要理解这个事儿首先得了解操作系统是怎么玩线程的。一个线程就是一个栈加一堆资源。操作系统一会让cpu跑线程A，一会让cpu跑线程B，靠A和B的栈来保存A和B的执行状态。每个线程都有他自己的栈。  
但是线程又老贵了，花不起那个钱，所以go发明了goroutine。大致就是说给每个goroutine弄一个分配在heap里面的栈来模拟线程栈。比方说有3个goroutine，A,B,C，就在heap上弄三个栈出来。然后Go让一个单线程的scheduler开始跑他们仨。相当于 { A(); B(); C() }，连续的，串行的跑。  
和操作系统不太一样的是，操作系统可以随时随地把你线程停掉，切换到另一个线程。这个单线程的scheduler没那个能力啊，他就是user space的一段朴素的代码，他跑着A的时候控制权是在A的代码里面的。A自己不退出谁也没办法。  
所以A跑一小段后需要主动说，老大（scheduler），我不想跑了，帮我把我的所有的状态保存在我自己的栈上面，让我歇一会吧。这时候你可以看做A返回了。A返回了B就可以跑了，然后B跑一小段说，跑够了，保存状态，返回，然后C再跑。C跑一段也返回了。  
这样跑完{A(); B(); C()}之后，我们发现，好像他们都只跑了一小段啊。所以外面要包一个循环，大致是：

goroutine\_list = [A, B, C]

while(goroutine):

for goroutine in goroutine\_list:

r = goroutine()

if r.finished():

goroutine\_list.remove(r)

比如跑完一圈A，B，C之后谁也没执行完，那么就在回到A执行一次。由于我们把A的栈保存在了HEAP里，这时候可以把A的栈复制粘贴会系统栈里（我很确定真实情况不是这么玩的，会意就行），然后再调用A，这时候由于A是跑到一半自己说跳出来的，所以会从刚刚跳出来的地方继续执行。比如A的内部大致上是这样

def A:

上次跑到的地方 = 找到上次跑哪儿了

读取所有临时变量

goto 上次跑到的地方

a = 1

print("do something")

go.scheduler.保存程序指针 // 设置"这次跑哪儿了"

go.scheduler.保存临时变量们

go.scheduler.跑够了\_换人 //相当于return

print("do something again")

print(a)

第一次跑A，由于这是第一次，会打印do something，然后保存临时变量a，并保存跑到的地方，然后返回。再跑一次A，他会找到上次返回的地方的下一句，然后恢复临时变量a，然后接着跑，会打印“do something again"和1

所以你看出来了，这个关键就在于每个goroutine跑一跑就要让一让。一般支持这种玩意（叫做coroutine）的语言都是让每个coroutine自己说，我跑够了，换人。goroutine比较文艺的地方就在于，他可以来帮你判断啥时候“跑够了”。

其中有一大半就是靠的你说的“异步并发”。go把每一个能异步并发的操作，像你说的文件访问啦，网络访问啦之类的都包包好，包成一个看似朴素的而且是同步的“方法”，比如string readFile（我瞎举得例子）。但是神奇的地方在于，这个方法里其实会调用“异步并发”的操作，比如某操作系统提供的asyncReadFile。你也知道，这种异步方法都是很快返回的。  
所以你自己在某个goroutine里写了

string s = go.file.readFile("/root")

其实go偷偷在里面执行了某操作系统的API asyncReadFIle。跑起来之后呢，这个方法就会说，我当前所在的goroutine跑够啦，把刚刚跑的那个异步操作的结果保存下下，换人：

// 实际上

handler h = someOS.asyncReadFile("/root") //很快返回一个handler

while (!h.finishedAsyncReadFile()): //很快返回Y/N

go.scheduler.保存现状()

go.scheduler.跑够了\_换人() // 相当于return，不过下次会从这里的下一句开始执行

string s = h.getResultFromAsyncRead()

然后scheduler就换下一个goroutine跑了。等下次再跑回刚才那个goroutine的时候，他就看看，说那个asyncReadFile到底执行完没有啊，如果没有，就再换个人吧。如果执行完了，那就把结果拿出来，该干嘛干嘛。所以你看似写了个同步的操作，已经被go替换成异步操作了。

还有另外一种情况是，某个goroutine执行了某个不能异步调用的会blocking的系统调用，这个时候goroutine就没法玩那种异步调用的把戏了。他会把你挪到一个真正的线程里让你在那个县城里等着，他接茬去跑别的goroutine。比如A这么定义

def A:

print("do something")

go.os.InvokeSomeReallyHeavyAndBlockingSystemCall()

print("do something 2")

go会帮你转成

def 真实的A:

print("do something")

Thread t = new Thread( () => {

SomeReallyHeavyAndBlockingSystemCall();

})

t.start()

while !t.finished():

go.scheduler.保存现状

go.scheduler.跑够了\_换人

print("finished")

所以真实的A还是不会blocking，还是可以跟别的小伙伴(goroutine)愉快地玩耍（轮流往复的被执行），但他其实已经占了一个真是的系统线程了。

当然会有一种情况就是A完全没有调用任何可能的“异步并发”的操作，也没有调用任何的同步的系统调用，而是一个劲的用CPU做运算（比如用个死循环调用a++）。在早期的go里，这个A就把整个程序block住了。后面新版本的go好像会有一些处理办法，比如如果你A里面call了任意一个别的函数的话，就有一定几率被踢下去换人。好像也可以自己主动说我要换人的，可以去查查新的go的spec