ET 框架学习笔记（四）- - 框架总结【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】

deepwaterooo

June 14, 2023

Contents

1 Root 客户端根场景管理以及必要的组件：【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!! 爱表哥，爱生活!!!】

- 昨天晚上，今天上午把这个根场景下的必要的组件，疑难点大致又看了一遍过了一遍。以后再有什么不懂，或是理解一点儿新的，再添加。【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】
- 把这个客户端的根场景相关的管理组件整理一下。在系统启动起来的时候，公用组件以及客户端组件的时候，分别有添加一些必要的组件。
- 这里，当把根场景 Root.Instance.Scene 下添加的组件整理好，就看见，几乎所有客户端必要需要的组件这里都添加了，那么刚才几分钟前，我想要自己添加一个的 SceneType.AllServer 里模仿参考项目想要添加的。这里需要想一下：两个都需要吗，还是 SceneType.AllServer 因为这个根场景下都加了，我自己画蛇添足的全服就可以不要了?

1.1 Root.cs

```
// 管理根部的 Scene: 这个根部，是全局视图的根节点
public class Root: Singleton<Root>, ISingletonAwake { // 单例类，自觉醒
    // 管理所有的 Entity:
    private readonly Dictionary<long, Entity> allEntities = new();
    public Scene Scene { get; private set; }
    public void Awake() {
        this.Scene = EntitySceneFactory.CreateScene(0, SceneType.Process, "Process");
    }
    public override void Dispose() {
        this.Scene.Dispose();
    }
    public void Add(Entity entity) {
        this.allEntities.Add(entity.InstanceId, entity);
    }

    public void Remove(long instanceId) {
        this.allEntities.Remove(instanceId);
    }
    public Entity Get(long instanceId) {
        Entity component = null;
```

```
        this.allEntities.TryGetValue(instanceId, out component);
        return component;
    }

    public override string ToString() {
        StringBuilder sb = new();
        HashSet<Type> noParent = new HashSet<Type>();
        Dictionary<Type, int> typeCount = new Dictionary<Type, int>();
        HashSet<Type> noDomain = new HashSet<Type>();
        foreach (var kv in this.allEntities) {
            Type type = kv.Value.GetType();
            if (kv.Value.Parent == null) {
                noParent.Add(type);
            }
            if (kv.Value.Domain == null) {
                noDomain.Add(type);
            }
            if (typeCount.ContainsKey(type)) {
                typeCount[type]++;
            }
            else {
                typeCount[type] = 1;
            }
        }
        sb.AppendLine("not set parent type: ");
        foreach (Type type in noParent) {
            sb.AppendLine($"\t{type.Name}");
        }
        sb.AppendLine("not set domain type: ");
        foreach (Type type in noDomain) {
            sb.AppendLine($"\t{type.Name}");
        }
        IOrderedEnumerable<KeyValuePair<Type, int>> orderByDescending = typeCount.OrderByDescending(s => s.Value);
        sb.AppendLine("Entity Count: ");
        foreach (var kv in orderByDescending) {
            if (kv.Value == 1) {
                continue;
            }
            sb.AppendLine($"\t{kv.Key.Name}: {kv.Value}");
        }
        return sb.ToString();
    }
}
```

1.2 EntryEvent1_InitShare.cs: Root 根场景添加组件

- 这里是双端共享组件启动的时候，也就是说，Root.Instance.Scene 并不仅仅只是客户端场景，也是服务端场景。

```
// 公用的相关组件的初始化:
[Event(SceneType.Process)]
public class EntryEvent1_InitShare: AEvent<EventType.EntryEvent1> {

    protected override async ETTask Run(Scene scene, EventType.EntryEvent1 args) {
        Root.Instance.Scene.AddComponent<NetThreadComponent>();
        Root.Instance.Scene.AddComponent<OpcodeTypeComponent>();
        Root.Instance.Scene.AddComponent<MessageDispatcherComponent>();
        Root.Instance.Scene.AddComponent<NumericWatcherComponent>();
        Root.Instance.Scene.AddComponent<AIDispatcherComponent>();
        Root.Instance.Scene.AddComponent<ClientSceneManagerComponent>();
        await ETTask.CompletedTask;
    }
}
```

1.3 EntryEvent2_InitServer: 服务端启动的时候添加的组件

```
[Event(SceneType.Process)]
public class EntryEvent2_InitServer: AEvent<ET.EventType.EntryEvent2> {
    protected override async ETTask Run(Scene scene, ET.EventType.EntryEvent2 args) {
        // 发送普通 actor 消息
        Root.Instance.Scene.AddComponent<ActorMessageSenderComponent>();
```

```
        // 发送 location actor 消息
        Root.Instance.Scene.AddComponent<ActorLocationSenderComponent>();
        // 访问 location server 的组件
        Root.Instance.Scene.AddComponent<LocationProxyComponent>();
        Root.Instance.Scene.AddComponent<ActorMessageDispatcherComponent>();
        Root.Instance.Scene.AddComponent<ServerSceneManagerComponent>();
        Root.Instance.Scene.AddComponent<RobotCaseComponent>();
        Root.Instance.Scene.AddComponent<NavmeshComponent>();
        //  【添加组件】: 这里，还可以再添加一些游戏必要【根组件】，如果可以在服务器启动的时候添加的话。会影响服务器启动性能

        StartProcessConfig processConfig = StartProcessConfigCategory.Instance.Get(Options.Instance.Process);
        switch (Options.Instance.AppType) {
        case AppType.Server: {
            Root.Instance.Scene.AddComponent<NetInnerComponent, IPEndPoint>(processConfig.InnerIPPort);
            var processScenes = StartSceneConfigCategory.Instance.GetByProcess(Options.Instance.Process);
            foreach (StartSceneConfig startConfig in processScenes) {
                await SceneFactory.CreateServerScene(ServerSceneManagerComponent.Instance, startConfig.Id, startConfig.Inst
            }
            break;
        }
        case AppType.Watcher: {
            StartMachineConfig startMachineConfig = WatcherHelper.GetThisMachineConfig();
            WatcherComponent watcherComponent = Root.Instance.Scene.AddComponent<WatcherComponent>();
            watcherComponent.Start(Options.Instance.CreateScenes);
            Root.Instance.Scene.AddComponent<NetInnerComponent, IPEndPoint>(NetworkHelper.ToIPEndPoint($"{startMachineConfi
            break;
        }
        case AppType.GameTool:
            break;
        }
        if (Options.Instance.Console == 1) {
            Root.Instance.Scene.AddComponent<ConsoleComponent>();
        }
    }
}
```

2 ETTask 和 ETVoid: 第三方库的 ETTask, 参考 ET-EUI 框架

- 特异包装：主要是实际了异步调用的流式写法。它方法定义的内部，是封装有协程异步状态机的? IAsyncStateMachine. 当要运行协程的下一步，也是调用和运行。NET 库里的 IAsyncStateMachine.moveNext()
- .NET 还提供了 AsyncMethodBuilder 的 type trait 来让你自己实现这个状态机和你自己的 Task 类型，因此你可以最大程度发挥想象来编写你想控制的一切。ETTask|ETVoid 就是使用底层的这些方法来封装的结果。async/await 是一个纯编译器特性。
- 这个框架里 ET7 里，就有相关模块 **【具体说是，两个实体类，实际定义了两种不同返回值 ETTask-ETVoid 的协程编译生成方法】**，能够实现对这个包装的自动编译成协程的编译逻辑方法定义。理解上，感觉像是 ET7 框架里，为了这个流式写法，定义了必要的标签系，和相关的协程生成方法，来帮助这个第三方库实现异步调用的流式写法。
- 上面的，写得把自己都写昏了。就是 ET7 框架是如何实现异步调用的流式写法的呢? 它把异步调用封装成协程。面对 ET7 框架里广泛用到的 ETTask|ETVoid 两类稍带个性化异步任务，如同 ETTask 和 ETVoid 是框架自己的封装一样，这个框架，也使用.NET 里的 IAsyncStateMachine 等底层接口 API 等，自定义了异步协程任务的生成方法。
- 这类方法里，都封装有一个 ETTask, 因为自定义封装在这些自定义类里，就对可能会用到的操作提供了必要的 API, 比如设置异常，拿取任务等等。
- 上面的自定义方法生成器:有三类,分别是 AsyncETVoidMethodBuilder, AsyncETTaskMethodBuilder 和 AsyncETTaskCompletedMethodBuilder
- 感觉因为这两大返回类型，我没有能看懂看透，所以上面一个部分的消息处理，两个函数 Handle() 和 Run() 的返回类型，以及参数被我改得乱七八糟，是不应该的。

- 磨刀不误砍柴工，我应该投入时间把这个第三方库一样的包装理解透彻，然后再去弄懂上面一个部分，再去改那些编译错误。

- **【ET-EUI】里：**原本类的定义什么的，也是一样的，那就是主要去看，他是怎么使用 ETVoid，为什么它使用 ETVoid 不会报错，而我在 ET7 里用就会。

- **【多线程同步】**关于多线程同步的理解：来自于网络：

  - ETTASK 的由于没有开新线程，也没有使用线程池 Task，所以肯定是在主线程运行的，那么游戏开始的 SynchronizationContext.SetSynchronizationContext(OneThreadSynchronizatio 这句代码有啥用呢?
  - 个人理解为，**在 ET 中虽然主逻辑是单线程的，但是与 IO 设备，比如从 socket 读取数据，或者从 TCP,KCP 获取网络数据得时候，是多线程的获取数据的，所以当数据到达时，为了保证是单线程，所以在获取数据的地方，以回调得方式，将回调方法扔到 OneThreadSynchronizationContext 中执行（async 设置了同步上下文是线程安全的**，说的应该也是这个 OneThreadSynchronizationContext() 什么的相关的）
  - 白话多线程同步原理如下：下面的也是 ET 框架中网络异步线程同步中干过的同步执行逻辑。那个类大概是 NetService.cs. 就是分主线程，异步线程，有队列，Update() 里同步的。
    - ET 是单线程的，所以不会管理线程
    - 跨线程都是把委托投递到一个队列，主线程不停从队列中取出委托执行
    - 你看看 asynctool 的代码，本质上就是把委托投递到主线程
    - 每帧取完队列中的所有委托，执行完
  - 这个细节，是自己第一个游戏里使用 ET-EUI 作为服务端，非 ET 框架的客户端与服务端连接时，自己曾经遇到过的。非 ET 框架的客户端，是使用了一个其它的 UnityPlayer 里一个 API 相关的第三方来同步异步线程的结果到主线程。所以这个细节还是印象深刻。

- 首先要能把控得住多线程，才能谈性能。其次，et 是服务端多进程，同样能利用多核。et 是逻辑单线程，并不意味着只能单线程，你能把控得住，照样可以多线程，一般人是不行的。（这些，看不懂，感觉更像是避重就轻吹牛皮一样。。。）

- **【ETTask-await 后面的执行线程:】**

  - async await 如果用的 Task, await 后面的部分是不确定在哪个线程执行的, 猫大以前 4.0 的做法就是把上下文抛到主线程, 让主线程执行.
  - 如果用的是 ETTask, await 后面的部分是一定在主线程执行的. 就完全相当于写了个回调方法了
  - Task 实际上也是回调, 不过这个回调方法的执行原本可能不在主线程罢了

- ETVoid 是代替 async void，意思是新开一个协程。ettask 跟 task 一样。当然 task 不去 await 也相当于新开协程，但是编辑器会冒出提示，提示你 await。所以新开协程最好用 ETVoid。4.0 用 async void。使用场景，自己写写就明白啦. 协程就是回调.

- **无 GC ETTask**, 其实是利用对象池，注意，必须小心使用，不懂不要乱用。无 GC 的原理同自己写第一个游戏，使用资源池是一样的，就是说，当一个 ETTask 使用完毕，不再使用的时候，不是要 GC 来回收，而是程序的逻辑自己管理，回收到对象池管理器，对于应用程序来说，就是不释放，自己管理它的再使用。不释放就不会引起 GC 回收，所以叫无 GC.

  - 请不要随便使用 ETTask 的对象池，除非你完全搞懂了 ETTask!!!

  - 假如开启了池,await 之后不能再操作 ETTask，否则可能操作到再次从池中分配出来的 ETTask，产生灾难性的后果。(自己的理解，await 之后，再操作 ETTask, 那么操作的极有可能是【当 boolean fromPool = true】从对象池新取出的一个异步任务，不是预期行为，当然就会引起一片混乱。。。可是，框架里仍然有狠多对异步任务 SetResult() 的地方，尤其是各种服的消息处理器处理逻辑里。什么情境下可以安全地使用 SetResult(), 需要自己去搞明白)
  - SetResult 的时候请现将 tcs 置空，避免多次对同一个 ETTask SetResult. (这里，对一个异步任务，设置结果 SetResult(), 可能会设置多次吗?)
  - 大部分情况可以用 objectwait 代替 ettask，推荐使用，绝对不会出问题
- 这里因为弄不明白，他们建议的学习方法是:
  - ettask 还要啥教程?
  - 要搞懂 ettask 下载一个 jetbrain peek 工具，反编译下看下生成的代码就行了。
  - 参考 Timercomponent，看懂就全明白了
  - 看网上的文章看十年也不会明白，自己写一下 timercomponet 啥都懂了
  - 接下来，自己尝试理解这部分的方法应该是: 给 VS 2022 安装第三方插件库 ILSpy, 然后借用插件把编译码自己弄出来，插日志，作任何可以帮助自己理解的东西来理解这部分。**【先给 VS 安装一个插件 ILSpy, 这样更容易反编译代码进行查看, 另外要注意反编译 async 和 await 的时候, 要把 C# 代码版本改为 4.0 哦。】**前在，这是网上提示的反编译方法。这个，改天再接着看，先再事理理解一点儿别的。今天一定更新一下。明天出行，没时间看和更新。
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!!】

2.1 IAsyncStateMachine

- 异步方法中，每遇见一个 await 调用，都会生成一个异步状态类，这个异步状态类会实现这个接口

```
namespace System.Runtime.CompilerServices {
    public interface IAsyncStateMachine {
        void MoveNext();
        void SetStateMachine(IAsyncStateMachine stateMachine);
    }
}
```

2.2 enum AwaiterStatus: IAwaiter.cs 文件里. 理解为异步任务的现执行进展状态

- 现框架里，扩展 IAwaiter, 自定义的现框架 ETTask 所可能有的三种不同状态。

```
public enum AwaiterStatus: byte {
    // The operation has not yet completed.
    Pending = 0,// 这个用在判断语句里比较多，主要用它来判断: 异步任务是否已经完成
    // The operation completed successfully.
    Succeeded = 1,
    // The operation completed with an error.
    Faulted = 2,
}
```

2.3 ETTaskCompleted: 已经完成了的异步任务。比较特殊：可以简单进行写结果？等等的必要回收工作，就可以返回异步任务对象池回收再利用？

- 因为我把 AsyncMethodBuilder 理解成为：异步任务的协程编译器编译逻辑。
- 所以这个类就是定义，异步任务协程中的一个特殊状态：异步任务结束了，但是还没能写、返回结果时的 IAsyncStateMachine|IAwater 状态
- 因为它是协程异步状态机中的一个相对特殊状态，本质上是异步状态机中的一个状态，也提供了必要的 API, 比较写结果。

```
[AsyncMethodBuilder(typeof (AsyncETTaskCompletedMethodBuilder))]
public struct ETTaskCompleted: ICriticalNotifyCompletion {
    [DebuggerHidden]
// 能不能理解为，已经结束了的异步任务 ETTaskCompleted, 也是协程异步状态机中的一个状态，是 IAwaker 的实体类实现状态，返回这个
    public ETTaskCompleted GetAwaiter() {
        return this;
    }
    [DebuggerHidden]
    public bool IsCompleted => true;
    [DebuggerHidden]
    public void GetResult() {
    }
// 就是说：下面的两个回调函数，可以帮  助把异步任务的执行结果给返回回去
    [DebuggerHidden]
    public void OnCompleted(Action continuation) {
    }
    [DebuggerHidden]
    public void UnsafeOnCompleted(Action continuation) {
    }
}
```

2.4 struct ETVoid: ICriticalNotifyCompletion. 这里涉及协程的分阶段的执行相关逻辑的生成方法自动化相关的标签

```
[AsyncMethodBuilder(typeof (AsyncETVoidMethodBuilder))]// 【异步方法生成标签】: 是。NET CompilerService 里的属性标签。自动生成协
internal struct ETVoid: ICriticalNotifyCompletion {
    [DebuggerHidden]
        public void Coroutine() { }
    [DebuggerHidden]
        public bool IsCompleted => true;
    [DebuggerHidden]
        public void OnCompleted(Action continuation) { }
    [DebuggerHidden]
        public void UnsafeOnCompleted(Action continuation) { }
}
```

2.5 ETTask: ICriticalNotifyCompletion:

- 这个类的定义比较大，分普通类，和泛型类。我的笔记需要记在同一个地方。今天早上这个类又记错地方记到 ET-EUI 上去了

```
[AsyncMethodBuilder(typeof (ETAsyncTaskMethodBuilder))]
public class ETTask: ICriticalNotifyCompletion {
    public static Action<Exception> ExceptionHandler;// 异常回调
    public static ETTaskCompleted CompletedTask {    // 异步任务结束后的封装
        get {
            return new ETTaskCompleted();
        }
    }
    private static readonly Queue<ETTask> queue = new Queue<ETTask>();// 异步任务对象池
    // 请不要随便使用 ETTask 的对象池，除非你完全搞懂了 ETTask!!!
    // 假如开启了池,await 之后不能再操作 ETTask, 否则可能操作到再次从池中分配出来的 ETTask, 产生灾难性的后果
    // SetResult 的时候请现将 tcs 置空，避免多次对同一个 ETTask SetResult
    public static ETTask Create(bool fromPool = false) {
        if (!fromPool)
            return new ETTask();
        if (queue.Count == 0)
```

```
            return new ETTask() {fromPool = true};
        return queue.Dequeue();
    }
    private void Recycle() { // 涉及 ETTask 无 GC 的逻辑实现:
        if (!this.fromPool)  // 因为不返回对象池，所以会 GC
            return; // 原则：只有从池里取出来的，才返回池
        this.state = AwaiterStatus.Pending;//  【没明白:】回收时还设置为 Pending, 什么时候写的当前结果？应该是在回收前
        this.callback = null;
        if (queue.Count > 1000)  // 因为对象池中，异步任务数目已达 1000, 不再回收，也会产生 GC
            return;
        queue.Enqueue(this); // 真正无 GC, 因为回收到对象池，队列里去了
    }
    private bool fromPool;
    private AwaiterStatus state;
    private object callback; // Action or ExceptionDispatchInfo
    private ETTask() {  }
    //  【不明白下面两个方法】: 不知道这两个方法，绕来绕去，在干什么？
    [DebuggerHidden] // 下面，旦凡带 async 关键字的方法，都是异步方法，编译器编译 async 方法时，会自动生成方法所对应的 Coro
    private async ETVoid InnerCoroutine() { // 这里，怎么就可以用 ETVoid 了呢？ private 内部异步方法
        await this; //  【不明白】: 每次看见 await 后面接一个单词，就不知道是在等什么？等待这个 ETTask 异步任务类初始化完成？
    }
    [DebuggerHidden]
    public void Coroutine() { // 公用无返回，非异步方法。它调用了类内部私有的异步方法 InnerCoroutine()
        InnerCoroutine().Coroutine();// 这里因为理解不透，总感觉同上面的方法，返回 this, 又调用了自己本方法 Coroutine() -
    }
    [DebuggerHidden]
    public ETTask GetAwaiter() {
        return this;
    }
    public bool IsCompleted {
        [DebuggerHidden]
        get {
            return this.state != AwaiterStatus.Pending; // 只要不是 Pending 状态，就是异步任务执行结束
        }
    }
    [DebuggerHidden]
    public void UnsafeOnCompleted(Action action) {
        if (this.state != AwaiterStatus.Pending) { // 如果当前异步任务执行结束，就触发非空回调
            action?.Invoke();
            return;
        }
        this.callback = action; // 任务还没有结束，就纪录回调备用
    }
    [DebuggerHidden]
    public void OnCompleted(Action action) {
        this.UnsafeOnCompleted(action);
    }
    [DebuggerHidden]
    public void GetResult() {
        switch (this.state) {
            case AwaiterStatus.Succeeded:
                this.Recycle();
                break;
            case AwaiterStatus.Faulted:
                ExceptionDispatchInfo c = this.callback as ExceptionDispatchInfo;
                this.callback = null;
                this.Recycle();
                c?.Throw();
                break;
            default:
                throw new NotSupportedException("ETTask does not allow call GetResult directly when task not complete
        }
    }
    [DebuggerHidden]
    public void SetResult() {
        if (this.state != AwaiterStatus.Pending) {
            throw new InvalidOperationException("TaskT_TransitionToFinal_AlreadyCompleted");
        }
        this.state = AwaiterStatus.Succeeded;
        Action c = this.callback as Action;
        this.callback = null;
        c?.Invoke();
    }
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    [DebuggerHidden]
```

```
    public void SetException(Exception e) {
        if (this.state != AwaiterStatus.Pending) {
            throw new InvalidOperationException("TaskT_TransitionToFinal_AlreadyCompleted");
        }
        this.state = AwaiterStatus.Faulted;
        Action c = this.callback as Action;
        this.callback = ExceptionDispatchInfo.Capture(e);
        c?.Invoke();
    }
}
[AsyncMethodBuilder(typeof (ETAsyncTaskMethodBuilder<>))]
public class ETTask<T>: ICriticalNotifyCompletion {
    private static readonly Queue<ETTask<T>> queue = new Queue<ETTask<T>>();
    // 请不要随便使用 ETTask 的对象池，除非你完全搞懂了 ETTask!!!
    // 假如开启了池,await 之后不能再操作 ETTask，否则可能操作到再次从池中分配出来的 ETTask，产生灾难性的后果
    // SetResult 的时候请现将 tcs 置空，避免多次对同一个 ETTask SetResult
    public static ETTask<T> Create(bool fromPool = false) {
        if (!fromPool)
            return new ETTask<T>();
        if (queue.Count == 0)
            return new ETTask<T>() { fromPool = true };
        return queue.Dequeue();
    }
    private void Recycle() {
        if (!this.fromPool)
            return;
        this.callback = null;
        this.value = default;
        this.state = AwaiterStatus.Pending;
        // 太多了
        if (queue.Count > 1000)
            return;
        queue.Enqueue(this);
    }
    private bool fromPool;
    private AwaiterStatus state;
    private T value;
    private object callback; // Action or ExceptionDispatchInfo
    private ETTask() {
    }
    [DebuggerHidden]
    private async ETVoid InnerCoroutine() {
        await this;
    }
    [DebuggerHidden]
    public void Coroutine() {
        InnerCoroutine().Coroutine();
    }
    [DebuggerHidden]
    public ETTask<T> GetAwaiter() {
        return this;
    }
    [DebuggerHidden]
    public T GetResult() {
        switch (this.state) {
        case AwaiterStatus.Succeeded:
            T v = this.value;
            this.Recycle();
            return v;
        case AwaiterStatus.Faulted:
            ExceptionDispatchInfo c = this.callback as ExceptionDispatchInfo;
            this.callback = null;
            this.Recycle();
            c?.Throw();
            return default;
        default:
            throw new NotSupportedException("ETask does not allow call GetResult directly when task not completed. Pl
        }
    }
    public bool IsCompleted {
        [DebuggerHidden]
        get {
            return state != AwaiterStatus.Pending;
        }
    }
```

```
        [DebuggerHidden]
        public void UnsafeOnCompleted(Action action) {
            if (this.state != AwaiterStatus.Pending) {
                action?.Invoke();
                return;
            }
            this.callback = action;
        }
        [DebuggerHidden]
        public void OnCompleted(Action action) {
            this.UnsafeOnCompleted(action);
        }
        [DebuggerHidden]
        public void SetResult(T result) {
            if (this.state != AwaiterStatus.Pending) {
                throw new InvalidOperationException("TaskT_TransitionToFinal_AlreadyCompleted");
            }
            this.state = AwaiterStatus.Succeeded;
            this.value = result;
            Action c = this.callback as Action;
            this.callback = null;
            c?.Invoke();
        }
        [DebuggerHidden]
        public void SetException(Exception e) {
            if (this.state != AwaiterStatus.Pending) {
                throw new InvalidOperationException("TaskT_TransitionToFinal_AlreadyCompleted");
            }
            this.state = AwaiterStatus.Faulted;
            Action c = this.callback as Action;
            this.callback = ExceptionDispatchInfo.Capture(e);
            c?.Invoke();
        }
    }
```

2.6 ETCancellationToken: 管理所有的取消？回调：因为可能不止一个取消回调，所以 HashSet 管理

```
public class ETCancellationToken {// 管理所有的【取消】回调：因为可能不止一个取消回调，所以 HashSet 管理
    private HashSet<Action> actions = new HashSet<Action>();
    public void Add(Action callback) {
        // 如果 action 是 null，绝对不能添加，要抛异常，说明有协程泄漏
        // 【不喜欢这个注释，看不懂，感觉它吓唬人的。。】
        this.actions.Add(callback);
    }
    public void Remove(Action callback) {
        this.actions?.Remove(callback);
    }
    public bool IsDispose() {
        return this.actions == null;
    }
    public void Cancel() {
        if (this.actions == null) {
            return;
        }
        this.Invoke();
    }
    private void Invoke() {
        HashSet<Action> runActions = this.actions;
        this.actions = null;
        try {
            foreach (Action action in runActions) {
                action.Invoke();
            }
        }
        catch (Exception e) {
            ETTask.ExceptionHandler.Invoke(e);
        }
    }
}
```

2.7 ETTaskHelper: 有个类中类 CoroutineBlocker 看不懂

```
public static class ETTaskHelper {
    public static bool IsCancel(this ETCancellationToken self) {
        if (self == null)
            return false;
        return self.IsDispose();
    }
    // 【看不懂】: 感觉理解这个类有难度
    private class CoroutineBlocker {
        private int count; // 不知道, 这个变量记的是什么？
        private ETTask tcs;
        public CoroutineBlocker(int count) {
            this.count = count;
        }
        public async ETTask RunSubCoroutineAsync(ETTask task) {
            try {
                await task;
            }
            finally {
                --this.count;
                if (this.count <= 0 && this.tcs != null) { // 写结果？
                    ETTask t = this.tcs;
                    this.tcs = null;
                    t.SetResult();
                }
            }
        }
        public async ETTask WaitAsync() {
            if (this.count <= 0)
                return;
            this.tcs = ETTask.Create(true);
            await tcs;
        }
    }
    public static async ETTask WaitAny(List<ETTask> tasks) {
        if (tasks.Count == 0)
            return;
        CoroutineBlocker coroutineBlocker = new CoroutineBlocker(1);
        foreach (ETTask task in tasks) {
            coroutineBlocker.RunSubCoroutineAsync(task).Coroutine();
        }
        await coroutineBlocker.WaitAsync();
    }
    public static async ETTask WaitAny(ETTask[] tasks) {
        if (tasks.Length == 0)
            return;
        CoroutineBlocker coroutineBlocker = new CoroutineBlocker(1);
        foreach (ETTask task in tasks) {
            coroutineBlocker.RunSubCoroutineAsync(task).Coroutine();
        }
        await coroutineBlocker.WaitAsync();
    }
    public static async ETTask WaitAll(ETTask[] tasks) {
        if (tasks.Length == 0)
            return;
        CoroutineBlocker coroutineBlocker = new CoroutineBlocker(tasks.Length);
        foreach (ETTask task in tasks) {
            coroutineBlocker.RunSubCoroutineAsync(task).Coroutine();
        }
        await coroutineBlocker.WaitAsync();
    }
    public static async ETTask WaitAll(List<ETTask> tasks) {
        if (tasks.Count == 0)
            return;
        CoroutineBlocker coroutineBlocker = new CoroutineBlocker(tasks.Count);
        foreach (ETTask task in tasks) {
            coroutineBlocker.RunSubCoroutineAsync(task).Coroutine();
        }
        await coroutineBlocker.WaitAsync();
    }
}
```

2.8 ETAsyncTaskMethodBuilder: 同样是换汤不换药的两个部分：普通类与泛型类

```
public struct ETAsyncTaskMethodBuilder {
    private ETTask tcs;
    // 1. Static Create method.
    [DebuggerHidden]
    public static ETAsyncTaskMethodBuilder Create() {
        ETAsyncTaskMethodBuilder builder = new ETAsyncTaskMethodBuilder() { tcs = ETTask.Create(true) };
        return builder;
    }
    // 2. TaskLike Task property.
    [DebuggerHidden]
    public ETTask Task => this.tcs;
    // 3. SetException
    [DebuggerHidden]
    public void SetException(Exception exception) {
        this.tcs.SetException(exception);
    }
    // 4. SetResult
    [DebuggerHidden]
    public void SetResult() {
        this.tcs.SetResult();
    }
    // 5. AwaitOnCompleted
    [DebuggerHidden]
    public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where TAwai
        awaiter.OnCompleted(stateMachine.MoveNext);
    }
    // 6. AwaitUnsafeOnCompleted
    [DebuggerHidden]
    [SecuritySafeCritical]
    public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where
        awaiter.OnCompleted(stateMachine.MoveNext);
    }
    // 7. Start
    [DebuggerHidden]
    public void Start<TStateMachine>(ref TStateMachine stateMachine) where TStateMachine : IAsyncStateMachine {
        stateMachine.MoveNext();
    }
    // 8. SetStateMachine
    [DebuggerHidden]
    public void SetStateMachine(IAsyncStateMachine stateMachine) {
    }
}
public struct ETAsyncTaskMethodBuilder<T> {
    private ETTask<T> tcs;
    // 1. Static Create method.
    [DebuggerHidden]
    public static ETAsyncTaskMethodBuilder<T> Create() {
        ETAsyncTaskMethodBuilder<T> builder = new ETAsyncTaskMethodBuilder<T>() { tcs = ETTask<T>.Create(true) };
        return builder;
    }
    // 2. TaskLike Task property.
    [DebuggerHidden]
    public ETTask<T> Task => this.tcs;
    // 3. SetException
    [DebuggerHidden]
    public void SetException(Exception exception) {
        this.tcs.SetException(exception);
    }
    // 4. SetResult
    [DebuggerHidden]
    public void SetResult(T ret) {
        this.tcs.SetResult(ret);
    }
    // 5. AwaitOnCompleted
    [DebuggerHidden]
    public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where TAwai
        awaiter.OnCompleted(stateMachine.MoveNext);
    }
    // 6. AwaitUnsafeOnCompleted
    [DebuggerHidden]
    [SecuritySafeCritical]
```

```
    public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where
        awaiter.OnCompleted(stateMachine.MoveNext);
    }
    // 7. Start
    [DebuggerHidden]
    public void Start<TStateMachine>(ref TStateMachine stateMachine) where TStateMachine : IAsyncStateMachine {
        stateMachine.MoveNext();
    }
    // 8. SetStateMachine
    [DebuggerHidden]
    public void SetStateMachine(IAsyncStateMachine stateMachine) {
    }
}
```

2.9 AsyncETTaskCompletedMethodBuilder:

```
public struct AsyncETTaskCompletedMethodBuilder {
    // 1. Static Create method.
    [DebuggerHidden]
    public static AsyncETTaskCompletedMethodBuilder Create() {
        AsyncETTaskCompletedMethodBuilder builder = new AsyncETTaskCompletedMethodBuilder();
        return builder;
    }
    // 2. TaskLike Task property(void)
    public ETTaskCompleted Task => default;
    // 3. SetException
    [DebuggerHidden]
    public void SetException(Exception e) {
        ETTask.ExceptionHandler.Invoke(e);
    }
    // 4. SetResult
    [DebuggerHidden]
    public void SetResult() { // do nothing
    }
    // 5. AwaitOnCompleted
    [DebuggerHidden]
    public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where TAwai
        awaiter.OnCompleted(stateMachine.MoveNext);
    }
    // 6. AwaitUnsafeOnCompleted
    [DebuggerHidden]
    [SecuritySafeCritical]
    public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where
        awaiter.UnsafeOnCompleted(stateMachine.MoveNext);
    }
    // 7. Start
    [DebuggerHidden]
    public void Start<TStateMachine>(ref TStateMachine stateMachine) where TStateMachine : IAsyncStateMachine {
        stateMachine.MoveNext();
    }
    // 8. SetStateMachine
    [DebuggerHidden]
    public void SetStateMachine(IAsyncStateMachine stateMachine) {
    }
}
```

2.10 AsyncETVoidMethodBuilder: 定义的是 async ETVoid 的编译方法?

```
// 异步 ETVoid 内部生成方法:
internal struct AsyncETVoidMethodBuilder {
    // 1. Static Create method.
    [DebuggerHidden]
    public static AsyncETVoidMethodBuilder Create() {
        AsyncETVoidMethodBuilder builder = new AsyncETVoidMethodBuilder();
        return builder;
    }
    // 2. TaskLike Task property(void)
    [DebuggerHidden]
    public ETVoid Task => default;
    // 3. SetException
    [DebuggerHidden]
    public void SetException(Exception e) {
```

```
            ETTask.ExceptionHandler.Invoke(e);
        }
        // 4. SetResult
        [DebuggerHidden]
        public void SetResult() {
            // do nothing: 因为它实际的返回值是 void
        }
        // 5. AwaitOnCompleted
        [DebuggerHidden]
        public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where TAwai
            awaiter.OnCompleted(stateMachine.MoveNext);
        }
        // 6. AwaitUnsafeOnCompleted
        [DebuggerHidden]
        [SecuritySafeCritical]
        public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where
            awaiter.UnsafeOnCompleted(stateMachine.MoveNext);
        }
        // 7. Start
        [DebuggerHidden]
        public void Start<TStateMachine>(ref TStateMachine stateMachine) where TStateMachine : IAsyncStateMachine {
            stateMachine.MoveNext();
        }
        // 8. SetStateMachine
        [DebuggerHidden]
        public void SetStateMachine(IAsyncStateMachine stateMachine) {
        }
    }
```

2.11 ICriticalNotifyCompletion:

```
namespace System.Runtime.CompilerServices {
    // 接口类：提供了一个，任务完成后的回调接口
        public interface ICriticalNotifyCompletion : INotifyCompletion {
            [SecurityCritical]
            void UnsafeOnCompleted(Action continuation);
        }
    }
```

2.12 AsyncMethodBuilderAttribute:.NET 系统的标签

- 自己先前没能理解，为什么标记了【AsyncMethodBuilder(typeof(className))】就能标记某个类的协程生成方法
- 是因为这个系统标签，它申明了 AttributeUsage 属性，申明了适用类型，可以是 (AttributeTargets.Class | AttributeTargets.Struct) 等等
- 所以，当 ETTask 异步库自定义了 ETTask, ETVoid, 和 ETTaskCompleted 三个类，就可以使用上面的系统标签，来标注申明：这个类是以上三个中特定指定此类的协程编译生成方法。

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct | AttributeTargets.Enum | AttributeTargets.Interface
public sealed class AsyncMethodBuilderAttribute : Attribute {
    public AsyncMethodBuilderAttribute(Type builderType);

    public Type BuilderType { get; }
}
```

3 C# 异步基础原理、状态机原理、逻辑整理

3.1 ETVoid C# Net-async|await 编程更底层一点儿的原理

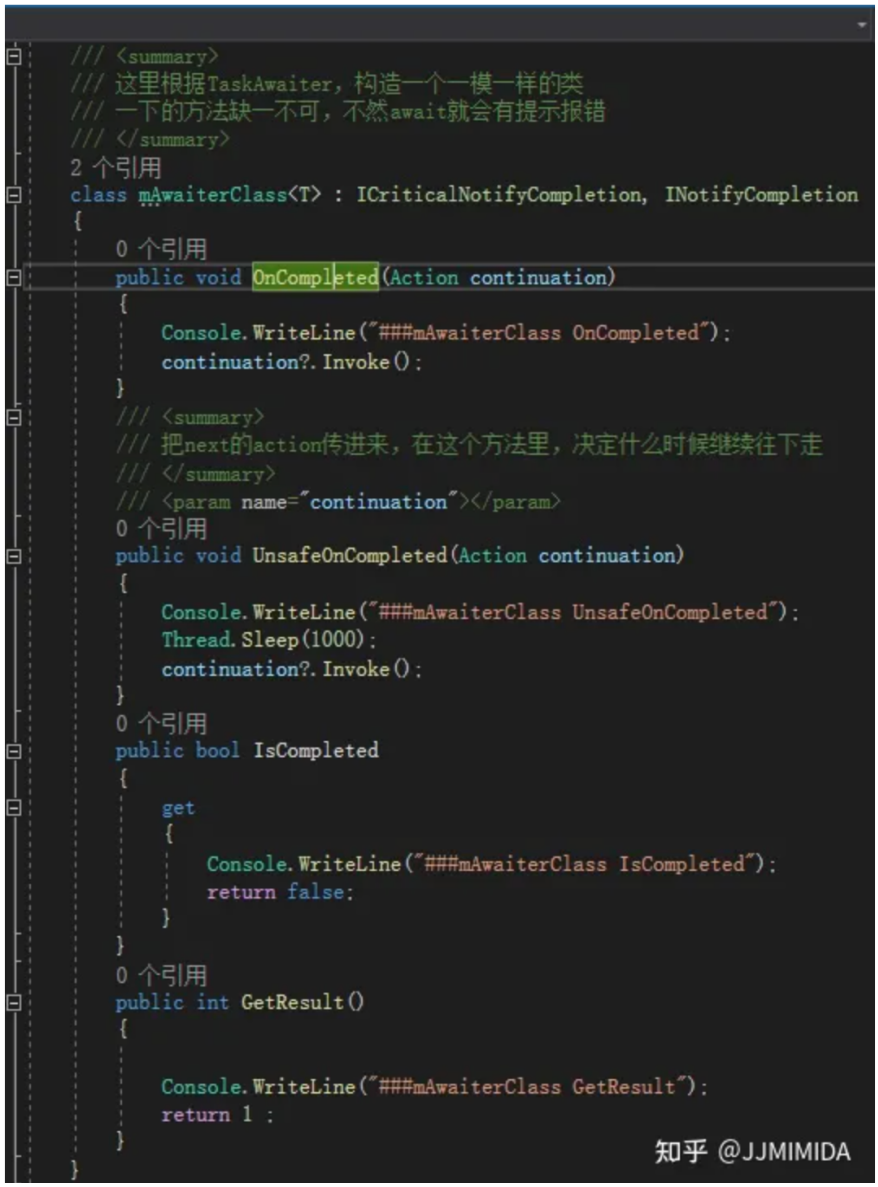
- 就是不懂底层的原理是什么，方法定义是什么，返回的是什么，在有 await 等关键字的时候，返回的内容等是如何变换的，以及它背后的那个异步状态机，就是想不明白。

- 现在参考网上的一个例子，记一下异步任务 C# 幕后封装的那些执行步骤什么的，把 async await 之类的关键字，背后的逻辑理解明白。
- 就是，可能也可以在异步任务的这个模块，添加无数的日志，通过读日志来把这块儿弄明白。下面就截图网上的这个参考例子。

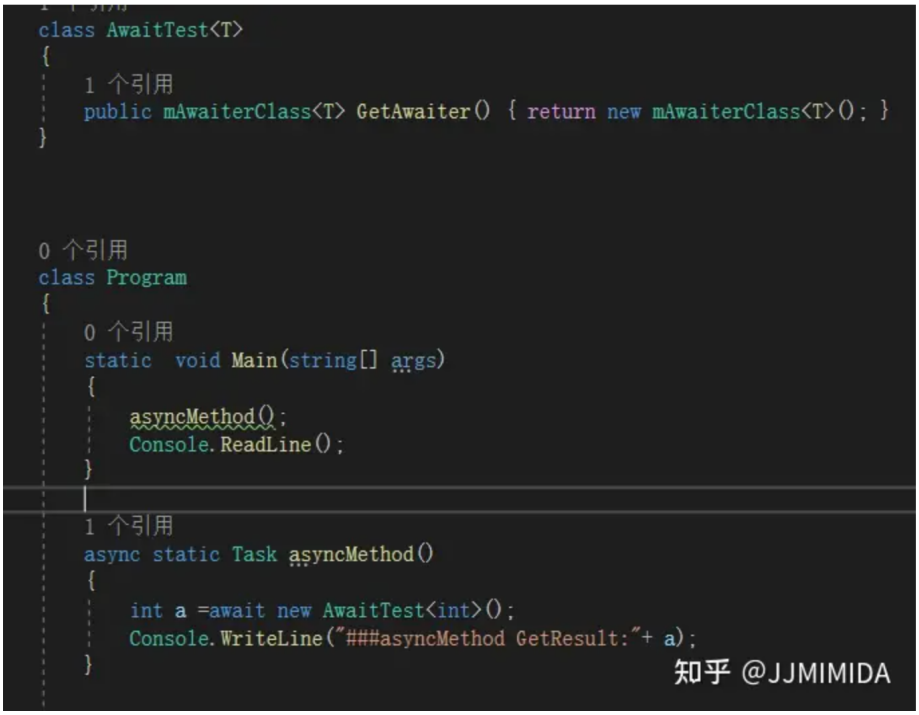
```
[AsyncMethodBuilder(typeof (AsyncETVoidMethodBuilder))]
internal struct ETVoid: ICriticalNotifyCompletion {
    [DebuggerHidden]
    public void Coroutine() { }
    [DebuggerHidden]
    public bool IsCompleted => true;
    [DebuggerHidden]
    public void OnCompleted(Action continuation) { }
    [DebuggerHidden]
    public void UnsafeOnCompleted(Action continuation) { }
}
```

- 上面是找了一个最短小的类 ETVoid, 网上例子自己构建一个类，这个类麻雀虽小五脏俱全的**几个【缺一不可】的方法**（所以知道 ETTask|ETVoid 自定义封装，这几个方法也是一定不能少的，只是多了 Coroutine() 方法不知道是怎么回事儿?）如上如下：

我们尝试去构造一个TaskAwaiter，看看await里面如何调用。



- 它的测试用例是这么写的：注意它传入的参数类型是 int. 后面的编译码里，和它的讲解里会用到提到。



- 看它编译出来的码（那堆编译出来的状态机的码），就是看不懂

```
// awaiter = new AwaitTest<int>().GetAwaiter();
IL_000e: nop
IL_000f: newobj instance void class AysncTest.AwaitTest`1<int32>::.ctor()
IL_0014: call instance class AysncTest.mAwaiterClass`1<!0> class AysncTest.AwaitTest`1<int32>::GetAwaiter()
IL_0019: stloc.1
// if (!awaiter.IsCompleted)
IL_001a: ldloc.1
IL_001b: callvirt instance bool class AysncTest.mAwaiterClass`1<int32>::get_IsCompleted()
IL_0020: brtrue.s IL_0062

// num = (<>1__state = 0);
IL_0022: ldarg.0
IL_0023: ldc.i4.0
IL_0024: dup
IL_0025: stloc.0
IL_0026: stfld int32 AysncTest.Program/'<asyncMethod>d__1'::'<>1__state'
// <>u__1 = awaiter;
IL_002b: ldarg.0
IL_002c: ldloc.1
IL_002d: stfld object AysncTest.Program/'<asyncMethod>d__1'::'<>u__1'
// <asyncMethod>d__1 stateMachine = this;
IL_0032: ldarg.0
IL_0033: stloc.2
// <>t__builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine);
IL_0034: ldarg.0
IL_0035: ldflda valuetype [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder Ays
IL_003a: ldloca.s 1
IL_003c: ldloca.s 2
IL_003e: call instance void [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder::
// return;
IL_0043: nop
IL_0044: leave.s IL_00c3

// awaiter = (mAwaiterClass<int>)<>u__1;
IL_0046: ldarg.0
IL_0047: ldfld object AysncTest.Program/'<asyncMethod>d__1'::'<>u__1'
IL_004c: castclass class AysncTest.mAwaiterClass`1<int32>
IL_0051: stloc.1
// <>u__1 = null;
IL_0052: ldarg.0
IL_0053: ldnull
IL_0054: stfld object AysncTest.Program/'<asyncMethod>d__1'::'<>u__1'
// num = (<>1__state = -1);
IL_0059: ldarg.0
IL_005a: ldc.i4.m1
IL_005b: dup
IL_005c: stloc.0
IL_005d: stfld int32 AysncTest.Program/'<asyncMethod>d__1'::'<>1__state'

// <>s__2 = awaiter.GetResult();
```


asyncMethod里面的真面目

```
summary>
F:\C#\AysncTest\bin\Debug\netcoreapp3.1\A
###mAwaiterClass IsCompleted
###mAwaiterClass UnsafeOnCompleted
###mAwaiterClass GetResult
###asyncMethod GetResult:1
```


- 结果分析：**【异步方法状态机，背后的执行顺序与逻辑：】**
  - 先检查 IsCompleted 标志位，如果已经完成，则调用 GetResult 作为 await 的返回值返回。
  - 如果未完成，经过 AsyncTaskMethodBuilder 的 AwaitUnsafeOnCompleted 方法之后，最后进入 UnsafeOnCompleted（nextAction），并且把完成后的下一步回调传进来。
  - 当我们获得 nextAction 之后，说明该调用由我们自己来控制，这里我在等待 1s 之后，执行 nextAction（），下一步 GetResult 返回。
- **【Async 关键字方法的编译原理：】**

对于上述问题2：带async签名的方法编译之后生成了什么？为什么带async签名的方法返回值一定是void、Task、Task<T>?

```
.class nested private auto ansi sealed beforefieldinit '<asyncMethod>d__1'
    extends [System.Runtime]System.Object
    implements [System.Runtime]System.Runtime.CompilerServices.IAsyncStateMachine
{
    .custom instance void [System.Runtime]System.Runtime.CompilerServices.CompilerGeneratedAttribute::.ctor() = (
        01 00 00 00
    )
    // Fields
    .field public int32 '<>1__state'
    .field public valuetype [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder '<>t__builder'
    .field private int32 '<a>5__1'
    .field private int32 '<>s__2'
    .field private object '<>u__1'

    // Methods
    .method public hidebysig specialname rtspecialname
        instance void .ctor () cil managed ...

    .method private final hidebysig newslot virtual
        instance void MoveNext () cil managed ...

    .method private final hidebysig newslot virtual
        instance void SetStateMachine (
            class [System.Runtime]System.Runtime.CompilerServices.IAsyncStateMachine stateMachine
        ) cil managed ...
} // end of class <asyncMethod>d__1
```

- 这个 async 关键字所标记的异步方法，主要两个点儿：
  - 编译器，把这个异步方法，编译成了一个类 class <asyncMethod>d__1;
  - 这个类 class, 它实现了 IAsyncStateMachine 接口，（**实现了这个接口，返回的是什么类型呢？**这个要想明白?）
  - 这个类 class, 的内部，有几个成员变量.field-sss.
  - 这个类 class, 的内部，有个特别重要的状态机执行函数 MoveNext() 来指挥指导，异步函数内不同节点如 await 节点等的执行逻辑。**【这个类 class, 它实现了 IAsyncStateMachine 接口】**，前面有列出 IAsyncStateMachine 接口定义的两个方法，所以实现实体类里也会有 SetStateMachine() 方法的实现。
  - 上面的逻辑，其实是就是扫描异步方法内，不同的 await 调用，每到一个这个关键字申明的异步调用，就是切换一个状态（背后有可能是线程的切换, 不一定每个分支都用不同的线程，但线程的切换可能是，必要的时候需要切换的?）分段执行。
- 网络上的分析者还给出了下机的截图：不是狠懂，这个截图是什么意思？因为不懂，要把编译码的方法名带上，方便以后再读和理解。

```
(
  AysncTest.Program/'<asyncMethod>d__1',
type [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder

bj instance void AysncTest.Program/'<asyncMethod>d__1'::.ctor()
c.0
c.0
valuetype [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder::Create()
d valuetype [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder AysncTest.Program/'<asyncMethod>d__1'::'<>t__builder'
c.0
i4.m1
d int32 AysncTest.Program/'<asyncMethod>d__1'::'<>1__state'
c.0
d valuetype [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder AysncTest.Program/'<asyncMethod>d__1'::'<>t__builder'
c.1
ca.s 1
ca.s 0
instance void [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder::Start<class AysncTest.Program/'<asyncMethod>d__1'>(!!0&)
c.0
da valuetype [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder AysncTest.Program/'<asyncMethod>d__1'::'<>t__builder'
instance class [System.Runtime]System.Threading.Tasks.Task [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder::get_Task()
d Program::asyncMethod
```

- 上面的异步方法，所生成的异步状态机类 class 里，有几个主要的方法：
  - 构造器方法 ctor()：
  - Create() 方法：
  - Start() 方法：

  - get_Task() 方法：

- 可是上面的几个方法是谁，哪个接口定义的呢？

- 网络上的分析者，对上面两个截图的分析如下：**【它讲解的这部分，我可能还是得自己编译一下，去具体看一下。】因为它的截图不完整，看不懂**下面还有个别人总结的状态机套路，感觉说得更彻底透彻。

  - 签名为 async Task asyncMethod() 的方法里，先创建一个继承自 IAsyncStateMachine 的 asyncMethod 类
  - 创建一个 AsyncTaskMethodBuilder，然后赋值给 Machine.（不知道，它这句，说的是哪里？第一个图的最后 SetStateMachine()?）
  - 初始化 Machine 的 state = -1.（两个截图里看不见，找不到）
  - 调用 AsyncTaskMethodBuilder.Start 方法，start 里面会进入 Machine 的 moveNext () 方法，详见问题 1。
  - AsyncTaskMethodBuilder.get_Task () 作为该方法的返回值返回。

- 多线程问题：Task 一定是多线程吗？

  - 不一定，在上述例子中，我们定义的 async static Task<int> aa()，里面就是在同一个线程执行的。只有调用 Task.Start 或者 Task.Run 里面自动启用多线程的时候，才是多线程。

- 看得另一个网页中的说法，因为感觉它也没有实现个什么公共定义约束的接口，理解得不够透彻。看下下面的：

- await 必须配合 Task/ValueTask 才能用吗? 当然不是。

  - 在 C# 中 **只要你的类中包含 GetAwaiter() 方法和 bool IsCompleted 属性，并且 GetAwaiter() 返回的东西包含一个 GetResult() 方法、一个 bool IsCompleted 属性和实现了 INotifyCompletion，那么这个类的对象就是可以 await 的**。这里说得还是不清楚，不透彻，换一个表达得更清晰的说法如下：

- 可以使用 await 的方法，返回值必须是 **awaitable 对象**，自定义 awaitable 对象比较麻烦，一个对象必须满足下列条件才行：

  - 必须有一个 **GetAwaiter()** 方法，扩展方法或者实例方法都可以
  - GetAwaiter() 方法返回值必须是 **awaiter 对象**。一个对象要成为 awaiter 对象必须满足下列条件:
    * 该对象 **实现接口 INotifyCompletion 或者 ICriticalNotifyCompletion**
    * 必须有 **IsCompleted 属性**
    * 必须有 **GetResult() 方法**，可以返回 void 或者其他返回值。

- 比如下面的自定义类：把几个类的本质理解得再深一点儿了吗？【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】

```
public class MyTask<T> {
    public MyAwaiter<T> GetAwaiter() {// 必须提供的方法
        return new MyAwaiter<T>();
    }
}
// 下面自定义的类 MyAwaiter<T= 亲爱的表哥 > 就是可以 await 的：
// 【任何时候，活宝妹就是一定要嫁给亲爱的表哥!!! 活宝妹还没能嫁给亲爱的表哥，活宝妹就是永远守候在亲爱的表哥的身边!!! 爱表哥，爱生活!!
public class MyAwaiter<T> : INotifyCompletion {// 必须实现的接口
    public bool IsCompleted { get; private set; }// 属性变量
    public T GetResult() {// 必须要有的方法
```

```
        throw new NotImplementedException();
    }
    public void OnCompleted(Action continuation) {
        throw new NotImplementedException();
    }
}
public class Program {
    static async Task Main(string[] args) {
        var obj = new MyTask<int>();
        await obj;
    }
}
```

- **【状态机套路】:**
- async 关键字标记方法是一个异步方法，编译器通过这个标记 **【async 关键字】**去改造这个方法体为创建状态机的方法。await 是关键字，是为了实现状态机中的一个状态，每当有一个 await，就会生成一个对应的状态。状态机就是根据这个状态，去一步步的调用异步委托，然后回调，包括状态机的解析。
- (1). 状态机的默认状态都是-1, 结束状态都是-2.
- (2). 每 await 一次就会产生一个 TaskAwaiter awaiter; 改变状态机的状态, 当有多个 await 的时候, 每个 await 都会改变状态机的状态，比如改为 0,1,2,3,4 等等，分别表示代码中 await xxx 这句话执行完成。
- (3). 状态机的执行套路:
  - A. 首先创建一个 d_num 的方法（这里说错了，应该是创建了一个类 class）, xxx 代表方法名,num 可能是 0,1,2,3 等，**实现 IAsyncStateMachine 接口**。
  - B. 在 MoveNext() 方法中, 源代码中每个 await xxxx 都会对应生成是一个 TaskAwaiter awaiter，然后 xxxx.GetAwaiter()
  - C. 判断状态机是否执行完 if (!awaiter.IsCompleted),
    - 没有执行完的话走 <>t__builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine); 代表释放当前线程
    - 执行完后走，<>s__1 = awaiter.GetResult(); 拿到返回值，继续走后面的代码。
- (此处写的比较抽象，看下面 3 结合代码编译再分析)
- 感觉今天读这个状态机:https://linuxcpp.0voice.com/?id=1380 终于有点儿开窍了!!【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!】

3.2 如果方法声明为 async，那么可以直接 return 具体的值，不再用创建 Task，由编译器创建 Task:

```
// 只要标记了 async 就会被编译成状态机
// 如果方法声明为 async，那么可以直接 return 具体的值，不再用创建 Task，由编译器创建 Task:
  public static async Task<int> F2Async() {
      return 2;
  }
```

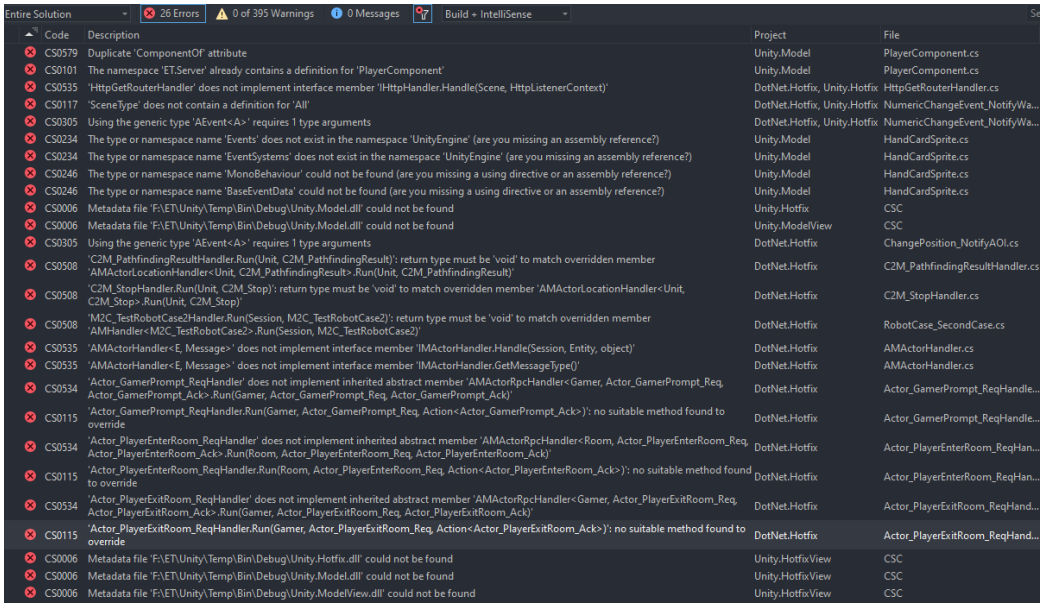
- F2Async：只加了 async，会生成状态机，但由于没有加 await 所以不会涉及到中间状态的变化，从-1 默认状态变为结束的-2 状态。
- F3Async：既有 async 也有 await (await 只有 1 个)，该方法是使用了 Task.Run，我们把它归为计算型的异步方法。
- 亲爱的表哥，活宝妹今天终于把这个看得稍微有点儿懂了，希望能够赶快从这个 ETTask 模块 move-forward. 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!! 活宝妹还没能嫁给亲爱的表哥，活宝妹就是永远守候在亲爱的表哥的身边!!! 爱表哥，爱生活!!!

4 现在的修改内容：【任何时候，活宝妹就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!】

- 【问题】：上次那个 ET-EUI 框架的时候，曾经出现过 opcode 不对应，也就是说，我现在生成的进程间消息，有可能还是会存在服务器码与客户端码不对应，这个完备的框架，这次应该不至于吧?
- 【UIType】部分类：这个类出现在了三四个不同的程序域，现在重构了，好像添加得不对。要再修改
- **【ET7 框架】**没有处理的逻辑是：**【ET7 框架里数据库的接入】**
- **【UILobbyComponent 可以测试】**：这个大厅组件，Unity 里预设简单，可以试运行一下，看是否完全消除这个 UI 组件的报错，这个屏的控件能否显示出来？还是错出得早，这个屏就出不来已经报错了?
  - 【客户端】的逻辑是处理好了，编译全过后可以测试
  - 【服务端】：处理用户请求匹配房间的逻辑，仍在处理：**C2G_StartMatch_ReqHandler**.
- **【TractorRoomComponent】**：因为是多组件嵌套，可以合并多组件为同一个组件；另早上看得一知半解的一个【ChildOf】标签，可以帮助组件套用吗？再找找理解消化一下
- 【房间组件】：几个现存的 working-on 的问题：
  - 多组件嵌套：手工合并为一个组件。彻底理解确认后，会合并
  - 【服务端】：处理用户请求匹配房间的逻辑. 这里的编译错误终于改完。到时就看运行时错误了。
    - 【数据库模块的整合】：网关服在转发请求匹配时，验证会话框有效后，验证用户身份时，需要去【用户数据库】拿用户数据。ET7 留了个 DBManagerComponent, 还没能整合出这个模块

    -【参考来源 **C2R_LoginHandler** 】：Realm 处理客户端的登录请求的服务端逻辑。这里看见，它随机分配一个网关服。也就是，我（原本本质上也是随机分配）一个匹配服给用。可以依照这里的例子来改写。
- **【服务端的编译错误】**基本上扫了一遍。【客户端】因为这些前期的工作，以及拖拉机项目重构设计还没有想透彻，暂停一下。
- 【接下来的内容】：**【重构拖拉机项目】**。把 ET7 框架里【参考项目】的设计看懂，并借助这个例子，把拖拉机项目设计好。
- 有时间，会试着尽早解决上面 ProtoBuf 里的几个小问题。但现在需要重构的设计思路，客户端的界面等才能够往下进行。
  - 【匹配服地址】网关服的处理逻辑里，验证完用户合格后，为代为转发消息到匹配服，但需要拿匹配服的地址。ET7 重构里，还没能改出这部分。服务器系统配置初始化时，可以链表管理各小构匹配服，再去拿相关匹配服的地址。ET7 框架里的路由器系统，自己还没有弄懂。
  - 这个地方有点儿脑塞，完全搜不到新框架里可以参考的例子，暂时写不到了。那可以去读一读更大的框架，去找别人用 ET7 的别人的例子里是怎么写的，再去参考一下别人的。【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】今天下午先去看 Tractor 游戏源码，设计重构思路

- 这些要找的也找不到。下午家里试着把 Component 组件再添加回去试试看 **【不能再添加 Component 组件。ET7 框架重构了，小单元也走热更新，在热更新层有天文小行星的生成系。可以参照 ET.pdf 里的服务端 PlayerSystem 来作例子】**？上午把项目设计的思路，源项目的破源码再读一读理一理，是希望游戏逻辑与游戏界面能够快速开发、项目进展往后移的。
  - User.cs 客户端的话，不知道要不要修改。晚点儿的时候留意一下。
  - Gamer.cs 客户端保留了 Dispose
- 还有 79 个小错误：protobuf 里还有小问题需要修改。先改了，一次把 Protobuf 里的小错误全部改完了。电脑没好好工作，前后文件不一致。。。。【活宝妹就是一定要嫁给亲爱的表哥!!!】爱表哥，爱生活!!!
  - **【IScene】**：不知道哪里崩出来的神龙见首不见尾的，两处，还不知道怎么修改这个编译错误。。。不知道源码是什么原因，弄得乱七八糟，参照原版本的改正一下就可以了，因为现项目里根本就不存在这个的接口类。
  - 有些文件加了两遍：当我能够从 VSC 里删除的时候，却无法从 VS 里删除掉索引。找到 VS 里可以看见文件的地主，感觉台式机真是慢，让我看见，已经是晚了 800 年了。。。现在找到该如何删除服务端的不必要的文件了，ET7 双端框架服务端的源码是引用的客户端的，所以所有必要的删除文件，都仍然是从客户端的源码里删除掉。
  - **【IMActorHandler】**: 在 ET7 的框架里, Handle() 方法的定义, 主要是 Actor 消息机制大概又重构得更为自动化一点儿, 就是说大致是说, 当有分门别类的 ActorMessageHandler 标签系实体类，大概 ET7 框架里只需要调用接口的申明方法就可以了？总之，就是 Handle() Run() 两大类方法的方法定义发生了变化，但现在还没弄明白本质，为什么在 ET7 框架里可以简化，实现简化的原理?
  - 【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】
  - **【PlayerComponent 类重复】**：狠奇怪：删除了说找不到类，不删除说重复了，感觉台式机应用有延迟？反应狠慢。。。。。文件嵌套想要显示所有嵌套文件的时候，要狠久狠久重启好几次才反应得过来
    - 原本有两个类都是如上面这个类这样，但有时候台式机反应稍快一点儿，就是一个类找不到出现上面的情况。破电脑的延迟反应，弄得我都要怀疑 VS 应用被别人操控了。。。
    - 【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!】
- 把还没有用到，但是报错了的几个类删掉：比如记一下：SessionInfoComponent,
  - 还剩最后 26 个最挑战活宝妹的编译错误，今天傍晚会家里改会儿，集中问题明天上午希望能够看懂。【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!】



- 把 Root 根场景以及启动时添加的组件大致看了一遍。想把上面的消息处理器再系统化地看一遍，理解一下，总改不到这个模块相关的编译错误。
- **【ETTask ETVoid 是必须弄懂的】**；看两个小时，像昨天晚上一样真正投入进去看。我相信自己看得懂，弄得透，只是需要投入一点儿时间。
  - 感觉前一个周左右的时间，倍受睡眠困扰。活宝妹做梦也不会想到，昨天的自己会困成那个样子（感觉开 1 小时的车极度困难，太容易睡着。。）。。现在试着一再调整状态，少喝咖啡多运动，最重要的，仍是把学习的状态调整出来调整回来。至少学到活宝妹可以嫁给亲爱的表哥的这一天!!!
  - 这个异步的原理，感觉是弄明白了，今天上午又看了一遍看了会儿。下午去改那些 IMHandler, 希望今天下午能够改彻底。就是真正弄明白了去改（现在的问题就是，几个 IMHandler 的实体实现类，改天这个顾不了那个，没弄明白，接口方法怎么申明定义，才能兼顾所有实例类消息处理器?），不是只改掉了当前的编译错误，等真正运行的时候，一个个运行错误或是异常往外冒!!! 今天脑袋还算清醒，下午好好弄弄这个
- 【爱表哥，爱生活!!! 任何时候，亲爱的表哥的活宝妹，都是一定要嫁给亲爱的表哥的!!!】【三楼上的贱鸡贱畜牲真多!!! 一天到底没想点儿好的】活宝妹还没能嫁给亲爱的表哥，活宝妹就是永远守候在亲爱的表哥的身边!!! 爱表哥，爱生活!!!
- 再然后, 再看下下面的 UnitGateComponent 相关。下午或傍晚有时间的时候，可以再折腾折腾 emacs-org-mode 下划线删除字体设置为斜体。
- **【UnitGateComponent】**加个方法用
  - 这里要把 ActorMessageSenderComponent 组件给弄明白。它有个有序管理字典，记着 actorId 与 ActorMessageSender 的一一对应关系，就可以封装维护消息的自动发送等，以及必要的超时消息管理。
- **【服务端 Actor_PlayerEnterRoom_ReqHandler 这个处理类】**现在还很多问题，需要弄懂，往下改
- 今天晚上会把刚才下午看见、意识到几个模块的问题试着分析明白，记下笔记。

- **ETTask-vs-ETVoid**: 框架里有狠多需要改的地方。今天上午的脑袋好使，把这块儿再仔细好好看下。今天上午把以前不懂的模块都稍微看下，再理解一下
  - 查网页感觉也查不出什么来。还是用源码帮助理解概念。【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!!】
  - 不能把所有基类的 async ETTask 返回参数直接改成 void, 因为框架的顶层应用，服务端或是客户端，当不异步等待结果，如资源包没能下载完成，就接着往下执行，会报空异常。
- 现在的问题是：Protobuf 里 repeated 关键字，好像还是没有处理好，找不到成员变量 Cards. 是因为 Proto2CS 的时候，确实把 repeated 关键字给处理丢了。因为我的.proto 文件里有错误。(这就是上面先前觉得奇怪的原因。因为改这个的过程中把那些错改正了，就可以生成成功并找到相关的消息了)。
- 这部分总感觉弄得不是狠透彻。就再花点儿时间。这段时间产量太低，可以先试着完成其它模块。
- 签于昨天晚上仍然休息不好，**今天上午只把昨天的四个恶心死人不偿命的几个题目稍微写写。希望从今天开始这个周，项目的进展能够更顺利一些。【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】**
- **【HandCardSprite 这个最近要弄明白】**不知道这个类是为什么，整了一堆的错误，它是 ETModel 里的。感觉是常规域，没弄明白为什么常规域还有 ILRuntime 的适配呢?
  - 要把 ILRuntime 热更新第三库，也再弄得明白一点儿【今天上午把这里再看，最好是能够结合源码看看】为什么这个类还要适配 ILRuntime ?
  - 这里这个类，整个框架里只找到这一个用的地方，所以它一定是添加在某个预设或是场景中的某个控件下的。只是参考项目的 unity 客户端，我运行不到打牌的这个界面，就先因为抛出异常而淡能运行。所以还没能找到哪个预设或是场景中的哪个控件添加了这个类，但是当然一定是跟玩家手牌相关的。**【HandCardSprite 是在 handcard 预设里添加了这个脚本】**
  - 这个类今天运行狠奇怪，VS022 里找不到了。。。就是说，VSC 里它是在 Model 客户端的源码里，但是从 VS 里打开，找不到这个类文件所在的文件夹和文件，没有索引好，再添加一下?
  - 那么，为什么前两天被这个 block 住，而那天，好像是有删除掉这个文件，但文件夹应该是还在的才对呀? 我可能还会试着再把它添加回去。
  - 但是，会在把当前几个编译错误改完，试着测试一下客户端现在有的界面之后，再试着添加回去，整理和 develop TractorRoomComponent 界面的内容。【爱表哥，爱生活!!! 活宝妹任何时候就是一定要嫁给亲爱的表哥!!】
  - 今天下午家里再运行一次，当客户端抛异常，应该是某个热更新的资源包没有找到什么的? 所以可以试着自己去解决这个客户端实时运行时抛出的异常。
  - **【参考项目斗地主客户端异常】**: 再运行一次，试着分析，是否可以 unity 里实时运行，如果不可以，为什么不可以?
    - 应该是 LandlordsRom 这个预设与 UI 类型没能连接起来，也就是找不到这个预设。
    - 那为什么打好包的可以呢? 因为打好包的预设包名 LandlordsRoom.unity3d 与游戏逻辑契合，可以找得到
    - 可是仍然感觉奇怪：LandlordsLogin 与 LandlordsLobby, 非常类似都可以找到，为什么就 LandlordsRoom 找不到? 可能 LandlordsRoom 预设还是有某点儿物对特殊的地方。

    * 上面这个暂时跳过。现在仍然主要去看 HandCardSprite 为什么参考项目里可以，而 ET7 里就不可以。
  - 就是上面那个异常，今天下午得去弄明白，为什么只在 unity 实时运行时会抛异常，而如果是三个打包好的客户端，就不会。也就是说，打包好的不存在找不到类、找不到预设、或是找不到任何相关资源的问题。
  - 这个项目 Unity.Model 是需要索引 UnityEngine 以及 UI 等相关模块人的.dll 的。暂时还没弄明白它是怎么加的
  -【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!】

- **ClientComponent** 参考项目组件：去看 ET7 里客户端的 PlayerComponent.
- 【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】今天下午先去看 Tractor 游戏源码，设计重构思路
- 【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- **【亲爱的表哥，这个世界上，只有一个活宝妹，这么心心恋恋，就是一定要嫁给亲爱的表哥!!! 问世间情为何物，直教人生死相许。。亲爱的表哥，一个温暖的怀抱拥抱的魂力可真大呀，管了这如许多年!! 这不，你的活宝妹为了这个温暖的怀抱拥抱，就是一定要嫁给亲爱的表哥!! 不嫁就永远守候在亲爱的表哥的身边!! 爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!!】**
- 亲爱的表哥，活宝妹相信舅舅十岁闯江湖的阅历，活宝妹深深相信亲爱的表哥。活宝妹就是稳稳地永远守候在亲爱的表哥的身边！爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!

5 TODO 其它的：部分完成，或是待完成的大的功能版块，列举

- emacs 那天我弄了好久，把 C-; ISpell 原定绑定的功能解除，重新绑定为自己喜欢的 expand-region. 今天第二次再弄，看一下几分钟能够解决完问题? 我的这个破烂记性呀。。。【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】mingw64 lisp/textmode/flyspell.el 键的重新绑定。这下记住了。还好，花得不是太久。有以前的笔记
  - Windows 10 平台下，C-; 是绑定到了 ISpell 下的某个功能，可是现在这个破 emacs 老报错，连查是绑定给哪个功能，过程报错都被阻止了。。。
- **【IStartSystem:】**感觉还有点儿小问题。认为：我应该不需要同文件两份，一份复制到客户端热更新域。我认为，全框架应该如其它接口类一样，只要一份就可以了。**【晚点儿再检查一遍】**
- 如果这个一时半会儿解决不好，就把重构的设计思路再理一理。同时尽量去改重构的 ET 框架里的编译错误。
- 【Tractor】原 windows-form 项目，源码需要读懂，理解透彻，方便重构。
- 去把【拖拉机房间、斗地主房间组件的，玩家什么的一堆组件】弄明白
- 【任何时候，活宝妹就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!】

6 拖拉机游戏：【重构 OOP/OOD 设计思路】

- 自己是学过，有这方面的意识，但并不是说，自己就懂得，就知道该如何狠好地设计这些类。现在更多的是要受 ET 框架，以及参考游戏手牌设计的启发，来帮助自己一再梳理思路，该如何设计它。

- ET7 重构里，各组件都该是自己设计重构原项目的类的设计的必要起点。可以根据这些来系统设计重构。【活宝妹就是一定要嫁给亲爱的表哥!!!】
- 【GamerComponent】玩家组件管理类，管理所有一个房间的玩家：是对一个房间里四个玩家的（及其在房间里的坐位位置）管理（分东南西北）。可以添加移除玩家。今天晚上来弄这一块儿吧。
- 【Gamer】：每一个玩家
- 【拖拉机游戏房间】：多组件构成
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】

7 IMHandler 接口实现的各种类型消息处理器：需要先理解透彻 ETTask 和 ETVoid

- 可以回去参考前一个游戏参考过的 ET-EUI 里有一部分 ETVoid 的相关使用，可以用作自己理解 ETTask ETVoid 的源码参考。ET-EUI 现笔记本里没有，可以拉一个下来。
  - 【抓下来后用法基本一样，ETVoid 使用的地方基本一样】。不一样的地方是，它扩展（或者说 **重新自定义了**）了不少 **消息处理器接口类**。那么就是说，ET-EUI 并没有、也做不到一个框架整个框架只使用一个消息处理器接口

- 【主要使用:】主要是【服务端】处理客户端消息请求，用来定义处理逻辑。但是网络调用与返回大多是异步的，所以会有很多使用 ETTask 或是 ETVoid|Void 作为返回值的地方。主要是两个常用方法的接口定义，兼顾整个框架的接口定义。
- **【参照 ET-EUI】**：如果再来参照这个例子项目，或许也可以多定义几个不同的消息处理器接口，就不必强制整个框架只实现一个接口而顾 A 顾不了 B 了。那么如果下午继续参照这个例子，头脑清醒的时候，就要搞明白：**不同接口类，到底适用哪类消息？【可以把这部分再分析理解一下，总结在下面，但是区分清楚，哪个来自 ET-EUI】**就是需要理解透彻再改，不要再循环无限制地改。
- 【两点不透彻:】ETTask|ETVoid|Void 到底使用什么返回值？另则，async-await 方法是定义为异步，还是非异步。如果 async 定义异步，什么地方必须有 await 调用？
- 感觉 ETTask|ETVoid 基本弄明白了，可是这里仍然是整个框架，感觉最为复杂不好修改的地方。可能我还是把网络异步调用没能弄得狠明白。
  - 要保证两个方法里，若是同步方法，方法中就一定不能有需要异步等待的地方（否则运行时会抛异常，对象为空之类的各种因为异步操作不能及时完成而抛的空异常）。就是，**如果需要使用异步方法，不可以改为同步，同步返回 void 或其它任何.**
  - 改的过程中，方法中曾经有过 await 调用异步方法的逻辑（方法），不能因想去掉编译错误就去掉 await 调用关键字，错误地改成同步方法，因为暂时去掉了编译错误，运行时一定会抛异常。
  - 下午改这个：把 ET 框架的原分支，ET-EUI 项目的消息处理器接口相关的类，需要对照原项目，确认没被自己弄得乱七八糟的
  - 活宝妹只是重构改一个游戏项目。他们大可不必发疯犯贱。写这些，都是在写活宝妹的心理阴影，被他们发疯弄怕了。。。
  - 【爱表哥，爱生活!!! 任何时候，亲爱的表哥的活宝妹，就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!】

7.1 IMHandler: interface 消息处理器接口类。它有 2 个实现抽象类：AMHandler，AMRpcHandler

- 实现这个接口类型主要分为两个抽象类：AMHandler，AMRpcHandler. 所以 ET 框架里现有一个接口 IMHandler, 两个抽象实现类。其它是参考自 ET-EUI.
- 今天上午暂时只看取了这个接口，和两个抽象方法这里。后面任何的个体实现类，都还没有细看，不太懂。

```
public interface IMHandler {
    // 下面，返回类型不对
    void Handle(Session session, object message); // 这里返回类型，仍然应该是 ETTask，或者可能的 ETVoid ？
    // 消息处理器帮助类，在程序域加载的时候，会自动扫描程序域里的 ActorMessageHander 标签，会想要拿消息的【发送类型】与消息的【返回
    Type GetMessageType();
    Type GetResponseType();// 这里现在想的话，可能存在的问题是：有些消息不带返回类型，那么就是说，那类消息不需要处理吗？它只是不需要
}
```

7.2 AMHandler<Message>: abstract 抽象基类：两个方法的返回类型，成为现在全框架的理解与实现难点

- AMHandler 类，这个类相比 AMRpcHandler 更加简单一些，因为这个类型的处理，不需要关心回消息
- 实现了接口 IMHandler 的 Handle 异步方法，具体逻辑为：

- 将传进来的 msg 首先转换为模板类，在 AMHandler 类里面为 Message，具体应该为实现 AMHandler 的类的具体数据类。
- 根据数据类，以及 session 生成一些报错日志，方便调试
- 调用 Run 方法，将 session 及具体的数据类传进去
- 实际继承抽象类 AMHandler 的类型，会实现这个接口，从而走向各自的处理。

```
public abstract class AMHandler<Message>: IMHandler where Message : class {

// 虽然我这么改，可以暂时消掉编译错误。但改得不对，现在消掉了编译错误，等编译通过，运行时错误会一再崩出来的。。。
    protected abstract void Run(Session session, Message message);
    // protected abstract ETTask Run(Session session, Message message);

    // public async ETVoid Handle(Session session, object msg)// 【参考来源:】ET-EUI, 返回类型 ETVoid 应该更合适,
    public void Handle(Session session, object msg) {
        Message message = msg as Message;
        if (message == null) {
            Log.Error($" 消息类型转换错误: {msg.GetType().Name} to {typeof (Message).Name}");
            return;
        }
        if (session.IsDisposed) {
            Log.Error($"session disconnect {msg}");
            return;
        }
        this.Run(session, message);
        // this.Run(session, message).Coroutine();
    }
    public Type GetMessageType() {
        return typeof (Message);
    }
    public Type GetResponseType() {
        return null;
    }
}
```

7.3 AMRpcHandler: 去抓的 ET7 框架的源码，可以用来【校正】其它被自己改错的

- 去抓的 ET7 框架的源码，可以用来校正其它被自己改错的类，或是方法定义。
- 这个类比 AMHandler 要多传入一个模板类，主要用于处理那些约定好带返回数据的。**【注意下面，是参考网络上别人的理解】**。他们同样与活宝妹一样，也是一知半解，狠多地方说得也未必对。
- 简单解释一下，现在 ET 协议数据主要分为两个类型，一种是来了消息直接自己处理的，另外一种是来了消息，自己处理完毕后，还需要将一些数据给返回的。【带返回类型，和不带返回类型】的消息。觉得它理解得不对，不适用于这里。因为 Rpc 这里感觉，更多的是说，进程间，或是现在 ET7 重构后的不同 SceneType 之间，比如注册登录服与网关服间的消息，内网消息等？Rpc-rpc-rpc...
- 主要的处理流程与 AMHandler 大体相同，需要注意的:
  - 传入的模板类有类型要求，除了是 class 外，第一个需要是实现 IRequest 接口，第二个是实现 IResponse 接口，他们分别对应了传进来的协议数据类型，以及需要返回的协议数据类型。
  - IRequest 类型，具有 RpcId，这个 id 用来标识一个传入协议数据，同时又将它设置到 response 返回数据中的 RpcId 中，这样发送数据返回的时候，就能找到那个和他具有相同 RpcId 传入协议数据，这种关系一对一，从而能进行进一步处理。**【谁发来的消息，就返回消息给谁——发送者】**
  - 回调函数 Reply（），即当处理完传入数据后，需要马上装配好返回数据，并将其发送回去，所以需要一个回调函数将 response，通过 session 发送回去. 前面说的是以前的

框架。现在的回调过程，直接通过 Session 会话框走网络层将消息发回去，不用再弄个 Action<T> 来触发调用回调了。

- 在 Run 中，需要传输上下文 session，接受的协议数据 request，需要返回的协议数据 response，以及回调函数 reply

```
public abstract class AMRpcHandler<Request, Response>: IMHandler where Request : class, IRequest where Response : class, IR
    protected abstract ETTask Run(Session session, Request request, Response response);
// 看一下：这个返回类型 void 算怎么回事？如果这个可以运行通，可以用作参照，来修改其它、另一个上一个不带返回消息的抽象类
    public void Handle(Session session, object message) {
        HandleAsync(session, message).Coroutine();
    }
    private async ETTask HandleAsync(Session session, object message) {
        try {
            Request request = message as Request;
            if (request == null)
                throw new Exception($" 消息类型转换错误：{message.GetType().Name} to {typeof (Request).Name}");
            int rpcId = request.RpcId;
            long instanceId = session.InstanceId;
            Response response = Activator.CreateInstance<Response>();
            try { // 不懂：下面一句是在干什么，执行对发送来消息的处理，写返回数据？
                await this.Run(session, request, response);
            }
            catch (Exception exception) { // 如果出异常：写异常结果
                Log.Error(exception);
                response.Error = ErrorCore.ERR_RpcFail;
                response.Message = exception.ToString();
            }
            // 等回调回来，session 可以已经断开了，所以需要判断 session InstanceId 是否一样
            if (session.InstanceId != instanceId)
                return;
            response.RpcId = rpcId; // 在这里设置 rpcId 是为了防止在 Run 中不小心修改 rpcId 字段。【谁发来的消息，就返回消息给谁一
            session.Send(response); // 把返回消息发回去
        }
        catch (Exception e) { // 捕获异步操作过程中的异常
            throw new Exception($" 解释消息失败：{message.GetType().FullName}", e);
        }
    }
    public Type GetMessageType() {
        return typeof (Request);
    }
    public Type GetResponseType() {
        return typeof (Response);
    }
}
```

7.4 IMHandler 【ET-EUI】：主要是与上面对比

- 因为对于 ET-EUI 里，它为什么要再多自定义几个接口，这个版块不够熟悉
- 我觉得把两个框架的几个接口与抽象实体类提出来对比理解一下是对的，因为没明白，为什么 ET7 里会重构成那个样子。可能也网上查下再帮助理解一下。

```
public interface IMHandler {
    void Handle(Session session, object message);
    Type GetMessageType();
    Type GetResponseType();
    // ETTask Run(Session session, M2C_TestActorMessage message); // 把这里去掉，感觉加得不对
}
```

7.5

7.6 IMActorHandler: 【ET-EUI: 仅作参考】大概参考 ET-EUI 来的，它的目的应该是把最基类的接口，与其它两类的接口相区分开来

- 从这个小节开始，主要是整理的 ET-EUI 里面的接口与抽象实现类。可能主要两个实现扩展类：（A|I）MHandler, （A|I）MRpcHandler, 都看一下

- 大概参考 ET-EUI 来的，它的目的应该是把最基类的接口，与其它两类的接口相区分开来。但是，我想、猜测、理解的话，应该是上面一个接口，如果能够把 ETTask 与 ETVoid 狠好地统一的话，应该一个接口可能可以足以整个框架使用的。
- 消息处理器：基本可以理解为，总都是【服务端】的消息处理器。【客户端】更多的是发请求消息，和接收消息。客户端狠少涉及什么消息处理。
- 下面的接口，哪个例子也没有参照，自己改的。极有可能都是错的

```
public interface IMActorHandler {
    // 下面，参考的是 ET-EUI 可能是 6.0 版本。ET7 里，可能接口还可以简化，还是 Actor 消息机制模块简化了，不一定如下面这样
    void Handle(Entity entity, int fromProcess, object actorMessage);
    // ETTask Handle(Entity entity, object actorMessage, Action<IActorResponse> reply);
    Type GetRequestType();
    Type GetResponseType();
}
```

7.7 AMActorLocationHandler: 源码被我改动了

- 源码被我改动了，正确性与否没有关系，主要是帮助自己梳理一下几大不同的类型，到改编译错误的时候，能够边修改边弄明白。

```
[EnableClass]
public abstract class AMActorLocationHandler<E, Message>: IMActorHandler where E : Entity where Message : class, IAct
    // protected abstract ETTask Run(E entity, Message message);
    protected abstract void Run(E entity, Message message);
    // public async ETTask Handle(Entity entity, int fromProcess, object actorMessage) {
    public void Handle(Entity entity, int fromProcess, object actorMessage) {
        if (actorMessage is not Message message) {
            Log.Error($" 消息类型转换错误: {actorMessage.GetType().FullName} to {typeof (Message).Name}");
            return;
        }
        if (entity is not E e) {
            Log.Error($"Actor 类型转换错误: {entity.GetType().Name} to {typeof (E).Name} --{typeof (Message).Name}");
            return;
        }
        ActorResponse response = new() {RpcId = message.RpcId};
        ActorHandleHelper.Reply(fromProcess, response);
        // await this.Run(e, message);
        this.Run(e, message);
    }
    public Type GetRequestType() {
        return typeof (Message);
    }
    public Type GetResponseType() {
        return typeof (ActorResponse);
    }
}
```

7.8 AMActorLocationRpcHandler: 上面接口实现类的一个使用例子。我把返回参数改了

```
[EnableClass]
public abstract class AMActorLocationRpcHandler<E, Request, Response>: IMActorHandler where E : Entity where Request : clas
    // protected abstract ETTask Run(E unit, Request request, Response response);
    protected abstract void Run(E unit, Request request, Response response);
    // public async ETTask Handle(Entity entity, int fromProcess, object actorMessage) {
    public void Handle(Entity entity, int fromProcess, object actorMessage) {
        try {
            if (actorMessage is not Request request) {
                Log.Error($" 消息类型转换错误: {actorMessage.GetType().FullName} to {typeof (Request).Name}");
                return;
            }
            if (entity is not E ee) {
                Log.Error($"Actor 类型转换错误: {entity.GetType().Name} to {typeof (E).Name} --{typeof (Request).Name}");
                return;
            }
            int rpcId = request.RpcId;
```

```
            Response response = Activator.CreateInstance<Response>();
            try {
                //await this.Run(ee, request, response);
                this.Run(ee, request, response);
            }
            catch (Exception exception) {
                Log.Error(exception);
                response.Error = ErrorCore.ERR_RpcFail;
                response.Message = exception.ToString();
            }
            response.RpcId = rpcId;
            ActorHandleHelper.Reply(fromProcess, response);
        }
        catch (Exception e) {
            throw new Exception($" 解释消息失败: {actorMessage.GetType().FullName}", e);
        }
    }
    public Type GetRequestType() {
        return typeof (Request);
    }
    public Type GetResponseType() {
        return typeof (Response);
    }
    // 这里涉及的就是那个接口方法的定义
    public ETTask Handle(Entity entity, object actorMessage, Action<IActorResponse> reply) => throw new NotImplementedExcep
}
```

7.9 AMActorLocationRpcHandler: Rpc 就是进程间消息（或是 ET7 重构为 SceneType 之后的多核间消息）

```
[EnableClass]
public abstract class AMActorLocationRpcHandler<E, Request, Response>: IMActorHandler where E : Entity where Request : clas
    // protected abstract ETTask Run(E unit, Request request, Response response);
    protected abstract void Run(E unit, Request request, Response response);
    // public async ETTask Handle(Entity entity, int fromProcess, object actorMessage) {
    public void Handle(Entity entity, int fromProcess, object actorMessage) {
        try {
            if (actorMessage is not Request request) {
                Log.Error($" 消息类型转换错误: {actorMessage.GetType().FullName} to {typeof (Request).Name}");
                return;
            }
            if (entity is not E ee) {
                Log.Error($"Actor 类型转换错误: {entity.GetType().Name} to {typeof (E).Name} --{typeof (Request).Name}");
                return;
            }
            int rpcId = request.RpcId;
            Response response = Activator.CreateInstance<Response>();
            try {
                //await this.Run(ee, request, response);
                this.Run(ee, request, response);
            }
            catch (Exception exception) {
                Log.Error(exception);
                response.Error = ErrorCore.ERR_RpcFail;
                response.Message = exception.ToString();
            }
            response.RpcId = rpcId;
            ActorHandleHelper.Reply(fromProcess, response);
        }
        catch (Exception e) {
            throw new Exception($" 解释消息失败: {actorMessage.GetType().FullName}", e);
        }
    }
    public Type GetRequestType() {
        return typeof (Request);
    }
    public Type GetResponseType() {
        return typeof (Response);
    }
}
```

7.10 ActorHandleHelper 静态帮助类：包装了必要的方法，帮助自动化回复相关回调消息

7.11 MessageDispatcherComponent: 这些是一个系统的，需要放在一起总结

7.12 MessageDispatcherComponentSystem: