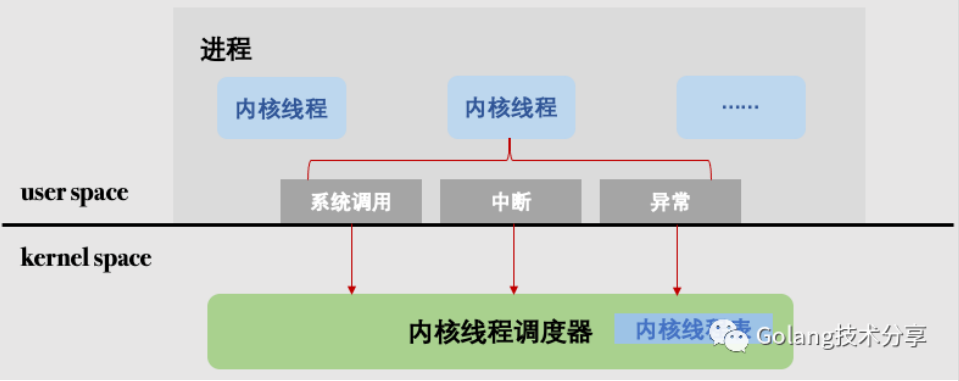
一文读懂channel设计

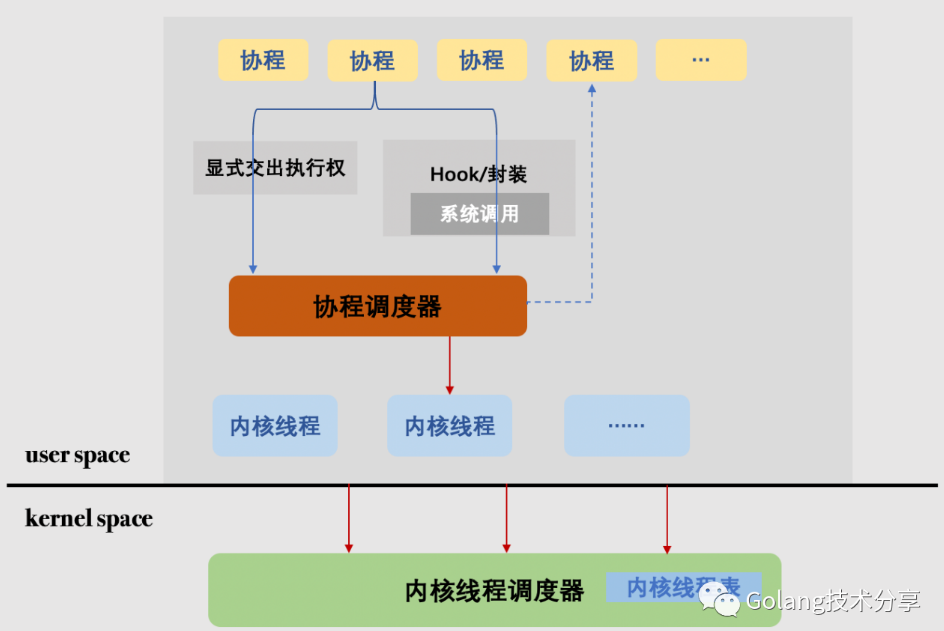
## 为什么会有goroutine？

现代操作系统中为我们提供了三种基本的构造并发程序的方法：多进程、I/O多路复用和多线程。其中最简单的构造方式当属多进程，但是多进程的并发程序，由于对进程控制和进程间通信开销巨大，这样的并发方式往往会很慢。

因此，操作系统提供了更小粒度的运行单元：线程（确切叫法是内核线程）。它是一种运行在进程上下文中的逻辑流，线程之间通过操作系统来调度，其调度模型如下图所示。



多线程的并发方式，相较于多进程而言要快得多。但是由于线程上下文切换总是不可避免的陷入内核态，它的开销依然较大。那么有没有不必陷入内核态的运行载体呢？有，用户级线程。用户级线程的切换由用户程序自己控制，不需要内核干涉，因此少了进出内核态的消耗。

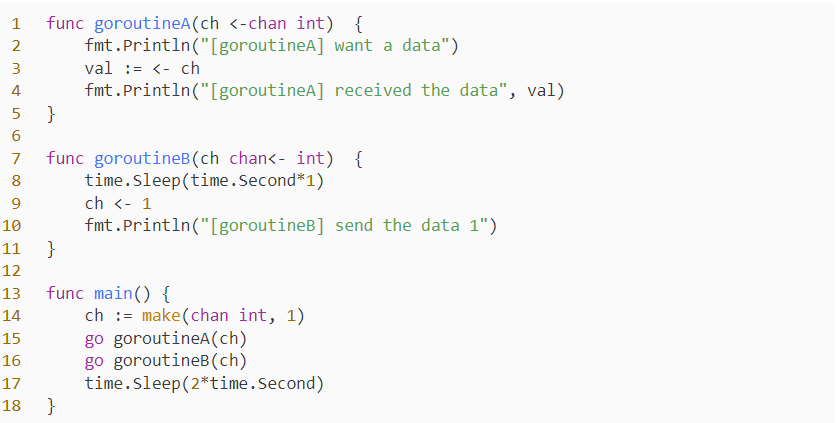


这里的用户级线程就是协程（coroutine），它们的切换由运行时系统来统一调度管理，内核态并不知道它的存在。协程是抽象于内核线程之上的对象，一个内核线程可以对应多个协程。但最终的系统调用仍然需要内核线程来完成。注意，线程的调度是操作系统来管理，是一种抢占式调度。而协程不同，协程之间需要合作，会主动交出执行权，是一种协作式调度，这也是为何被称为协程的原因。

Go天生在语言层面支持了协程，即我们常说的goroutine。Go的runtime系统实现的是一种M:N调度模型，通过GMP对象来描述，其中G代表的就是协程，M是线程，P是调度上下文。在Go程序中，一个goroutine就代表着一个最小用户代码执行流，它们也是并发流的最小单元

## channel的存在定位

从内存的角度而言，并发模型只分两种：基于**共享内存**和基于**消息通信**（内存拷贝）。在Go中，两种并发模型的同步原语均有提供：sync.\*和atomic.\*代表的就是基于共享内存；channel代表的就是基于消息通信。而Go提倡后者，它包括三大元素：goroutine（执行体），channel（通信），select（协调）。



## channel源码解析