协程

协程的历史

有的同学可能认为协程是一种比较新的技术，然而其实协程这种概念早在1958年就已经提出来了，**要知道这时线程的概念都还没有提出来。**

到了1972年，终于有编程语言实现了这个概念，这两门编程语言就是Simula 67 以及Scheme。

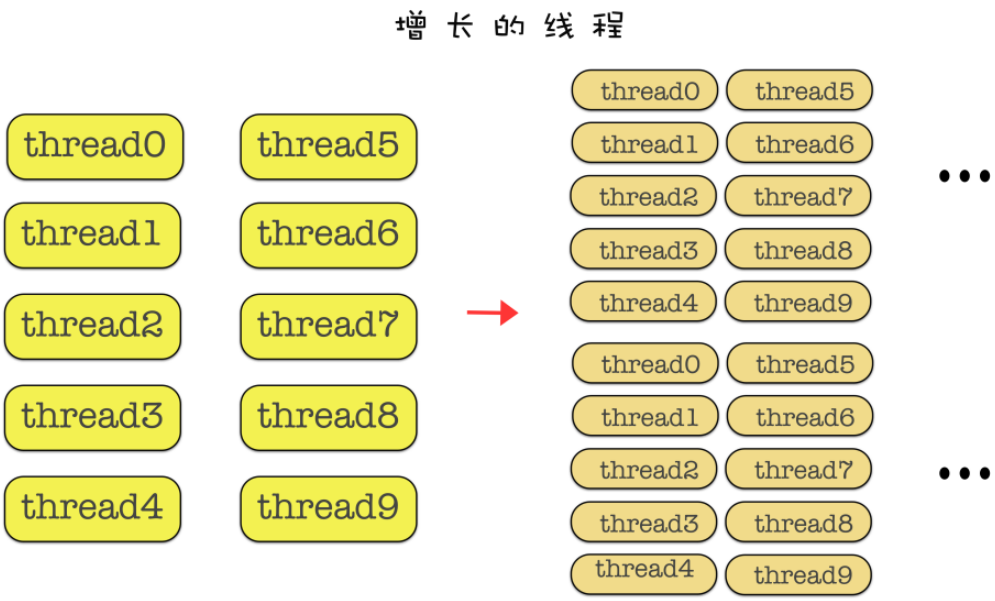
但协程这个概念始终没有流行起来，甚至在1993年还有人考古一样专门写论文挖出协程这种古老的技术。

因为这一时期还没有线程，如果你想在操作系统写出并发程序那么你将不得不使用类似协程这样的技术，后来线程开始出现，操作系统终于开始原生支持程序的并发执行，就这样，协程逐渐淡出了程序员的视线。

直到近些年，随着互联网的发展，尤其是移动互联网时代的到来，服务端对高并发的要求越来越高，协程再一次重回技术主流，各大编程语言都已经支持或计划开始支持协程。

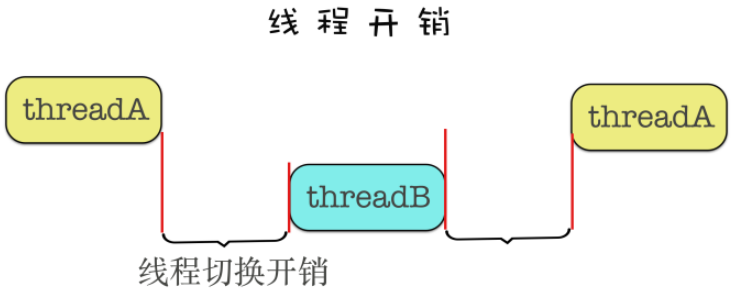
为什么需要协程

我们在执行多任务的时候通常采用多线程的方式并发执行。我们以最近非常火热的电商促销茅台为例，不管茅台是在缓存中还是后端的数据，最开始的用户也就是10个，每当收到10条付款信息就开启10个线程去查询数据库，此时用户量少，马上就可返回，第二天增加到100人，使用100个线程去查询，感觉确实效果不错，加大促销力度，当同时出现1000个人的时候感觉到有点吃力了



1000-10000，看了前面的内容应该清楚创建销毁线程还是挺费资源的，假设每个线程占用 4M内存空间，那么10000个线程大概需要消耗 39G 内存，可是服务器也就 8G 内存。

此时的方案要么增加服务器要么提升代码效率。多个线程在进行作业的时候，难免会遇到某个线程等待 IO 的情况，此时会阻塞当前线程切换到其他线程，使得其他线程照常执行，线程少的时候没什么问题，当线程数量变多就会出现问题，线程数量的增加不仅占用非常多的内存空间且过多的线程的切换也会占用大量的系统时间。

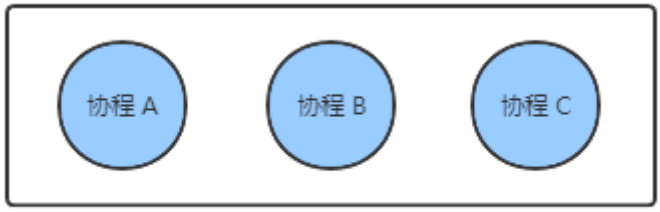


协程的作用

大多数web服务跟互联网服务本质上大部分都是 IO 密集型服务，IO 密集型服务的瓶颈不在CPU处理速度，而在于尽可能快速的完成高并发、多连接下的数据读写。以前有两种解决方案：

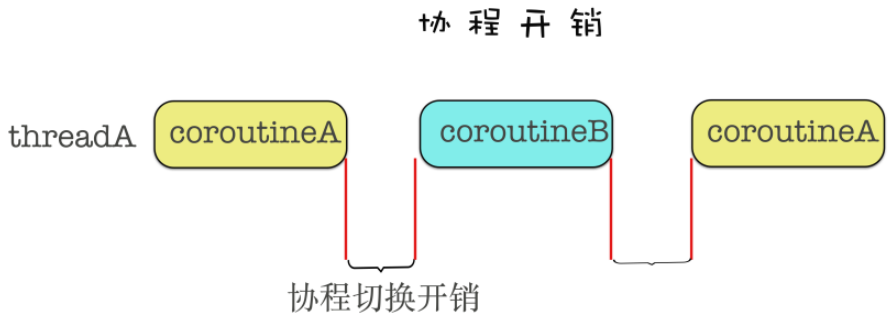
1. 多进程：存在频繁调度切换问题，同时还会存在每个进程资源不共享的问题，需要额外引入进程间通信机制来解决。
2. 多线程：高并发场景的大量 IO 等待会导致多线程被频繁挂起和切换，非常消耗系统资源，同时多线程访问共享资源存在竞争问题。

此时协程出现了，协程 Coroutines 是一种比线程更加轻量级的微线程。类比一个进程可以拥有多个线程，一个线程也可以拥有多个协程。可以简单的把协程理解成子程序调用，每个子程序都可以在一个单独的协程内执行。



线程 1

协程运行在线程之上，当一个协程执行完成后，可以选择主动让出，让另一个协程运行在当前线程之上。**协程并没有增加线程数量，只是在线程的基础之上通过分时复用的方式运行多个协程**，还有关键一点是它的切换发生在用户态，所有也不存在用户态到内核态的切换，代价更低



类比上面，我们只需要启动 100 个线程，然后每个线程跑100个协程就可以完成上述同时处理10000个任务。

使用协程的注意事项

刚说协程运行于线程之上，如果线程等待 IO 的时候阻塞了，这时候会出现什么情况？其实操作系统主要关心线程，协程调用阻塞IO的时候，操作系统会让进程处于阻塞状态，此时当前的协程和绑定在线程之上的协程都会陷入阻塞而得不到调度，这样就很难受了

因此在协程中不能调用导致线程阻塞的操作，比如打印、读取文件、Socket接口等。`协程只有和异步IO结合`起来才能发挥最大的威力。并且**协程只有在IO密集型的任务中才会发挥作用。**

协程 + 异步IO

- 比较简答的思路是当调用阻塞 IO 的时候，重新启动一个线程去执行这个操作，等执行完成后，协程再去读取结果，这是不是和多线程很像
- 将系统 IO 进行封装，改为异步调用的方式，此时需要大量的工作，所以需要寄生于编程语言的原生支持

所以对于计算密集型的任务不太建议使用协程，计算机密集型的任务需要大量的线程切换，线程切换涉及太多的资源交换