向的调用-被调用的关系。

所以协程可以看做是对普通程序（Subroutine）的泛化（generalize），而普通程序是协程的一种特例（0个暂停点、单向的调用关系）。就像是打排球一样，如果把双方看成是有调用关系的程序，双方你来我往，地位是对等的，所以是一种 Co 的关系。引用发明协程这一概念的作者的原话，协程是：“as subroutines who act as the master program”。

普通程序和协程的对比见下图：



其实协程还代表了一种非抢占式的多任务并发的调度思想：协作式调度。每个任务执行完毕之后才将 CPU 的使用权转移给另一个任务，没有优先级和抢占，十分和谐。

我们可以参考做核酸对比下协作式和抢占式调度。我们把大白当做 CPU，每个居民当做一个任务。协作式调度就是一旦有居民坐在椅子上了，后面的人就必须等他主动让出椅子才能做核酸，否则就要一直等。而抢占式就有了优先级的概念，就是允许更高优先级的居民先做，即使前面的人已经坐在椅子上张开嘴巴了，此时后面来个人说“我家小朋友要上网课，能让我们先做吗？”，这时前面的人可以把位置让给这个更高优先级的人。



此时我们再回头看维基百科的定义，可能就清晰多了：

**Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed.**

广义上而言，协程是一个概念，就像肉夹馍一样，每个地方的具体做法都不完全相同。狭义上来说，协程是一段程序。介于二者之间，还有编程语言关键字、编译器/执行引擎支持、运行时系统等辅助系统，这些统称为组件（Component）。什么样的程序呢？是对普通程序的泛化的程序。能够支持协作式多任务调度，而且是通过代码执行的暂停和恢复的方式实现的。

从这个定义里面，我们可以得出协程最明显的特征：（1）转移控制权时保存上下文（2）可以被暂定，而后从暂停点开始继续执行。

“Talk is cheap, show me the code”。假如我们发明了一门新的编程语言叫 BePositiveBePatient（在做核酸的日子里，我们更需要保持积极的心态和耐心，大家千万不要翻译成“成为阳性、成为病人”），简称BPBP，它有两个关键字 yield/resume。我们用 BPBP 语言写出如下打印斐波那契数列的代码：

```
function main() {  
    var gen = fib()  
  
    var a = resume(gen) // 1  
    var b = resume(gen) // 2  
    var c = resume(gen) // 3  
    var d = resume(gen) // 5  
}  
  
// 斐波那契数列生成器, 1, 2, 3, 5  
function fib() -> Number {  
    yield 1  
    yield 2  
    yield 3  
    return 5  
}
```

执行到 resume 时就会进入 fib()，执行到 yield 时就会返回到 main()，yield 的作用跟 return 类似，但不会终止当前方法体的执行，执行过程见下图：



## 与线程的关系



协程是 1963 年正式提出的，而在之后的 1966 年，才有了线程（thread）的概念。

同一时刻，同一个 CPU 的某个核心（Core）上，只有一个进程的一个线程的一个协程（如果有）在运行。

一个进程包含至少一个线程（主线程），一个线程里面有 0 个或多个协程，一个协程是以线程为宿主进行的计算活动。协程一旦确定宿主线程，一般不会再更改。

进程是资源分配的基本单位，线程（内核态）是 CPU 调度的基本单位，协程对于 OS 来说是透明的。协程被认为是用户态的线程，协程的调度由用户完成。

进程向自己所属线程开放内存空间，线程有自己的堆栈、程序计数器、寄存器数据。

一个线程消耗的内存一般在 MB 级别，而协程占用内存一般在几十到几百字节，Goroutine 经过层层优化后占用 2KB。为了解决线程之间竞争的问题，每个线程还会在自己的内存空间中额外申请64MB内存来作为堆内存池，使得操作系统的内存无法支撑几万个线程的并发，但是这对协程却不是个问题。

线程上下文切换的成本在几十纳秒到几微秒间，当线程繁忙且数量众多时，这些切换会消耗绝大部分的CPU运算能力。



## 协程简史

1958 年，Melvin Conway 创造了协程（Coroutine）一词。

1963 年，协程这一概念正式发布于论文 Design of a Separable Transition-Diagram Compiler 中。

在这篇论文中，Conway 提出将 COBOL 编译过程中的词法分析和语法分析结合起来完成，而不是将二者看成是完全独立的步骤。词法分析和语法分析分别有自己的控制流，在 Conway 的方法里，这两个控制流可以保存自己运行的上下文并移交执行权给对方，并在合适的时候恢复自己的上下文并继续执行。这种概念就叫做协程。

## 论文原图



## 基本思想



- 1966 年，线程（thread）的概念被提出。
- 1968 年，Dijkstra 发表论文《GOTO 语句是有害的》，结构化编程的理念深入人心，自顶向下的程序设计思想成为主流，协程“跳来跳去”的执行行为类似 goto 语句，违背自顶向下的设计思想。
- 1979 年，Marlin 提交博士论文 Coroutines : A Programming Methodology, A Language Design, and An Implementation，是协程理论的集大成之作。
- 1980 年及之后的 20 余年，多线程成为并发编程的代名词，抢占式击败协作式成为主流的调度方式，协程逐渐淡出主流编程语言舞台。
- 2003 年，Lua v5.0 版本开始支持协程。
- 2005 年，Python 开始支持生成器和 yield/send 关键字，之后数年一直在演化。
- 2009 年，Go 语言问世，以 Goroutine 的方式支持并发编程，一代传奇拉开序幕。
- 2012 年，C# 开始支持 async 函数和 await 表达式，标志着协程王者归来。



- 2015 年，Python 支持 `async/await` 语法。
- 2017 年，`async/await` 纳入 ES2017 标准。
- 2017 年，Kotlin 另辟蹊径，以 `suspend` 关键字的形式实现了协程。
- 2019 年，Dart 支持 `Future`、`async/await` 语法。
- 2020 年，C++ 20 支持 `co_async/co_await`。
- 2022 年 3 月，JDK 19 预览版（Early-Access）中引入了一种新的并发编程模型（织布机计划）——虚拟线程，非最终版，可能随时被删除。

如果我们将上述 60 年的历史事件绘成图，一个圆代表一个事件，用面积大小代表其影响力大小，则有：



/ 种类划分 /

以下分类方式是参考论文 *Revisiting Coroutines* 得出。

### 按调用栈分类

实现协程的关键是暂停/恢复代码的执行，实现方法有两种：栈，状态机&闭包。

首先，抛开具体语言实现，我们来回顾下一个线程中方法调用的一般流程。方法的执行过程是借助方法调用栈实现的，栈内每个元素称为栈帧（Frame Stack）。还有一些和栈相关的寄存器，包括栈指针 ESP，指向栈的顶部，其存贮的地址随着栈帧的入栈和出栈而不断变化，总是指向栈的最后一个元素；基指针 EBP，指向当前运行的方法的一个固定位置，起到锚点的作用，为访问函数参数和局部变量提供参考点；EIP/PC 寄存器，指向 CPU 即将执行的下一条指令。调用一个子方法时该方法会以栈帧的形式入栈，子方法执行完成之后会出栈，结合寄存器内容的变化，CPU 可以完成方法调用过程。



假如我们有如下的代码：

```
function main() {  
  const a = 1  
  const b = 2  
  const c = add(a, b)  
  print(c)  
  return  
}  
  
function add(a, b) {  
  const d = a + b  
  return d  
}
```

则上述代码在执行过程中的调用栈变化如下图：



参考线程实现暂停/恢复的方式，我们可以把协程暂停点涉及的调用栈、EIP（程序计数器 PC）、EBP、ESP 等寄存器数据保存一份“快照”，当协程需要从暂停点恢复执行时，只需恢复这个快照即可。利用这种方式实现的协程被称为有栈协程（Stackful Coroutine），代表作 Go 和 Lua。有栈协程有时被称为 Fibers 或者用户态线程。

我们回顾一下设计模式中的迭代器模式。我们在开发过程中会用到各种数据结构，包括数组、列表、树等。每种数据结构有不同的遍历方式，有的用下标，有的用引用、指针。而实际我们只关心数据结构中是否还有元素以及获取下一个元素，并不关心其内部构造。根据面向对象编程中封装的原则，我们可以各种数据结构或集合实现一个统一的接口 Iterator，对外暴露 hasNext() 和 next() 方法，这样我们每次调用 next() 方法时就可以获取上次调用之后应该得到的元素，就跟迭代器“暂停”后又被“恢复”执行了一样。

但是我们知道，迭代器并没使用魔法，甚至也没有借助额外的“栈”去保存状态，迭代器宿主本身是个变量，可以保存状态，再借助适当的代码结构，达到“暂停”和“恢复”的执行效果。这也是实现协程的一种思路，常见手段是利用状态机和闭包，通过状态机和闭包来存储暂停点代码的现场信息。利用这种方式实现的协程称为无栈协程（Stackless Coroutine），代表作是 JavaScript、Kotlin 等，共同点是关键字 async/await。我们会在后续章节中通过 Kotlin 的协程实现来具体说明。

有栈和无栈协程的对比表如下：



## 按调度方式分类

协程的暂停和恢复涉及转移控制权，可以分为非对称协程和对称协程。

非对称协程一般有两个转移指令：暂停指令和继续指令。非对称协程暂停之后其控制权必须转移给继续指令所在的协程，二者之间存在一种较弱的调用方和被调用方的关系。非对称协程又被成为半对称协程（semi-symmetric coroutine）或半协程（semi coroutine）。非对称协程常见于生成器或迭代器场景，用于产生数据流。比如我们在之前章节创造的 BPBP 语言就属于非对称协程：



对称协程只有一个转移指令，各个协程之间的地位是“平等”的，控制权可以在多个协程之间转移：



比如 Go 的协程控制权是通过 channel 来完成转移的，channel 内部执行了类似 `yield/resume` 原语。

Go语言通过 `go` 关键字来构建协程。比如下面的程序实现了一个生产者协程（writer）和两个消费者协程（reader），生产者往 channel 里面写入数字，而消费者从 channel 中读取数字。

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var channel = make(chan int)
var readChannel <-chan int = channel
var writeChannel chan<- int = channel
var waitGroup = sync.WaitGroup{}

func main() {
    waitGroup.Add(3)

    go writer()
    go reader1()
    go reader2()

    waitGroup.Wait()
}

// reader1
```

```

func reader1() {
    fmt.Println("wait1 for read")
    for i := range readChannel {
        fmt.Println("read1", i)
    }
    fmt.Println("read1 end")
    waitGroup.Done()
}

// reader2
func reader2() {
    fmt.Println("wait2 for read")
    for i := range readChannel {
        fmt.Println("read2", i)
    }
    fmt.Println("read2 end")
    waitGroup.Done()
}

// writer
func writer() {
    for i := 0; i < 3; i++ {
        fmt.Println("write", i)
        writeChannel <- i
        time.Sleep(time.Second)
    }
    close(writeChannel)
    waitGroup.Done()
}

```

从代码运行结果我们可以看出，三个协程之间的控制权转移是“平等”的，没有严格的“定向”转移关系（可以使用在线编译工具复现）：

```

wait2 for read
wait1 for read
write 0
read2 0
write 1
read1 1
write 2
read2 2
read1 end
read2 end

```

下表是2021 IEEE 编程语言排行榜，我们列举了排名前 20 的语言以及 Objective-C 和 Lua，并标记了每种编程语言是否原生支持协程以及其所属协程种类（以下数据基于 2022.04 月得出）：



从上面的表格可以看出，主流编程语言都已经支持协程或者正在支持的路上。让人不由的感慨：编程语言也好卷，不支持协程的语言出门都不好意思跟人打招呼（Java：你直接报我身份证吧）。

## / 异步编程 /

异步编程可以看成是并发编程的近义词。需要区分下并行和并发。并行（Parallel）指物理上并行，即至少要有2个CPU或核心，硬件上支持同时至少2个线程同时运行。而并发（Concurrent）则是逻辑意义上的并行，即使硬件上只有1个CPU或1个核心，也可以通过时间分轮转等算法实现多任务调度，给人感官上的“并行”，其实我们知道某一时刻只有一个线程在运行。

我们首先来看下异步编程的必要性。下图给出了一些常见的计算机操作耗时对比：



我们近似的认为执行一条计算机指令耗时 1 纳秒，访问内存耗时 100 纳秒，SSD 随机读取耗时 16 微妙，一次网络请求耗时在 150 毫秒。

为了个更直观的进行对比，我们抽象出时间颗粒的概念。假设 1 个时间颗粒等于微观中的 1 纳秒、宏观世界中的 1 秒。在这种假设前提下有各个操作的耗时：



相对于常见的 IO 操作，包括文件读写、数据库读写和网络请求等，CPU 实在是太快了。让 CPU 等待本身就是对计算资源的严重浪费。这就好比是，一个网络操作对 CPU 说：你在这里不要动，我去买几个橘子。这一去就是 12.8 年。

对于大前端编程场景尤其如此，我们时刻需要考虑如何避免主线程被阻塞，否则会导致 App 无响应。

异步编程能有效避免主线程被阻塞，所以我们需要异步编程。



常见的异步编程解决方案有多线程、回调、Promise、Rx 和协程。以抖音发布视频为例，我们假设用户点击发布按钮后经历了三个阶段：

- 准备阶段，通过网络请求提交视频/图片、获取签名相关数据，为耗时操作，耗时在 100 毫秒数量级；
- 提交请求，向抖音后台上传视频、标题、话题等用户数据，为耗时操作，耗时在 100 毫秒数量级；
- 处理提交请求的结果，将成功/失败结果告知用户，在主线程中执行；

上述三个步骤之间具有依赖关系，后者必须等前者执行完成才能开始，在不考虑阻塞主线程的情况下，我们使用同步代码（假设是我们发明的 BPBP 语言，下同）。实现如下：

```
function postItem(item: Item) { // 一个 Item 为一条待发版的抖音视频数据模型
    val token = preparePost() // 网络请求，耗时操作
    val post = submitPost(token, item) // 网络请求，耗时操作
    processPost(post)
}

function preparePost(): Token {...}
function submitPost(t: Token, i: Item): Post {...}
function preparePost(p: Post) {...}多线程
```

## 多线程

假设我们的 BPBP 语言也是支持多线程的，用 thread 关键字构建线程。那么我们可以用单独的线程执行耗时操作：

```
function postItem(item: Item) {
    thread {
        val token = preparePost()
        val post = submitPost(token, item)
        processPost(post)
    }
}

function preparePost(): Token {...}
function submitPost(t: Token, i: Item): Post {...}
function preparePost(p: Post) {...}
```

同时我们也应该意识到多线程的方式存在诸多缺点：

- 创建开销较大，线程调度时需要上下文切换；
- 数量受限，能启动的最大线程数受操作系统限制，这个限制对于后端应用的影响尤为明显；
- 平台受限，某些平台比如 JavaScript 不支持自定义线程，注意 WebWorker 是浏览器提供的 API 而非 JavaScript 的能力；
- 数据同步带来额外性能开销；
- 对象生命周期不一致造成的内存泄漏；
- 使用难度大，调试多线程、线程安全让人头大；

一提到多线程编程，往往会想到那令人匪夷所思的执行结果，就像下面的2张图一样。

### 你期待的效果



### 实际的效果



## 回调

如果我们用回调的方式解决该问题：

```
function postItem(item: Item) {  
    preparePostAsync { token ->  
        submitPostAsync(token, item) { post ->  
            processPost(post)  
        }  
    }  
}
```

回调的方式简单易懂、开销低、适用范围广，几乎适用于任何异步任务，但是也存在缺陷：

- 嵌套地狱，嵌套层数多，不易理解，维护困难；
- 不易处理异常，如果其中一层回调发生异常，异常的传递和处理都比较困难；
- 多个回调收口困难，比如要等多个回调都完成之后统一执行某个动作；

- 对于回调执行的线程容易出错，常见的问题是回调在子线程执行但是却操作了UI更新；
- 对 for/while、try-catch 等场景不友好；

只要你愿意，你甚至可以写出下面的嵌套代码：



## Promise

Promise 背后的思想是当我们触发一个耗时操作时，我们同步的获取一个实例，在后续某个时间点通过这个实例操作耗时操作的结果。即，这是一个耗时操作，我没法立即给你结果，但是我可以给你一个承诺（Promise），承诺你在不久的 Future 把执行结果告知你。

在不同的平台，这个实例有不同的叫法，比如 Promise|JavaScript、CompletableFuture|Java、Future|Dart、Deferred|Kotlin 等。

因为 Kotlin 是一个跨平台的语言，可以和 Java、Swift 等进行混合编程，甚至可以导出 JavaScript 代码，为了避免引起混乱，最好避开 Promise、Future 这些已有名词，从而定了个 Future 的近义词 Deferred。

这是后起之秀的常见烦恼，纵然你可以站在巨人的肩膀上，规避前辈（Java、JavaScript 等）踩过的坑，但是也要承受这种“无名可用”的尴尬。包括我们常见的 Vue.js，最初尤雨溪打算将其命名为 view.js，但是这个名字被占用了，他就把 view 用谷歌翻译成各种语言，发现

法语的 `vue` 还没人用，就用了这个名字。其他行业也有类似的烦恼，特斯拉的车型命名，本想凑个 `SEXY`，但是 `Model E` 被福特抢注了，只能将字母 `E` 翻转下成为 `3`。

```
function postItem(item: Item) {  
  preparePostAsync()  
    .then { token -> submitPostAsync(token, item) }  
    .then { post -> processPost(post) }  
}  
  
function preparePostAsync(): Promise<Token> {...}  
function submitPostAsync(): Promise<Post> {...}  
function processPost(post: Post) {...}
```

`Promise` 采用了一种调用链式的编程模型，简化了异步操作，避免了回调地狱，但是同样也存在一些问题，比如：

- 不同平台上 API 命名差异，有的为 `thenCompose()`、`thenAccept()` 等等；
- 不符合同步编程习惯，对 `for/while`、异常处理不友好；
- 对每一步的返回值类型有特殊要求，必须是 `Promise`，而不能是实际的数据类型；
- 错误处理变得复杂，不易将不同阶段产生的错误一路传递下去；
- 不同阶段之间共享数据困难；

## 响应式编程

`Reactive Extension`（简称 `Rx`）编程模型最初由 `Erik Meijer` 引入 `C#`，但是当时并未受到业界重视。

直到 `Netflix` 将其移植到 `Java` 平台，并产生了 `RxJava`，从此一发不可收，各种 `RxXX` 如 `RxJS`、`RxSwift` 等开始涌现。

`Rx` 的核心思想是将一切都当做是可观测的数据流，实现方式是观察者设计模式+一系列的扩展方法。

跟 `Promise` 类似，`Rx` 的写法跟我们平时写的同步代码也存在差异，这其实也是一种新的编程模型。

Rx 由于封装的过于“简洁”，导致其调试困难，数量众多的扩展函数让人眼花缭乱，存在较抖的学习曲线，容易出错、不易维护。

同时，Rx 在各个平台上的 API 基本具有一致性，而且处理错误的方式也更加友好。

假如我们有 RxBPBP，那么用 RxBPBP 实现笔记发布流程的代码大概是：

```
function postItem(final item: Item) {
    Observable
    .create { subscriber -> {
        Token t = preparePost(); // 获取签名
        subscriber.onNext(t);
        subscriber.onCompleted();
    }
    }
    .subscribeOn(Schedulers.io())
    .map { token -> {
        val p: Post = submitPost(token, item); // 发送网络请求
        return p;
    }
    }
    .subscribeOn(Schedulers.io())
    .subscribeOn(Schedulers.main())
    .subscribe{ {
        function onCompleted() {}
        function onError(e) {}
        function onNext(post) { processPost(post) } // 处理结果
    }
    }
}
```

## 协程及常见实现

最近十几年，互联网、移动互联网、物联网、车联网等产生的网络请求数量激增，但是大部分请求并不是 CPU 计算密集型而是 IO 密集型的，大部分的请求都是 请求到来——少量计算——调用公共服务——读写数据库——返回数据，当处于读写阻塞时，线程处于阻塞状态，内核调度器会将这个线程挂起，执行其他线程，如果之前的阻塞解除了，再切换上下文，执行之前挂起的线程，CPU 大部分时间花在了切换上下文上。而线程是宝贵的计算资源，数量有限，本身占用存储资源，最好是一直跑，别阻塞、别切换上下文。

以下图为例，我们可以构建三个线程 thread\_0、thread\_1、thread\_2 去处理网络请求，每个线程处理一个请求，但是每个线程大部分时间都处于 IO 阻塞状态。我们也可以只创建一个

线程，而用三个协程去处理这三个网络请求，当请求0处理 IO 时将其对应的协程挂起，继续运行请求2，当请求2处理 IO 时再将其挂起去处理请求1，当请求1挂起时切换到已经就绪的请求2、请求0、请求1。通过这种方式来实现十万级甚至百万级的高并发。



协程这一异步编程模型的优势包括：

- 轻量级：（1）协程的创建、销毁、调度发生在用户态，避免了系统内核级线程频繁切换带来的 CPU 资源浪费，提升了 CPU 的利用率和吞吐量；（2）内存占用小，线程的内存占用在 MB 级别，系统内存的制约导致我们无法开启更多线程实现高并发，而在协程编程模式下，可以轻松有十几万协程，这是线程无法比拟的；
- 提升开发效率：（1）借助协程我们可以以近似同步代码的方式完成异步操作，方便代码的阅读和后续维护；（2）通过结构化并发限定控制域，减少内存泄漏

协程的局限性：

- 不符合结构化编程的理念
- 对于单线程语言，无法利用多核优势
- `async/await` 限制返回值类型
- 对于Goroutine 排查内存类（Goroutine 逃逸、内存泄漏）错误，排查难度大

- 下面我们以 JavaScript、Dart、Kotlin、Swift 和 Go 为例，看下其对协程的实现情况。

## JavaScript

ES2015 引入 Promise/a+、生成器 Generators、关键词 yield。生成器可以赋予函数执行暂停/保存上下文/恢复执行状态的功能，新关键词 yield 使生成器函数暂停。语法如下：

```
function* gen(){
  yield 1;
  yield 2;
  return 3;
}

// 生成器返回值类型
interface Iterator {
  next(): IteratorResult;
}

// yield 表达式返回值类型
interface IteratorResult {
  value: any; // yield 表达式返回值
  done: boolean; // 是否还有其他 yield, 有 - false, 无 - true
}
```

ES2017 引入 async/await 语法糖，二者均为关键字，async 用于修饰异步函数（将 Generator 函数和自动执行器，包装在一个函数里），此函数需要返回一个 Promise 对象。await 修饰表达式，可以等待一个 Promise 对象 resolve，并拿到结果。

Promise 中也利用了回调函数，在 then 和 catch 方法中都传入了一个回调函数，分别在 Promise 被满足和被拒绝时执行，这样就就能让它能够被链接起来完成一系列任务。

JavaScript 的协程成熟体是 async/await，我们用 async/await 方式实现上述案例如下：

```
async function postItem(item) {
  const token = await preparePostAsync()
  const post = await submitPostAsync(token, item)
  processPost(post)
}

function preparePostAsync() {
  return new Promise(resolve => {
    setTimeout(() => {
```



```

        resolve('token');
    }, 2000);
});
}

function submitPostAsync() {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve('post');
        }, 2000);
    });
}

function processPost() {}

```

原理：async/await 是由 Promise + Generator 实现的语法糖。将 async/await 转成 Promise+Generator：

```

// 转换成 Promise 的形式
function* postItem(item) {
    let token = yield preparePostAsync()
    let post = yield submitPostAsync(token, item)
    processPost(post)
}

const iterator = postItem({});
iterator.next().value
    .then((val) =>
        iterator.next(val).value)
    .then((data) => {
        processPost(data);
    });

// 生成器返回值类型
interface Iterator {
    next(): IteratorResult;
}

// yield 表达式返回值类型
interface IteratorResult {
    value: any;    // yield 表达式返回值
    done: boolean; // 是否还有其他 yield, 有 - false, 无 - true
}

runner(postItem);

```

可能你会说：这种把每个yield翻译成Promise的方式有点傻，太不通用了，即使要毁灭地球，我也要写个以 planet 作为入参的通用函数，然后把地球作为入参传进去。是的，其实我们可以更一般化的实现这个转换过程（注意下述代码忽略了对异常的处理）：

```
// 更一般的形式
function runner(genFn) {
  let itr = genFn(); // Iterator 类型，genFn() 不会立即执行，直到 .next() 方法被调用

  function run(arg) {
    let result = itr.next(arg); // IteratorResult 类型，调用next()开始执行到下个 yield，返回一个
    if (result.done) {
      return result.value;
    } else {
      return Promise.resolve(result.value).then(run); // 等待 Promise 结果，将结果作为 arg
    }
  }

  run();
}
```

在 Generator+yield 在 V8 引擎中是怎么实现的呢？

Generator+yield 属于无栈协程，仍然跟普通的函数用的是同一个调用栈，并没有借助额外的栈去保存暂停点的现场信息，yield 关键字的作用类似属于 return，但是利用状态机和闭包完成了保存上下文的目的，yield 执行之后 Generator 函数仍然会被出栈，当 .next() 方法调用时，再将 Generator 方法入栈，根据之前保存的现场数据（包括程序计数器）来从 yield 暂停点开始继续往下执行。

## Dart

Dart async/await 的用法几乎跟 JavaScript 一模一样，甚至连单线程模型、消息队列等都一样，只是在语法上 async 的位置有所不同。

Dart 的目标是取代 JavaScript，虽然是单线程语言，但是其使用 Isolate 实现了多线程。

用 Dart 实现该案例：

```
void main() {
  postItem(Item());
}

void postItem(Item item) async {
  var token = await preparePost(); // 请求签名
  var post = await submitPost(token, item); // 发送网络请求
}
```

```
    processPost(post);                // 处理结果
}

Future<Token> preparePost() {
    return Future.delayed(Duration(seconds: 3), () => Token()); // 返回 Future
}

Future<Post> submitPost(Token token, Item item) {
    return Future.delayed(Duration(seconds: 5), () => Post()); // 返回 Future
}

void processPost(Post post) {}
```

## Kotlin

Kotlin 是一门 JVM 编程语言，跟 Java 一样，经过编译后生成字节码运行于 JVM 上。



虽然 `async/await` 方式可以让我们写出近似同步的代码，避免了嵌套地狱，但是也带来了新的问题。最明显的就是 `async` 修饰的方法的返回值类型受限，必须是 `Promise/Future`，这种类型代表异步操作。这就导致了我们的代码复用受限，比如有些逻辑我们需要在多个地方使用，有的地方返回异步的 `Promise/Future`，而有的地方可能就是想要用于子线程中，就是想要同步操作。而且 `async/await` 的方式对异步操作的用法限制过死，限制了开发者发挥的空间，Kotlin 想仅仅制定一个简单而且底层的规则，然后让开发者自有发挥、构建其他的第三方库。

基于这些考虑，Kotlin 并未提供 `async/await` 关键字，而是仅仅提供了一个 `suspend` 关键字，用其修饰耗时函数，而且对返回值没有特殊的类型限制。进而通过这个关键字构建出其他的协程操作，比如 `async/await` 扩展方法。

跟 JavaScript/Dart 不同的是，Kotlin 可以构造线程（线程池），我们可以指定协程在哪类线程中开始执行以及恢复执行。

另外值得注意的是构建 Kotlin 协程的写法，与构建线程的方式非常相似，而且在 `suspend` 方法内部写的代码就跟同步代码一样：

```
fun main() {
    GlobalScope.launch(Dispatchers.Main) { // 在主线程中触发协程
        postItem(Item())
    }
    Thread.sleep(100)
}

suspend fun postItem(item: Item) {
    val token = preparePost() // 暂停点1: 获取签名
    val post = submitPost(token, item) // 暂停点2: 发送网络请求
    processPost(post) // 处理结果
}

suspend fun preparePost() : Token = withContext(Dispatchers.IO) { ... }
suspend fun submitPost(t: Token, i: Item) : Post = withContext(Dispatchers.IO) { ... }
fun processPost(post: Post) { ... }
```

使用 Kotlin `async/await` 方式实现，`async()` 返回的是一个类似 Promise/Future 的类型——`Deferred`（为了避免与 Java/JavaScript 混合编程是混淆）：

```
fun postItem(item: Item) {
    GlobalScope.launch {

        // 获取签名
        val deferredToken: Deferred<Token> = async { preparePost() }
        val token: Token = deferredToken.await()

        // 发送网络请求
        val deferredPost: Deferred<Post> = async { submitPost(token, item) }
        val post = deferredPost.await()

        // 处理结果
        processPost(post)
    }
}
```

```
suspend fun preparePost() : Token = withContext(Dispatchers.IO) { ... }
suspend fun submitPost(t: Token, i: Item) : Post = withContext(Dispatchers.IO) { ... }
fun processPost(post: Post) { ... }
```

注意，在耗时方法里面都有 `withContext(Dispatchers.IO)`，表示这个方法要在 IO 线程中运行。

我们可以借助安卓开发 IDE AndroidStudio 将 Kotlin 的协程代码反编译成 Java 代码，从中窥探其实现方法，但是为了避免陷于繁琐的代码细节之中，我们只讲原理。

Kotlin 没有借助 JVM 额外的魔法，是怎么实现协程的暂停和恢复的呢？

既然 Kotlin 有自己的编译器，那可不可以直接将协程代码转成回调嵌套呢？是可以的，但是并未这么做，而是转换成了状态机。

原理：编译器会将上述 `suspend` 相关代码转换成状态机相关的代码。

首先 `suspend` 函数会被编译器转成 CPS（Continuation-Passing Style，小名：回调函数）的形式。

比如：

```
suspend fun preparePost() : Token { ... }
```

会被转换成：

```
fun preparePost(Continuation<Token> cont) { ... }

public interface Continuation<in T> {
    val context: CoroutineContext // 协程上下文，包括所运行的线程等
    fun resumeWith(result: Result<T>) // 暂停点
}
```

然后为每个暂停点添加一个标记：

```
suspend fun postItem(item: Item) {
```

```

// state 0
val token = preparePost() // 暂停点1: 获取签名

// state 1
val post = submitPost(token, item) // 暂停点2: 发送网络请求

// state 2
processPost(post) // 处理结果
}

```

再添加状态机：

```

suspend fun postItem(item: Item, cont: Continuation) {
    var stateMachine = cont as? ThisSM ?: object: Continuation { // 可复用状态机
        var state = 0
        override fun resumeWith() {
            postItem(null, this)
        }
    }

    switch(stateMachine.state) {
        case 0:
            stateMachine.state = 1
            stateMachine.item = item
            preparePost(stateMachine) // 暂停点1: 获取签名
            return
        case 1:
            stateMachine.state = 2
            val token = stateMachine.result as Token
            val item = stateMachine.item
            submitPost(token, item, stateMachine) // 暂停点2: 发送网络请求
            return
        case 2:
            val post = stateMachine.result as Post
            processPost(post, stateMachine) // 处理结果
            return
    }
}

suspend fun preparePost() : Token = withContext(Dispatchers.IO) { ... }

```

其执行过程如下图：



整个执行过程是借助 stateMachine 在主线程中执行了 3 遍 postItem(), 每1遍对应1个状态, 每1遍执行1个 case 分支, 执行完 case 分支立即 return。执行1个 case 分支不代表同步执行完成, 也有可能是只是切换到 IO 线程, 即触发耗时操作的执行。

在将耗时方法切到IO线程之后, 主线程就空闲出来去处理其应该做的事情, 比如视图渲染等, 从而避免主线程被卡死。

由此可以看出, Kotlin 并未借助栈来实现协程的暂停和恢复, 而是通过状态机和闭包来实现的, 状态机和闭包作为局部变量是存储在堆上的, 占用的内存空间更小。

Kotlin 协程的暂停和恢复同样离不开线程的切换, 从某种程度上说, Kotlin 协程本质上是一个线程调度的框架。

一旦启动一个协程, 如何取消呢? 不取消会产生资源泄漏的问题。JavaScript是通过 AbortController, 而Kotlin与Swift 类似, 是通过结构化并发的方式实现的。

## Swift

在 Swift 中我们使用 Task 构建协程实例, 一个 Task 就是一个协程, 用 Swift 协程实现笔记发布流程:

```
override func viewDidLoad() {
    super.viewDidLoad()
    Task {
        let token = try await preparePost()
        let post = try await submitPost(token, item)
        processPost(post)
    }
}

func preparePost() async throws -> Token
```

```
func submitPost(_ t: Token, _ i: Item) async throws -> Post
```

```
@MainActor
```

```
func processPost(_ p : Post) async throws -> Void
```

使用 `async/await` 不会阻塞主线程，在一个 `Task` 中使用 `await` 时后面的任务将会被挂起，等到 `await` 任务执行完后，会回到被挂起的地方继续执行。Swift 的协程也是一种结构化并发（Structured Concurrency），类似于 Kotlin 中的 `Job`，`Task` 之间具有树状父子关系，取消父协程会同步取消其子孙协程。

## Go

虽然我们这边文章主要面向大前端读者，但是说到协程这个话题，无论如何绕不开 Go 语言的。但是由于笔者本人才疏学浅（不夸张的说，光一个 GMP 模型估计都能写本书），实在不敢班门弄斧，我们浅尝辄止。

Go 中没有线程概念，只有 Goroutine，可以看做是轻量级的线程，Goroutine 是实现高并发的关键。

- Goroutine 有栈协程，每个协程有独立的栈，而栈既保留了变量的值，也保留了方法的调用关系、参数和返回值。
- Go 采用 GMP 模型实现高并发，GMP 分别代表：
- G - Goroutine，Go 协程，是参与调度与执行的最小单位
- M - Machine，指的是系统级线程
- P - Processor，指的是逻辑处理器，P 关联了的本地可运行 G 的队列（也称为 LRQ），最多可存放 256 个 G

GMP 调度流程大致如下：

- 线程 M 想运行任务就需得获取 P，即与 P 关联
- 然从 P 的本地队列 (LRQ) 获取 G
- 若 LRQ 中没有可运行的 G，M 会尝试从全局队列 (GRQ) 拿一批 G 放到 P 的本地队列，
- 若全局队列也未找到可运行的 G 时候，M 会随机从其他 P 的本地队列偷一半放到自己 P 的本地队列
- 拿到可运行的 G 之后，M 运行 G，G 执行之后，M 会从 P 获取下一个 G，不断重复下去





在Go语言中，我们不需要使用 `async/await`、`suspend` 关键字，也不需要对外耗时函数返回结果进行特别修饰、做特殊包装，只需一个 `go` 关键字即可。

可以说 Go 语言是协程的终极形态，用 Go 写异步操作就跟同步代码没有什么区别，在开发体验上达到了完美的程度。

我们用 Go 代码实现笔记发布流程（可用在线编译环境复现）：

```
package main
```

<https://mp.weixin.qq.com/s/2q1dxT4QxsuxP3wmrUa4Ag>

```
import (
    "fmt"
    "sync"
)

var waitGroup = sync.WaitGroup{}

func main() {
    waitGroup.Add(1)
    go postItem(Item{})
    waitGroup.Wait()
}

func postItem(item Item) {
    fmt.Println("Hello, 世界")

    var token = preparePost()
    var post = submitPost(token, item)
    processPost(post)

    waitGroup.Done()
}

func preparePost() Token {
    fmt.Println("preparePost")
    return Token{}
}

func submitPost(token Token, item Item) Post {
    fmt.Println("submitPost")
    return Post{}
}

func processPost(post Post) {
    fmt.Println("processPost")
}

type Token struct{}
type Post struct{}
type Item struct{}
```

## 推荐阅读：

[我的新书，《第一行代码 第3版》已出版！](#)

[Android 13运行时权限变更一览](#)

[PermissionX 1.7发布，全面支持Android 13运行时权限](#)

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

为什么C++中有函数指针还需要std::function?  
码农的荒岛求生



最适合孩子入门的十大编程语言  
CSDN



总结 mysql 的所有 buffer，一网打尽就这篇了！  
yes的练级攻略

