☐ 一：Async/await关键字

作用：Async和await关键字联合起来使用的时候其实就是await后面的就变成主线程的回调函数。相当于把后面的代码包装成委托了

个人理解：当用Async标记方法后，改方法就变成异步方法了。用Await来组和使用，Await后面就是具体需要异步的代码，await相当于开了一个新的线程，相当于Task.Run()方法。在同一个方法里面多个Await的话，会顺序执行Await的各个方法.所以要异步执行的话，多个Asunc方法配合await即可，而不需要多个Await这样玩。

当方法用async标识时，编译器主要做了什么呢?

（1）告诉编译器这个方法里面可能会用到await关键字来标识该方法是异步的，如此之后，编译器将会在状态机中编译此方法。接着该方法执行到await关键字时会处于挂起的状态直到该异步动作完成后才恢复继续执行方法后面的动作。

（2）告诉编译器解析出方法的结果到返回类型中，比如说Task或者Task<TResult>，也就是说将返回值存储到Task中，如果返回值为void那么此时应该会将可能出现的异常存储到上下文中

自己封装异步逻辑

了解了上面的东西之后，相信对 .NET 中的异步机制应该理解得差不多了，可以看出来这一套是名副其实的 coroutine，并且在实现上是 stackless 的。至于有的人说的什么状态机什么的，只是实现过程中利用的手段而已，并不是什么重要的东西。

那我们要怎么样使用 `Task` 来编写我们自己的异步代码呢?

事件驱动其实也可以算是一种异步模型，例如以下情景：

`A` 函数调用 `B` 函数，调用发起后就直接返回不管了（`BeginInvoke`），`B` 函数执行完成后触发事件执行 `C` 函数。

```
private event Action CompletedEvent;

void A()
{
    CompletedEvent += C;
    Console.WriteLine("begin");
    ((Action)B).BeginInvoke();
}

void B()
{
    Console.WriteLine("running");
    CompletedEvent?.Invoke();
}

void C()
{
    Console.WriteLine("end");
}
```

那么我们现在想要做一件事，就是把上面的事件驱动改造为利用 `async`/`await` 的异步编程模型，改造后的代码就是简单的：

	`` async Task A() { Console.WriteLine("begin"); await B(); Console.WriteLine("end"); } Task B() { Console.WriteLine("running"); return Task.CompletedTask; } ```

你可以看到，原本 `C` 函数的内容被放到了 `A` 调用 `B` 的下面，为什么呢?其实很简单，因为这里 `await B();` 这一行以后的内容，本身就可以理解为 `B` 函数的回调了，只不过在内部实现上，不是直接从 `B` 进行调用的回调，而是 `A` 先让出控制权，`B` 执行完成后，CLR 切换上下文，将 `A` 调度回来继续执行剩下的代码。