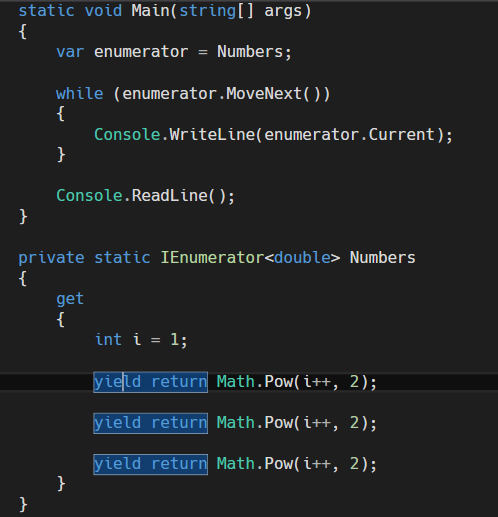
### 如何理解C#新的异步编程模型——Async和Await

这几天在和异步编程死磕，源于前阵子看了《CLR via C#》里Jeffrey Richter对C#异步等待编程模型的精彩讲解，让我终于对async和await这对兄弟有了比较好的理解。本想将书里的东西整理出来分享，但是当要坐下来写的时候却发现无非就是给几个Demo，然后.Net Reflector反编译下，看看编译器到底做了什么，我承认这是理解所有C#语法糖最好的方式，但总是这个套路未免显得有些枯燥。另一方面是因为这东西本身理解起来并不难，看一下书里作者的讲解很容易就能明白实现原理（墙裂建议看一下这本书第4版的相关内容，因为它比所有之前我在网上看到所有文章讲得都要好！讲得都要好！讲得都要好！）。所以我今天干嘛来了？经过这两天的冥思苦想，我发现可以从另一个来角度来理解这个编程模型。我希望通过我在下班无聊的时间里码下这些碎碎念的文字，能够对你理解异步编程有一丢丢的帮助。自然，如果没有什么帮助，也是正常的，因为我不是大神，不是天才，只是个普普通通的码农，嗯。

进入正题。

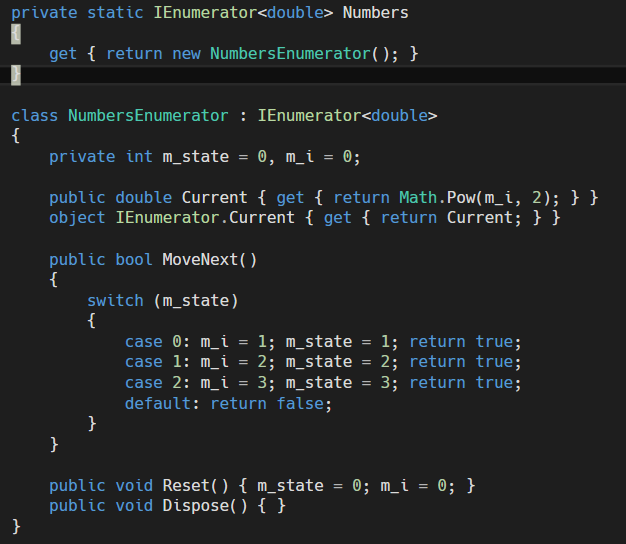
万事都有必要先给出一段代码（前方高能，有大量代码要贴）：



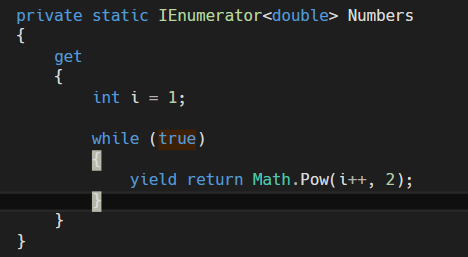
主函数调用Numbers，得到的是一个实现了IEnumerator<double>接口的对象（枚举器），这个接口我们再熟悉不过了，它的作用是支持枚举。我们拿到这个枚举器后只需要不断地调用它的MoveNext和Current向这个对象请求一个数字，最后直到MoveNext返回false，告诉我们里面的数字已经拿光了。上面的代码输出结果是1，4，9。这里边其实有这么个问题：这3个数字是在我们向枚举器首次请求数字之前（或者说在调用完Numbers）准备好的还是我们每请求一次它产生一个给我们，答案是后者，你可以在每个yield之前输出进行验证。这里起关键作用的是上面的yield关键字。我们可以以这样一种方式来理解上面的整个过程：

1. 调用Numbers获取枚举器，这个时候Numbers里的代码并不会执行；
2. 首次调用MoveNext，枚举器执行第1个yield语句，它准备好第1个数字，也就是1，用于在下次调用Current的时候返回给客户端；此时调用者从第1个yield语句（中断）退出；
3. 再次调用MoveNext，调用者从上次退出的位置继续执行第2个yield语句，它准备第2个数字，也就是4，用于在下次调用Current的时候返回给客户端；此时调用者从第2个yield语句（中断）退出；
4. 第三次调用MoveNext，调用者从上次退出的位置继续执行第3个yield语句，它准备第3个数字，也就是9，用于在下次调用Current的时候返回给客户端；
5. 以后每次调用枚举器的MoveNext，都不会有任何结果，因为Numbers里的代码已经执行完了；

从上面的这一个过程可以看出，yield关键字的作用是分阶段执行一个函数，这似乎很难想象：一个函数可以被中断，然后下次接着从上一次退出的地方往下执行。当然函数不可能真的按照这种方式执行，所有的这一切都是C#编译器造成的假象，我们看到的只是一个抽象的结果。编译器编译上面的代码后大致是下面这样的：



主函数代码不变，仅仅是把原先在Numbers里的代码逻辑抽出来放到了一个叫做NumbersEnumerator的类里边，Numbers返回的其实就是编译器生成的这个NumbersEnumerator类对象。yield的作用似乎超出我们的想象，如下代码：

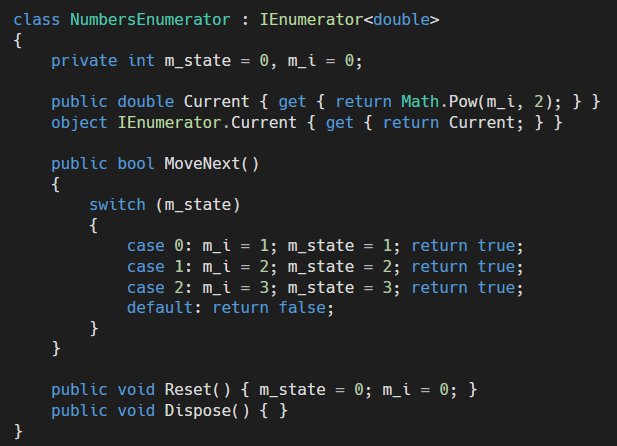


上面的代码并不会强制客户端代码陷入死循环（记住，yield具有中断代码执行的作用哦）。相反，只要你向枚举器请求，它便会提供一个数字给你，取之不尽。

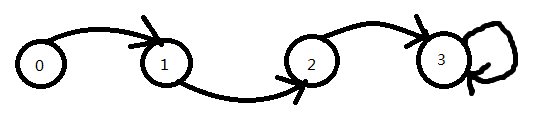
你可能会觉得，我为什么到现在还没有提到异步编程的东西，嗯，我也意识到了，我在这儿提起只是为了提醒我自己不要忘了正事儿，即使说完这句话我也并不打算开始讲这个：）。我下面要说的是一个叫做“状态机”的东西，很多时候它都被叫做“有限状态机”。它是一个什么东东？可以把它理解成是这样的一个机器：

1. 它有各种状态（通常是有限数量的状态）；
2. 在不同的状态下做不同的事；
3. 每做完一件事，就根据当前的结果跳转到下一个状态；

生活中的很多东西都是一个“有限状态机”：红绿灯，时钟，电梯。我们电脑的CPU也是个状态机，我甚至觉得我们写的每个程序都可看成是状态机。为什么要扯这个，因为我要再次贴出上面的NumbersEnumerator类：

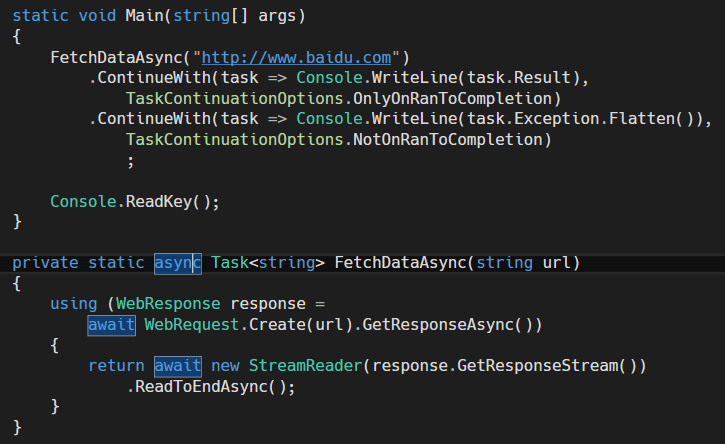


这个类就是一个状态机！上面的m\_state字段就表示状态机的状态，看到没有，MoveNext里的代码就是根据状态机的状态来执行不同的代码，执行完后切换到新的状态。我用图来表示这个状态机的执行过程：

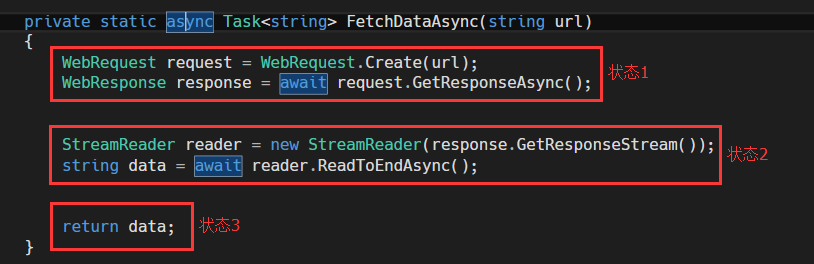


为了使这个状态机运行（其实我更喜欢称这个动作为“走步”），需要不断地调用MoveNext，分别从状态0切换到1，再切换到2，最后到3，之后每次切换都永远停留在状态3。我拿状态机来说事，只是为了得出一个结论：像这种模拟将函数分阶段执行的过程，可以通过类似的状态机来实现。

嗯，我还没有忘记正事儿。async和await的内心是崩溃的，主角最终演变成了配角。我觉得我还是应该贴一段async和await的代码：



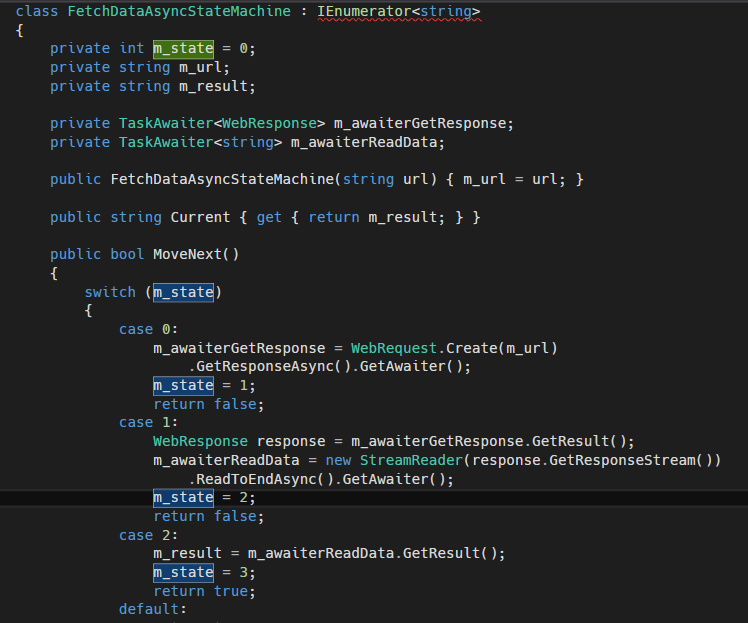
看到这对孪生兄弟了么？贴心的vs已经把它们一起高亮了。FetchDataAsync的作用是发一个http请求，得到请求返回的数据，最后返回给调用端。我在之前的一篇文章里提到过所有的IO操作都应该由异步的方式来完成，在这个例子里，主要有两个需要IO等待的过程，第1个是发送http请求等待响应的过程，第2个读取响应数据的过程（我把它看成是一个完整的过程，但实际上不是，因为数据肯定是一点一点地读，这中间有N次IO等待）。所以这两个方式都要以异步的方式进行。为了方便说明，我把FetchDataAsync里的代码调整了下：



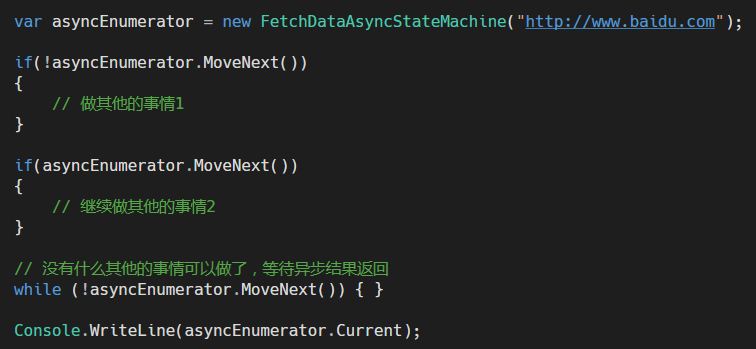
我根据IO等待把FetchDataAsync分成了3段，也就是说，这个方法要分3个阶段来执行：

1. 执行状态1里面的代码，当发送完请求之后调用者马上退出方法，去干别的事情去了；
2. 当请求返回响应的时候，接着上次退出的地方往下执行状态2里的代码，当发送完读取网络数据的请求后马上返回，又干别的事情去了；
3. 当读取数据结果返回后，又接着上次退出的地方执行状态3里的代码，这时候直接返回读取到的数据，方法到这里执行结束；

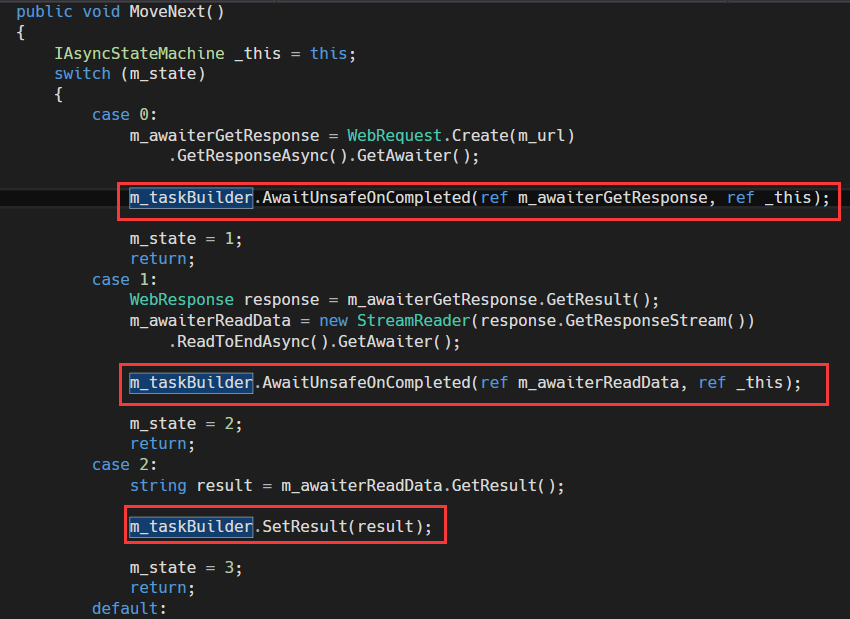
整个执行过程都是以异步的方式进行的。野屌们，你们体会到了什么？我们以后可以以同步的方式写异步代码了！！！不得不让我再次地想说一句话：微软真的是太贴心了！不过我软的优秀码农怎会拿了糖就直接往嘴里塞（async和await这颗语法糖确实够甜）？怎么也得先掰开来看看里面都裹了些啥是吧。我们只要看看最终编译器编译出来的代码，研读那么1，2个小时，自然会理解其中的原理，我也不打算细说这个。我之所以在前面花大篇幅讲yield关键字，是因为它和await有异曲同工之妙，他们解决的核心问题是实现函数的分阶段执行，我在前面得出结论：通过状态机可实现函数的分阶段执行。所以我现在要做的是把上面的代码转换成状态机，我自作聪明地尝试着转化得到了下面的代码：



MoveNext返回false代表状态机还没有结束，返回true代表状态机结束了，可以拿到结果，然后客户端的代码可以这么使用这个类：



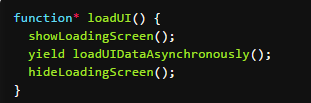
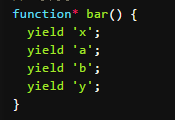
似乎还有点模样，但是上面的改造最大的问题在于，我得手动调用状态机查询状态，而且有很可能损失性能，因为当我调用MoveNext切换状态机状态时，如果此时状态机正在IO等待，那么我的主线程代码将会同时阻塞，等待IO结果返回。说白了就是这个状态机无法自动管理。所谓自动管理，就是当每个异步结果返回的时候，状态机能够自动接收到通知，切换到下一个状态。不过这些事情微软已经帮我们解决了，上面代码真正编译后大概就是这个样子：



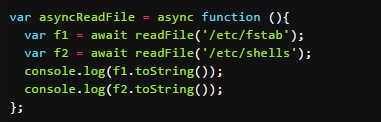
关键的地方在于AsyncTaskMethodBuilder类的AwaitUnsafeOnCompleted和SetResult方法，前者作用是当一个异步结果返回后自动通知状态机调用MoveNext方法切换到下一个状态，后者作用在状态机完成后设置异步任务的结果。

同样地，await也可以用在循环里面，就像yield一样，都可以通过状态机来实现。

关于C#的异步等待我想说的就这么多，我并没有讲这个东西具体的内部细节到底是怎么样的，因为这些通过看书和自己研读代码也能知道，而我要表达的意思是：很多技术上的实现它们的思想是相通的。我为什么这么说，是因为我发现上面讲的yield的竟然同时用在了ES6的特性：枚举和异步上，而后者正是我之前看ES6的时候看的云里雾里的，而这次看C#的async和await特性以及状态机的东西，对ES6的这个异步特性恍然大悟。我分别贴两段代码：



前者是yield用于枚举，后者是yield用于异步操作。目前yield在ES6里要实现状态机的自动管理需要写很多额外的代码，然后，关键是然后，据说ES7为了解决这个问题，这货也引入了async和await，来看看代码是怎样的吧：



有没有种感动得想哭的感觉，世界上最近距离是你我踏上不同的路，最后默契地在某个目的地相遇，然后相视一笑~