第三版：C# 19 道

请简述 async 函数的编译方式

async/await 是 C# 5.0 推出的异步代码编程模型，其本质是编译为状态机。只要函数前带上 async，就会将函数转换为状态机。

请简述 Task 状态机的实现和工作机制

CPS 全称是 Continuation Passing Style，在.NET 中，它会自动编译为：

1、 将所有引用的局部变量做成闭包，放到一个隐藏的状态机的类中；

2、 将所有的 await 展开成一个状态号，有几个 await 就有几个状态号；

3、 每次执行完一个状态，都重复回调状态机的 MoveNext 方法，同时指定下一个状态号；

4、 MoveNext 方法还需处理线程和异常等问题。

请简述 await 的作用和原理，并说明和 GetResult()有什么区别

从状态机的角度出发，await 的本质是调用 Task.GetAwaiter()的 UnsafeOnCompleted(Action)回调，并指定下一个状态号。

从多线程的角度出发，如果 await 的 Task 需要在新的线程上执行，该状态机的 MoveNext()方法会**立即返回**，此时，**主线程被释放出来了**，然后在

UnsafeOnCompleted 回调的 action 指定的线程上下文中继续 MoveNext() 和下一个状态的代码。

而相比之下，GetResult()就是在当前线程上立即等待 Task 的完成，在 Task 完成前，当前线程**不会释放**。

注意：Task 也可能不一定在新的线程上执行，此时用 GetResult()或者 await 就只有会不会创建状态机的区别了。

Task 和 Thread 有区别吗？如果有请简述区别

Task 和 Thread 都能创建用多线程的方式执行代码，但它们有较大的区别。

Task 较新，发布于.NET 4.5，能结合新的 async/await 代码模型写代码，它不止能创建新线程，还能使用线程池（默认）、单线程等方式编程，在 UI 编程领域，Task 还能自动返回 UI 线程上下文，还提供了许多便利 API 以管理多个 Task，用表格总结如下：

区别	Task	Thread
.NET 版本	4.5	1.1
async/await	支持	不支持
创建新线程	支持	支持
线程池/单线程	支持	不支持
返回主线程	支持	不支持
管理 API	支持	不支持

TL;DR 就是，用 Task 就对了。

简述 yield 的作用

yield 需配合 IEnumerable<T>一起使用，能在一个函数中支持多次（不是多个）返回，其本质和 async/await 一样，也是状态机。

如果不使用 yield，需实现 IEnumerable<T>，它只暴露了 GetEnumerator<T>，这样确保 yield 是可重入的，比较符合人的习惯。

注意，其它的语言，如 C++/Java/ES6 实现的 yield，都叫 generator（生成器），这相当于.NET 中的 IEnumerator<T>（而不是 IEnumerable<T>）。这种设计导致 yield 不可重入，**只要其迭代过一次，就无法重新迭代了**，需要注意。

利用 IEnumerable<T>实现斐波那契数列生成

```
IEnumerable<int> GenerateFibonacci(int n)
{
    int current = 1, next = 1;

    for (int i = 0; i < n; ++i)
    {
        yield return current;
        next = current + (current = next);
    }
}
```

简述 stackless coroutine 和 stackful coroutine 的区别，并指出 C# 的 coroutine 是哪一种

1、 stackless 和 stackful 对应的是协程中栈的内存，stackless 表示栈内存位置不固定，而 stackful 则需要分配一个固定的栈内存。

2、 在继续执行（Continuation/MoveNext()）时，stackless 需要编译器生成代码，如闭包，来自定义继续执行逻辑；而 stackful 则直接从原栈的位置继续执行。

3、 性能方面，stackful 的中断返回需要依赖控制 CPU 的跳转位置来实现，属于骚操作，会略微影响 CPU 的分支预测，从而影响性能（但影响不算大），这方面 stackless 无影响。

4、 内存方面，stackful 需要分配一个固定大小的栈内存(如 4kb)，而 stackless 只需创建带一个状态号变量的状态机，stackful 占用的内存更大。

5、 骚操作方面，stackful 可以轻松实现完全一致的递归/异常处理等，没有任何影响，但 stackless 需要编译器作者高超的技艺才能实现（如 C#的作者），注意最初的 C# 5.0 在 try-catch 块中是不能写 await 的。

6、 和已有组件结合/框架依赖方面，stackless 需要定义一个状态机类型，如 Task<T>/IEnumerable<T>/IAsyncEnumerable<T>等，而 stackful 不需要，因此这方面 stackless 较麻烦。

7、 Go 属于 stackful，因此每个 goroutine 需要分配一个固定大小的内存。

8、 C#属于 stackless，它会创建一个闭包和状态机，需要编译器生成代码来指定继续执行逻辑。

总结如下：

功能	stackless	stackful
内存位置	不固定	固定
继续执行	编译器定义	CPU 跳转
性能/速度	**快**	快，但影响分支预测
内存占用	**低**	需要固定大小的栈内存
编译器难度	难	**适中**
组件依赖	不方便	**方便**
嵌套	不支持	**支持**
举例	C#/js	Go/C++ Boost

请简述 SelectMany 的作用

相当于 js 中数组的 flatMap，意思是将序列中的**每一条数据**，转换为 **0 到多条**数据。

SelectMany 可以实现过滤/.Where，方法如下：

```
public static IEnumerable<T> MyWhere<T>(this IEnumerable<T> seq,
Func<T, bool> predicate)
{
    return seq.SelectMany(x => predicate(x) ?
        new[] { x } :
        Enumerable.Empty<T>());
}
```

SelectMany 是 LINQ 中 from 关键字的组成部分，这一点将在第 10 题作演示。

请实现一个函数 Compose 用于将多个函数复合

```
public static Func<T1, T3> Compose<T1, T2, T3>(this Func<T1, T2> f1,
Func<T2, T3> f2)
{
    return x => f2(f1(x));
}
```

然后使用方式：

```
Func<int, double> log2 = x => Math.Log2(x);
Func<double, string> toString = x => x.ToString();

var log2ToString = log2.Compose(toString);

Console.WriteLine(log2ToString(16)); // 4
```

实现 Maybe<T> monad，并利用 LINQ 实现对 Nothing（空值）和 Just（有值）的求和

本题比较难懂，经过和大佬确认，本质是要实现如下效果：

```
void Main()
{
    Maybe<int> a = Maybe.Just(5);
    Maybe<int> b = Maybe.Nothing<int>();
```

```
    Maybe<int> c = Maybe.Just(10);

    (from a0 in a from b0 in b select a0 + b0).Dump(); // Nothing
    (from a0 in a from c0 in c select a0 + c0).Dump(); // Just 15
}
```

按照我猴子进化来的大脑的理解，应该很自然地能写出如下代码：

```
public class Maybe<T> : IEnumerable<T>
{
    public bool HasValue { get; set; }
    public T Value { get; set;}

    IEnumerable<T> ToValue()
    {
        if (HasValue) yield return Value;
    }

    public IEnumerator<T> GetEnumerator()
    {
        return ToValue().GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return ToValue().GetEnumerator();
    }
}

public class Maybe
{
```

```
    public static Maybe<T> Just<T>(T value)
    {
        return new Maybe<T> { Value = value, HasValue = true};
    }

    public static Maybe<T> Nothing<T>()
    {
        return new Maybe<T>();
    }
}
```

这种很自然，通过继承 IEnumerable<T>来实现 LINQ to Objects 的基本功能，但却是错误答案。

正确答案：

```
public struct Maybe<T>
{
    public readonly bool HasValue;
    public readonly T Value;

    public Maybe(bool hasValue, T value)
    {
        HasValue = hasValue;
        Value = value;
    }

    public Maybe<B> SelectMany<TCollection, B>(Func<T,
Maybe<TCollection>> collectionSelector, Func<T, TCollection, B> f)
    {
        if (!HasValue) return Maybe.Nothing<B>();
```

```
            Maybe<TCollection> collection = collectionSelector(Value);
            if (!collection.HasValue) return Maybe.Nothing<B>();

            return Maybe.Just(f(Value, collection.Value));
        }

        public override string ToString() => HasValue ? $"Just {Value}" :
"Nothing";
}

public class Maybe
{
    public static Maybe<T> Just<T>(T value)
    {
        return new Maybe<T>(true, value);
    }

    public static Maybe<T> Nothing<T>()
    {
        return new Maybe<T>();
    }
}
```

注意：

首先这是一个函数式编程的应用场景，它应该使用 struct——值类型。

其次，不是所有的 LINQ 都要走 IEnumerable<T>，可以用手撸的 LINQ 表达式——SelectMany 来表示。（关于这一点，其实特别重要，我稍后有空会深入聊聊这一点。）

简述 LINQ 的 lazy computation 机制

1、 Lazy computation 是指延迟计算，它可能体现在**解析阶段**的**表达式树**和**求值阶段**的**状态机**两方面。

2、 首先是**解析阶段**的**表达式树**， C# 编译器在编译时，它会将这些语句以表达式树的形式保存起来，在求值时， C# 编译器会将所有的 表达式树 翻译成求值方法（如在数据库中执行 SQL 语句）。

3、 其次是**求值阶段**的**状态机**， LINQ to Objects 可以使用像 IEnumemrable<T> 接口，它本身不一定保存数据，**只有在求值时**，它返回一个迭代器—— IEnumerator<T> ，它才会根据 MoveNext() / Value 来求值。

4、 这两种机制可以确保 LINQ 是可以延迟计算的。

利用 SelectMany 实现两个数组中元素做笛卡尔集，然后一一相加

```
// 11\、利用 `SelectMany` 实现两个数组中元素的两两相加
int[] a1 = { 1, 2, 3, 4, 5 };
int[] a2 = { 5, 4, 3, 2, 1 };
a1
    .SelectMany(v => a2, (v1, v2) => $"{v1}+{v2}={v1 + v2}")
    .Dump();
```

解析与说明：大多数人可能只了解 SelectMany 做一转多的场景（两参数重载，类似于 flatMap ），但它还提供了这个三参数的重载，可以允许你做多对多——笛卡尔集。因此这些代码实际上可以用如下 LINQ 表示：

```
from v1 in a1
from v2 in a2
select $"{v1}+{v2}={v1 + v2}"
```

执行效果完全一样。

请为三元函数实现柯里化

解析：柯里化是指将 f(x, y) 转换为 f(x)(y) 的过程，三元和二元同理：

```
Func<int, int, int, int> op3 = (a, b, c) => (a - b) * c;
Func<int, Func<int, Func<int, int>>> op11 = a => b => c => (a - b) * c;
op3(4, 2, 3).Dump(); // 6
op11(4)(2)(3).Dump(); // 6
```

通过实现一个泛型方法，实现**通用的**三元函数柯里化：

```
Func<T1, Func<T2, Func<T3, TR>>> Currylize3<T1, T2, T3,
TR>(Func<T1, T2, T3, TR> op)
{
    return a => b => c => op(a, b, c);
}

// 测试代码：
var op12 = Currylize3(op3);
op12(4)(2)(3).Dump(); // (4-2)x3=6
```

现在了解为啥 F# 签名也能不用写参数了吧，因为参数确实太长了

请简述 ref struct 的作用

ref struct 是 C# 7.2 发布的新功能，主要是为了配合 Span<T> ，防止 Span<T> 被误用。

为什么会被误用呢？因为 Span<T> 表示一段连续、固定的内存，可供托管代码和非托管代码访问（不需要额外的 fixed ）这些内存可以从 stackalloc 中来，也能从 fixed 中获取托管的位置，也能通过 Marshal.AllocHGlobal() 等方式直接分配。这些内存**应该**是固定的、不能被托管堆移动。但之前的代码并不能很好地确保这一点，因此添加了 ref struct 来确保。

基于**不被托管堆管理**这一点，我们可以总结出以下结论：

1、 不能对 ref struct 装箱（因为装箱就变成引用类型了）——包括不能转换为 object 、 dynamic

2、 禁止实现任何接口（因为接口是引用类型）

3、 禁止在 class 和 struct 中使用 ref struct 做成员或自动属性（因为禁止随意移动，因此不能放到托管堆中。而引用类型、 struct 成员和自动属性都可能是在托管内存中）

4、 禁止在迭代器（ yield ）中使用 ref struct （因为迭代器本质是状态机，状态机是一个引用类型）

5、 在 Lambda 或 本地函数 中使用（因为 Lambda / 本地函数 都是闭包，而闭包会生成一个引用类型的类）

以前常有一个疑问，我们常常说值类型在栈中，引用类型在堆中，那**放在引用类型中的值类型成员，内存在哪？（在堆中，但必须要拷到栈上使用）**

加入了 ref struct ，就再也没这个问题了。

请简述 ref return 的使用方法

这也是个类似的问题， C# 一直以来就有 值类型 ，我们常常类比 C++ 的类型系统（只有值类型），它天生有性能好处，但 C# 之前**很容易产生没必要的复制**——导致 C# 并没有很好地享受 值类型 这一优点。

因此 C# 7.0 引入了 ref return ，然后又在 C# 7.3 引入了 ref 参数可被赋值。

使用示例：

```
Span<int> values = stackalloc int[10086];

values[42] = 10010;
int v1 = SearchValue(values, 10010);
v1 = 10086;
Console.WriteLine(values[42]); // 10010

ref int v = ref SearchRefValue(values, 10010);
v = 10086;
Console.WriteLine(values[42]); // 10086;

ref int SearchRefValue(Span<int> span, int value)
{
    for (int i = 0; i < span.Length; ++i)
```

```
    {
        if (span[i] == value)
            return ref span[i];
    }
    return ref span[0];
}

int SearchValue(Span<int> span, int value)
{
    for (int i = 0; i < span.Length; ++i)
    {
        if (span[i] == value)
            return span[i];
    }
    return span[0];
}
```

注意事项：

1、 参数可以用 Span<T> 或者 ref T

2、 返回的时候使用 return ref val

3、 注意返回值需要加 ref

4、 在赋值时，等号两边的变量，都需要加 ref 关键字（ ref int v1 = ref v2 ）

其实这个 ref 就是 C/C++ 中的指针一样。

请利用 foreach 和 ref 为一个数组中的每个元素加 1

```
int[] arr = { 1, 2, 3, 4, 5};
Console.WriteLine(string.Join(",", arr)); // 1,2,3,4,5

foreach (ref int v in arr.AsSpan())
{
    v++;
}

Console.WriteLine(string.Join(",", arr)); // 2,3,4,5,6
```

注意 foreach 不能用 var ，也不能直接用 int ，需要 ref int ，注意 arr 要转换为 Span<T> 。

请简述 ref 、 out 和 in 在用作函数参数修饰符时的区别

1、 ref 参数可同时用于输入或输出（变量使用前必须初始化）；

2、 out 参数只用于输出（使用前无需初始化）；

3、 in 参数只用于输入，它按引用传递，它能确保在使用过程中不被修改（变量使用前必须初始化）；

可以用一个表格来比较它们的区别：

修饰符/区别	ref	out	in	无
是否复制	✘	✘	✘	✔
能修改	✔	✔	✘	✘
输入	✔	✘	✔	✔

输出	✔	✔	✘	✘
需初始化	✔	✘	✔	✔

其实 in 就相当于 C++中的 const T&，我多年前就希望 C#加入这个功能了。

请简述非 sealed 类的 IDisposable 实现方法

正常 IDisposable 实现只有一个方法即可：

```
void Dispose()
{
    // free managed resources...
    // free unmanaged resources...
}
```

但它的缺点是必须手动调用 Dispose()或使用 using 方法，如果忘记调用了，系统的垃圾回收器不会清理，这样就会存在资源浪费，如果调用多次，可能会存在问题，因此需要 Dispose 模式。

Dispose 模式需要关心 C#的终结器函数（有人称为析构函数，但我不推荐叫这个名字，因为它并不和 constructor 构造函数对应），其最终版应该如下所示：

```
class BaseClass : IDisposable
{
    private bool disposed = false;

    ~BaseClass()
    {
        Dispose(disposing: false);
```

```
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposed) return;

        if (disposing)
        {
            // free managed resources...
        }

        // free unmanaged resources...
        disposed = true;
    }

    public void Dispose()
    {
        Dispose(disposing: true);
        GC.SuppressFinalize(this);
    }
}
```

它有如下要注意的点：

1、 引入 disposed 变量用于判断是否已经回收过，如果回收过则不再回收；

2、 使用 protected virtual 来确保子类的正确回收，注意不是在 Dispose 方法上加；

3、 使用 disposing 来判断是.NET 的终结器回收还是手动调用 Dispose 回收，终结器回收不再需要关心释放托管内存；

4、 使用 GC.SuppressFinalize(this)来避免多次调用 Dispose；

至于本题为什么要关心非 sealed 类，因为 sealed 类不用关心继承，因此 protected virtual 可以不需要。

在子类继承于这类，且有更多不同的资源需要管理时，实现方法如下：

```
class DerivedClass : BaseClass
{
    private bool disposed = false;

    protected override void Dispose(bool disposing)
    {
        if (disposed) return;

        if (disposing)
        {
            // free managed resources...
        }

        // free unmanaged resources...
        base.Dispose(disposing);
    }
}
```

注意：

1、 继承类也需要定义一个新的、不同的 disposed 值，不能和老的 disposed 共用；

2、 其它判断、释放顺序和基类完全一样；

3、 在继承类释放完后，调用 base.Dispose(disposing)来释放父类。

delegate 和 event 本质是什么？请简述他们的实现机制

delegate 和 event 本质都是多播委托（MultipleDelegate），它用数组的形式包装了多个 Delegate，Delegate 类和 C 中函数指针有点像，但它们都会保留类型、都保留 this，因此都是类型安全的。

delegate（委托）在定义时，会自动创建一个继承于 MultipleDelegate 的类型，其构造函数为 ctor(object o, IntPtr f)，第一个参数是 this 值，第二个参数是函数指针，也就是说在委托赋值时，自动创建了一个 MultipleDelegate 的子类。

委托在调用()时，编译器会翻译为.Invoke()。

注意:delegate 本身创建的类，也是继承于 MultipleDelegate 而非 Delegate，因此它也能和事件一样，可以指定多个响应：

```
string text = "Hello World";

Action v = () => Console.WriteLine(text);
v += () => Console.WriteLine(text.Length);
v();
// Hello World
// 11
```

注意，+=运算符会被编译器会翻译为 Delegate.Combine()，同样地-=运算符会翻译为 Delegate.Remove()。

事件是一种由编译器生成的特殊多播委托，其编译器生成的默认（可自定义）代码，与委托生成的 MultipleDelegate 相比，事件确保了+=和-=运算符的线程安全，还确保了 null 的时候可以被赋值（而已）。