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 ContinuationPassingStyle，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 .NET4.5，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。

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

1. IEnumerable<int> GenerateFibonacci(int n)
2. {
3. if (n >= 1) yield return 1;
4. int a = 1, b = 0;
5. for (int i = 2; i <= n; ++i)
6. {
7. int t = b;
8. b = a;
9. a += t;
10. yield return a;
11. }
12. }

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：

| **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 |

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：

1. public static IEnumerable<T> MyWhere<T>(this IEnumerable<T> seq, Func<T, bool> predicate)
2. {
3. return seq.SelectMany(x => predicate(x) ?
4. new[] { x } :
5. Enumerable.Empty<T>());
6. }

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

1. public static Func<T1, T3> Compose<T1, T2, T3>(this Func<T1, T2> f1, Func<T2, T3> f2)
2. {
3. return x => f2(f1(x));
4. }

然后使用方式：

1. Func<int, double> log2 = x => Math.Log2(x);
2. Func<double, string> toString = x => x.ToString();
3. var log2ToString = log2.Compose(toString);
4. 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：

1. void Main()
2. {
3. Maybe<int> a = Maybe.Just(5);
4. Maybe<int> b = Maybe.Nothing<int>();
5. Maybe<int> c = Maybe.Just(10);
6. (from a0 in a from b0 in b select a0 + b0).Dump(); // Nothing
7. (from a0 in a from c0 in c select a0 + c0).Dump(); // Just 15
8. }

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：

1. public class Maybe<T> : IEnumerable<T>
2. {
3. public bool HasValue { get; set; }
4. public T Value { get; set;}
5. IEnumerable<T> ToValue()
6. {
7. if (HasValue) yield return Value;
8. }
9. public IEnumerator<T> GetEnumerator()
10. {
11. return ToValue().GetEnumerator();
12. }
13. IEnumerator IEnumerable.GetEnumerator()
14. {
15. return ToValue().GetEnumerator();
16. }
17. }
18. public class Maybe
19. {
20. public static Maybe<T> Just<T>(T value)
21. {
22. return new Maybe<T> { Value = value, HasValue = true};
23. }
24. public static Maybe<T> Nothing<T>()
25. {
26. return new Maybe<T>();
27. }
28. }

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

正确答案：

1. public struct Maybe<T>
2. {
3. public readonly bool HasValue;
4. public readonly T Value;
5. public Maybe(bool hasValue, T value)
6. {
7. HasValue = hasValue;
8. Value = value;
9. }
10. public Maybe<B> SelectMany<TCollection, B>(Func<T, Maybe<TCollection>> collectionSelector, Func<T, TCollection, B> f)
11. {
12. if (!HasValue) return Maybe.Nothing<B>();
13. Maybe<TCollection> collection = collectionSelector(Value);
14. if (!collection.HasValue) return Maybe.Nothing<B>();
15. return Maybe.Just(f(Value, collection.Value));
16. }
17. public override string ToString() => HasValue ? $"Just {Value}" : "Nothing";
18. }
19. public class Maybe
20. {
21. public static Maybe<T> Just<T>(T value)
22. {
23. return new Maybe<T>(true, value);
24. }
25. public static Maybe<T> Nothing<T>()
26. {
27. return new Maybe<T>();
28. }
29. }

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.NETCore那样的超高性能网络服务器、中间件，或者 Unity3D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。