## **解析“60k”大佬的19道C#面试题（上）**

## **先略看题目：**

1. 请简述async函数的编译方式
2. 请简述Task状态机的实现和工作机制
3. 请简述await的作用和原理，并说明和GetResult()有什么区别
4. Task和Thread有区别吗？如果有请简述区别
5. 简述yield的作用
6. 利用IEnumerable<T>实现斐波那契数列生成
7. 简述stackless coroutine和stackful coroutine的区别，并指出C#的coroutine是哪一种
8. 请简述SelectMany的作用
9. 请实现一个函数Compose用于将多个函数复合
10. 实现Maybe<T>monad，并利用LINQ实现对Nothing（空值）和Just（有值）的求和
11. 简述LINQ的lazy computation机制
12. 利用SelectMany实现两个数组中元素的两两相加
13. 请为三元函数实现柯里化
14. 请简述ref struct的作用
15. 请简述ref return的使用方法
16. 请利用foreach和ref为一个数组中的每个元素加1
17. 请简述ref、out和in在用作函数参数修饰符时的区别
18. 请简述非sealed类的IDisposable实现方法
19. delegate和event本质是什么？请简述他们的实现机制

没错，这是一位来自【广州.NET技术俱乐部】微信群的偏Programming Languages（编程语言开发科学）的大佬，本文我将**斗胆**回答一下这些题目

由于这些题目（对我来说）比较**难**，因此我这次只**斗胆**回答前10道题，发作上篇，另外一半的题目再等我慢慢查阅资料，另行回答

## **解析：**

## **1. 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。

## **2. 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。

## **3. 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。

## **4. 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：



TL;DR就是，用Task就对了。

## **5. 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。

## **6. 利用IEnumerable<T>实现斐波那契数列生成**

IEnumerable<**int**> GenerateFibonacci(**int** n){

**if** (n >= 1) **yield** **return** 1;

**int** a = 1, b = 0;

**for** (**int** i = 2; i <= n; ++i)

{

**int** t = b;

b = a;

a += t;

**yield** **return** a;

}}

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

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。

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。

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。

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。

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。

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。

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

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：



## **8. 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。

## **9. 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*

## **10. 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。）

## **总结**

这些技术平时可能比较冷门，全部能回答正确也并不意味着会有多有用，可能很难有机会用上。

但如果是在开发像ASP.NET Core那样的超高性能网络服务器、中间件，或者Unity 3D那样的高性能游戏引擎、或者做一些高性能实时ETL之类的，就能依靠这些知识，做出比肩甚至超过C/C++的性能，同时还能享受C#/.NET便利性的产品。

群里有人戏称面试时出这些题的公司，要么是心太大，要么至少得开60k，因此本文取名为60k大佬。

敬请期待我的下篇