Unity Android SDK/NDK 俄罗斯方块砖 3D 小游戏

deepwaterooo

September 27, 2022

Contents

1 模块搭建 **1**

2 把原理弄懂 **1**

3 环境弄得比较好的包括： **1**

4 ILRuntime 库的系统再深入理解 **2**
4.1 ILRuntime 基本原理 . . . 2
4.2 ILRuntime 热更流程 . . . 2
4.3 ILRuntime 主要限制 . . . 3
4.4 ILRuntime 启动调试 . . . 3
4.5 线上项目和资料 . . . 3

5 Framework.Core 核心理解：现在有足足的干劲把 ILRuntime + MVVM 热更新框架里的方方面面都理解消化透彻！爱表哥爱生活 **3**
5.1 Adaptor . . . 3
5.1.1 IDisposableAdaptor : CrossBindingAdaptor . . . 3
5.1.2 IEnumeratorObjectAdaptor : CrossBindingAdaptor { // 叠代器适配器 . . . 4
5.1.3 InterfaceCrossBindingAdaptor : CrossBindingAdaptor：就是实现基类里的三个方法呀 . . . 5
5.1.4 MonoBehaviourAdapter : CrossBindingAdaptor: ./Plugins/ILRuntime/Adapters 包裹里的 . . . 5
5.2 Factory . . . 6
5.3 Inject . . . 6
5.4 Wrap . . . 6
5.5 GameApplication: 游戏入口类 . . . 6
5.6 HotFixILRunTime ： SingletonMono<HotFixILRunTime>, IHotFixMain: 因为跨域交互，所以即使是热更新包，也继承自 unity MonoBehaviour . . . 8
5.7 HotFixReflector : SingletonMono<HotFixReflector>, IHotFixMain . . . 10
5.8 IHotFixMain interface . . . 11

6 Framework.MVVM: Unity 中定义好的 MVVM 架构；使用 ILRuntime 定义实现了必要的跨域跨程序集适配，以及数据观察回调等 **11**
6.1 DataBinding: MVVM 具备双向数据绑定功能；这里这个框架里通过代理观察者模式来实现数据的改变通知与回调 . . . 11
6.1.1 BindableProperty<T> ： 泛型基类，可绑定的属性 . . . 11
6.1.2 ObservableList<T> : IList<T> ： 可被观察的链表，同样包装成泛型基类 . . . 11
6.1.3 PropertyBinder<ViewModelBase>: ViewModel(的基类以及继承类) 的绑定辅助相关方法定义类 . . . 13
6.2 Message: 自定义的消息机制吗？这里还没能联系上下文，完全不知道这里是在做什么？ 13
6.2.1 MessageArgs<T>: 跨域跨程序集的消息参数泛型类？ . . . 13

6.2.2 MessageAggregator<T> : Singleton<MessageAggregator<T>>: 单例模式的泛型管理类? 用一个字典来管理消息? . . . 13
6.3 Module: 这里是模块级别的定义; 在模块层面级别上; 在 ViewModel 层面级别上等不面层面上的跨域适配定义与实现 . . . 14
6.3.1 ModuleBase.cs . . . 14
6.3.2 ModuleBaseAdapter : CrossBindingAdaptor: 在模块层面上自定义实现了: 模块层面级别的跨域跨程序集适配器 . . . 14
6.4 View . . . 15
6.4.1 IView<ViewModelBase> . . . 15
6.4.2 UnityGuiView: IView<ViewModelBase>, 实现或是覆写基类以及泛型里的各种定义过的方法 . . . 15
6.4.3 UnityGuiViewAdapter : CrossBindingAdapto: 最鼻祖的实体基类 . . . 17
6.5 ViewModel . . . 21
6.5.1 ViewModelBase.cs . . . 21
6.5.2 ViewModelBaseAdapter : CrossBindingAdaptor: 自定义实现了 ViewModel 的跨域跨程序集适配 . . . 22

7 Framework.Util 各种帮助类 **24**
7.1 . . . 24
7.2 . . . 24
7.3 . . . 24
7.4 . . . 24
7.5 Singleton<T> + SingletonMono<T> : MonoBehaviour . . . 24

8 HotFix 中使用 MVVM 架构实现热更新的搭配与相关的链接 **24**

9 ILRuntime 类库里源码的基本理解, 最重要的涉及到的相关的类与方法摘要 **24**
9.1 CrossBindingAdaptor : IType 跨域 (程序集) 绑定适配器 + CrossBindingAdaptorType interface 公用接口类 (为什么要这个公用接口类?) . . . 24
9.2 . . . 27
9.3 . . . 27
9.4 . . . 27
9.5 . . . 27

1 模块搭建

- # only inlcude two levels in TOC
- ILRuntime 的消化理解, 以及与 MVVM 同用时的搭配理解消化
- 热更新模块服务器模块的理解与消化搭建:

2 把原理弄懂

- 热更新模块的实充: 以前的设计模式和实现的功能还是比较完整的; 现在更成熟一点儿, 需要把热更新模块补充出来;
- ILRuntime + MVVM 框架设计: 两者结合, 前几年的时候没能把 MVVM 理解透彻;
- 上次前几年主要的难点: 好像是在把 MVVM 双向数据绑定理解得不透彻; 那么这次应该就狠没有问题了, 更该寻求更好的设计与解决方案
- 性能优化: 另外是对其实高级开发的越来越熟悉, 希望应用的性能表现, 尤其是渲染性能与速度等、这些更为高级和深入的特性成为这次二次开发的重点。

- 现在是把自己几年前的写的游戏全忘记了，需要回去把自己的源码找出来，再读一读熟悉一下自己的源码，了解当时设计的估缺点，由此改进更将

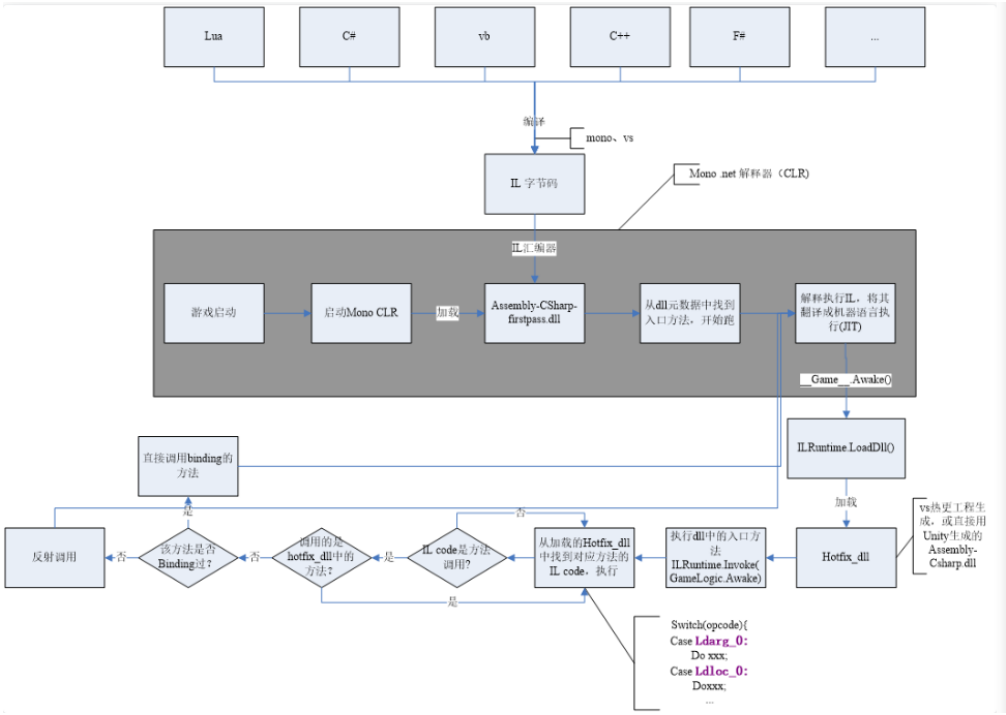
3 环境弄得比较好的包括：

- 输入法的搭建：终于用到了自己之前用过的好用的输入法
- 这两天开车疲累，最迟明天中午会去南湾找房间出租，尽快解决搬家的问题；昨天晚上回来得太晚了，一路辛苦，路上只差睡着，回到家里补觉补了好多个小时。
- 小电脑，笔记本电脑里的游戏环境搭建，今天下午去图书馆里弄（今天下午去图书馆里把需要借助快速网络来完成的事情都搭建好；家里被恶房东故意整了个腾腾慢的网，故意阻碍别人的发展，谁还愿意再这样的环境中继续住下去呢?!!!）

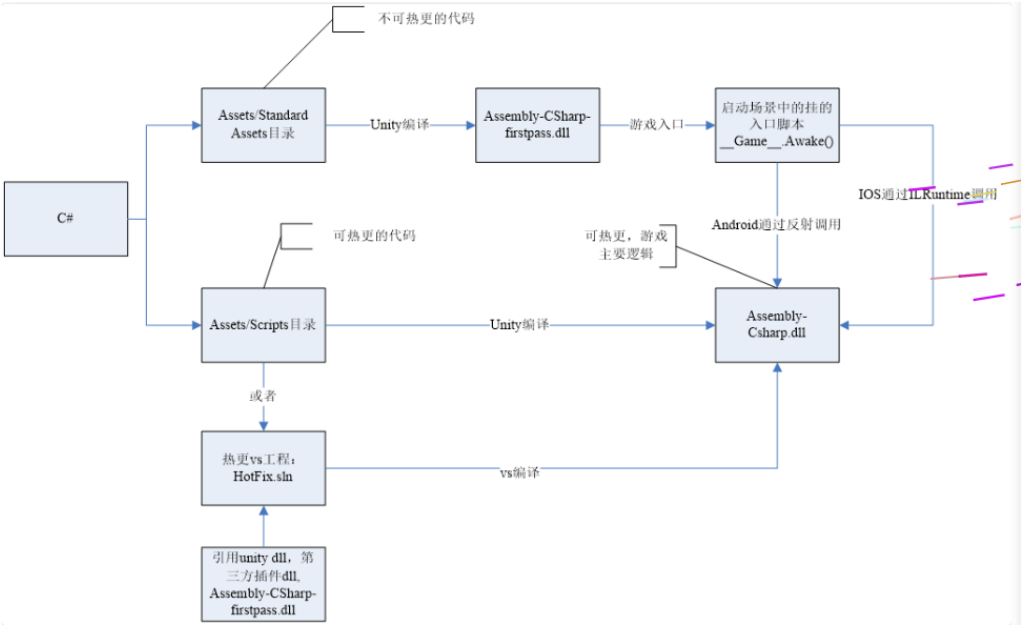
4 ILRuntime 库的系统再深入理解

4.1 ILRuntime 基本原理

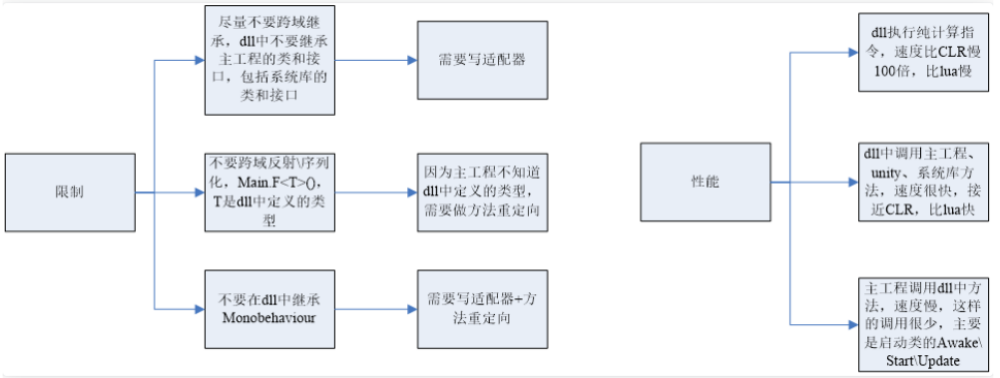
- ILRuntime 借助 Mono.Cecil 库来读取 DLL 的 PE 信息，以及当中类型的所有信息，最终得到方法的 IL 汇编码，然后通过内置的 IL 解译执行虚拟机来执行 DLL 中的代码。IL 解释器代码在 ILIntepreter.cs，通过 Opcode 来逐语句执行机器码，解释器的代码有四千多行。



4.2 ILRuntime 热更流程



4.3 ILRuntime 主要限制



4.4 ILRuntime 启动调试

- ILRuntime 建议全局只创建一个 AppDomain，在函数入口添加代码启动调试服务

```
appdomain.DebugService.StartDebugService(56000)
```

- 运行主工程 (Unity 工程)
- 在热更的 VS 工程中点击 - 调试 - 附加到 ILRuntime 调试，注意使用一样的端口
- 如果使用 VS2015 的话需要 Visual Studio 2015 Update3 以上版本

4.5 线上项目和资料

- 掌趣很多项目都是使用 ILRuntime 开发，并上线运营，比如：真红之刃，境 · 界灵压对决，全民奇迹 2，龙族世界，热血足球

- 初音未来: 梦幻歌姬使用补丁方式：https://github.com/wuxiongbin/XIL
- 本文流程图摘自：ILRuntime 的 QQ 群的《ILRuntime 热更框架.docx》(by a 704757217)
- Unity 实现 c# 热更新方案探究 (三): https://zhuanlan.zhihu.com/p/37375372

5 Framework.Core 核心理解: 现在有足足的干劲把 ILRuntime + MVVM 热更新框架里的方方面面都理解消化透彻！爱表哥爱生活

5.1 Adaptor

5.1.1 IDisposableAdaptor : CrossBindingAdaptor

```
public class IDisposableAdaptor : CrossBindingAdaptor {
// 实现基类里所定义的三个方法
    public override Type BaseCLRType {
        get {
            return typeof(IDisposable);
        }
    }
    public override Type AdaptorType {
        get {
            return typeof(Adaptor);
        }
    }
    public override object CreateCLRInstance(ILRuntime.Runtime.Enviorment.AppDomain appdomain, ILTypeInstance instance) {
        return new Adaptor(appdomain, instance);
    }

    internal class Adaptor : IDisposable, CrossBindingAdaptorType {
        ILTypeInstance instance;  // 基类 CrossBindingAdaptorType   里所必须有的一个实例
        ILRuntime.Runtime.Enviorment.AppDomain appdomain;

        public Adaptor() {  }
        public Adaptor(ILRuntime.Runtime.Enviorment.AppDomain appdomain, ILTypeInstance instance) {
            this.appdomain = appdomain;
            this.instance = instance;
        }
        public ILTypeInstance ILInstance { get { return instance; } } // getter

        IMethod _Dispose;
        public void Dispose() {
            if (_Dispose == null)
                _Dispose = instance.Type.GetMethod("Dispose", 0);
            if (_Dispose != null)
                appdomain.Invoke(_Dispose, instance);
        }
    }
}
```

5.1.2 IEnumeratorObjectAdaptor : CrossBindingAdaptor { // 叠代器适配器

```
public class IEnumeratorObjectAdaptor : CrossBindingAdaptor { // 叠代器适配器
// 基类里的三个方法的实现
    public override Type BaseCLRType {
        get {
            return typeof(IEnumerator<object>);
        }
    }
    public override Type AdaptorType {
        get {
            return typeof(Adaptor);
        }
    }
    public override object CreateCLRInstance(ILRuntime.Runtime.Enviorment.AppDomain appdomain, ILTypeInstance instance) {
        return new Adaptor(appdomain, instance);
    }

// 摘不明白：这些个 IXxxxx IEnumerator<T>   之类的，是在哪里定义的，还是说是 C #语言里本来就有的呢？
```

```
// 弄不明白的这些个以 I 打头的：都定义在 plugins 里的 ILRuntime 的包裹里，改天可以捡起来看一下
    internal class Adaptor : IEnumerator<object>, CrossBindingAdaptorType {
        ILTypeInstance instance;
        ILRuntime.Runtime.Enviorment.AppDomain appdomain;

        public Adaptor() {  }
        public Adaptor(ILRuntime.Runtime.Enviorment.AppDomain appdomain, ILTypeInstance instance) {
            this.appdomain = appdomain;
            this.instance = instance;
            _get_Current = instance.Type.GetMethod(".get_Current", 0);
        }

        public ILTypeInstance ILInstance { get { return instance; } }  // getter

        public object Current { // 叠代器适配器所特有的方法，当前元素
            get {
                var obj = appdomain.Invoke(_get_Current, null);
                return obj;
            }
        }
// 这里的几个方法，是因为叠代器需要用到，所以不得不定义的吗？还有什么基类之类的吗？
        IMethod _MoveNext;
        IMethod _get_Current;
        IMethod _Reset;
        IMethod _Dispose;
        public bool MoveNext() {
            if (_MoveNext == null)
                _MoveNext = instance.Type.GetMethod("MoveNext", 0);
            if (_MoveNext != null)
                return (bool)appdomain.Invoke(_MoveNext, instance);
            return false;
        }
        public void Reset() {
            if (_Reset == null)
                _Reset = instance.Type.GetMethod("MoveNext", 0);
            if (_Reset != null)
                appdomain.Invoke(_Reset, instance);
        }
        public void Dispose() {
            if (_Dispose == null)
                _Dispose = instance.Type.GetMethod("Dispose", 0);
            if (_Dispose != null)
                appdomain.Invoke(_Dispose, instance);
        }
    }
}
```

5.1.3 InterfaceCrossBindingAdaptor：CrossBindingAdaptor：就是实现基类里的三个方法呀

```
public class InterfaceCrossBindingAdaptor : CrossBindingAdaptor { // 就是实现基类里的三个方法呀
    public override Type BaseCLRType {
        get {
            return typeof(IEnumerator);
        }
    }
    public override Type AdaptorType {
        get {
            return typeof(IEnumeratorObjectAdaptor.Adaptor);
        }
    }
    public override object CreateCLRInstance(ILRuntime.Runtime.Enviorment.AppDomain appdomain, ILTypeInstance instance) {
        return new IEnumeratorObjectAdaptor.Adaptor(appdomain, instance);
    }
}
```

5.1.4 MonoBehaviourAdapter：CrossBindingAdaptor：./Plugins/ILRuntime/Adapters包裹里的

```
// ./Plugins/ILRuntime/Adapters/MonoBehaviourAdapter.cs: 注意这个程序所在的包裹
public class MonoBehaviourAdapter : CrossBindingAdaptor {
// 实现基类里的三个抽象方法
    public override Type BaseCLRType {
        get {
            return typeof(MonoBehaviour);
```

```
        }
    }
    public override Type AdaptorType {
        get {
            return typeof(Adaptor);
        }
    }
    public override object CreateCLRInstance(ILRuntime.Runtime.Enviorment.AppDomain appdomain, ILTypeInstance instance) {
        return new Adaptor(appdomain, instance);
    }

// 为了完整实现 MonoBehaviour 的所有特性，这个 Adapter 还得扩展，这里只抛砖引玉，只实现了最常用的 Awake, Start 和 Update
// 像我狠熟悉的安卓 Activity/Fragment 的生命周期有很多回调方法一样，MonoBehavior 也有好几个生命周期回调方法可供实现扩展
    public class Adaptor : MonoBehaviour, CrossBindingAdaptorType {
        ILTypeInstance instance;
        ILRuntime.Runtime.Enviorment.AppDomain appdomain;

        public Adaptor() { }
        public Adaptor(ILRuntime.Runtime.Enviorment.AppDomain appdomain, ILTypeInstance instance) {
            this.appdomain = appdomain;
            this.instance = instance;
        }
        public ILTypeInstance ILInstance { get { return instance; } set { instance = value; } }
        public ILRuntime.Runtime.Enviorment.AppDomain AppDomain { get { return appdomain; } set { appdomain = value; } }

// Awake() Start() Update() 三个生命周期架设方法的跨域适配
        IMethod mAwakeMethod;
        bool mAwakeMethodGot;
        public void Awake() {
            // Unity 会在 ILRuntime 准备好这个实例前调用 Awake，所以这里暂时先不掉用
            if (instance != null) {
                if (!mAwakeMethodGot) {
                    mAwakeMethod = instance.Type.GetMethod("Awake", 0);
                    mAwakeMethodGot = true;
                }
                if (mAwakeMethod != null) {
                    appdomain.Invoke(mAwakeMethod, instance, null);
                }
            }
        }
        IMethod mStartMethod;
        bool mStartMethodGot;
        void Start() {
            if (!mStartMethodGot) {
                mStartMethod = instance.Type.GetMethod("Start", 0);
                mStartMethodGot = true;
            }
            if (mStartMethod != null) {
                appdomain.Invoke(mStartMethod, instance, null);
            }
        }
        IMethod mUpdateMethod;
        bool mUpdateMethodGot;
        void Update() {
            if (!mUpdateMethodGot) {
                mUpdateMethod = instance.Type.GetMethod("Update", 0);
                mUpdateMethodGot = true;
            }
            if (mStartMethod != null) {
                appdomain.Invoke(mUpdateMethod, instance, null);
            }
        }

        public override string ToString() {
            IMethod m = appdomain.ObjectType.GetMethod("ToString", 0);
            m = instance.Type.GetVirtualMethod(m);
            if (m == null || m is ILMethod) {
                return instance.ToString();
            } else
                return instance.Type.FullName;
        }
    }
}
```

5.2 Factory

5.3 Inject

5.4 Wrap

5.5 GameApplication: 游戏入口类

```
// 游戏入口类
public class GameApplication : MonoBehaviour {
    private const string TAG = "GameApplication";

    private static GameApplication _instance;
    public static GameApplication Instance {
        get {
            return _instance;
        }
    }
    public IHotFixMain HotFix {
        get;
        set;
    }

// 五个用户、客户端可配置变量，以及它们的根据用户配置（读取系统中用户配置文件里的相关五个配置）
    // 是否使用 PDB 调试信息
    public bool usePDB = false;
    // 是否使用 ILRuntime 模式热更新
    public bool useILRuntime = false;
    // 是否使用本地资源
    public bool useLocal = false;
    // 资源服务器路径
    public string webRoot = string.Empty;
    // 强制登录
    public bool forceLogin = false;

    public ScreenRaycaster ScreenRaycaster {
        get;
        private set;
    }
// 这个公用方法应该是跟游戏中时常需要接入一个或是多个游戏 SDK 相关，提供便利接入方法
    public ShareSDK ShareSDK {
        get;
        private set;
    }
    void Awake() {
        _instance = this;
        ScreenRaycaster = GameObject.Find("Gestures").GetComponent<ScreenRaycaster>();
        DontDestroyOnLoad(gameObject); // <<<<<<<<<<<<<<<<<<<< 我自己的游戏中实现过这个，可是现在回想得好辛苦呀。。。爱表哥爱生活爱
        //InitializeClientConfig();
        //InitializeSDKs();
        CoroutineHelper.StartCoroutine(Initialize()); // 协程：
#region TestSamples
        //FingerEventTemp.Instance.RegisterGestureEvents();
        //TestNTS.Instance.TestLinesAngle();
        //GeometryManager.Instance.Test();
#endregion
    }
// 客户端的配置是写在一个文件里的，需要的话直接读就可以了，安卓系统很多地方也这样
    void InitializeClientConfig() {
        var str = FileHelp.ReadString("ClientConfig.txt");
        if (!string.IsNullOrEmpty(str)) {
            JsonObject jsonObject = JsonSerializer.Deserialize(str) as JsonObject;
            if (jsonObject != null) {
                if (jsonObject.ContainsKey("usePDB"))
                    usePDB = (bool)jsonObject["usePDB"];
                if (jsonObject.ContainsKey("useILRuntime"))
                    usePDB = (bool)jsonObject["useILRuntime"];
                if (jsonObject.ContainsKey("useLocal"))
                    usePDB = (bool)jsonObject["useLocal"];
                if (jsonObject.ContainsKey("webRoot"))
                    ResourceConstant.ResourceWebRoot = jsonObject["webRoot"].ToString();
                if (jsonObject.ContainsKey("forceLogin"))
                    forceLogin = (bool)jsonObject["forceLogin"];
            }
        }
    }
```

```
    void InitializeSDKs() {
        if (Application.platform == RuntimePlatform.IPhonePlayer || Application.platform == RuntimePlatform.Android)
            InitializeShareSDK();
    }
    void InitializeShareSDK() {
        ShareSDK = GetComponent<ShareSDK>();
        ShareSDK.authHandler = AuthResultHandler;
        ShareSDK.Authorize(PlatformType.WeChat);
    }
    // ShareSDK 执行授权回调
    void AuthResultHandler(int reqID, ResponseState state, PlatformType type, Hashtable result) {
        if (state == ResponseState.Success) {
            Debug.Log("ShareSDK authorize success!");
        } else if (state == ResponseState.Fail) {
            Debug.Log("fail! error code = " + result["error_code"] + "; error msg = " + result["error_msg"]);
        } else if (state == ResponseState.Cancel) {
            Debug.Log("cancel!");
        }
    }

// 协程是说：游戏启动时，给这个控件 (gameObject) 加载运行时元件 ResourceMap（本质上是个脚本程序）；加载完毕自动触发 StartHotFix()
    IEnumerator Initialize() {
        ResourceMap resourceMap = gameObject.AddComponent<ResourceMap>();
        resourceMap.OnInitializeSuccess += StartHotFix;
        ResourceConstant.Loader = resourceMap;
        yield return new WaitForEndOfFrame();
    }
    public void StartHotFix() {
        Debug.Log(TAG + ": StartHotFix()");
        Debug.Log(TAG + " useILRuntime: " + useILRuntime);
        if (Application.platform == RuntimePlatform.IPhonePlayer) {
            HotFix = HotFixILRunTime.Instance;
        } else {
            if (useILRuntime) { // 使用热更新程序集的实例
                HotFix = HotFixILRunTime.Instance;
            } else { // 这里是，还需要再理解消化一下
                HotFix = HotFixReflector.Instance;
            }
        }
    }
}
```

5.6 HotFixILRunTime：SingletonMono<HotFixILRunTime>，IHotFixMain：因为跨域交互，所以即使是热更新包，也继承自 unity MonoBehaviour

```
public class HotFixILRunTime : SingletonMono<HotFixILRunTime>, IHotFixMain { // SingletonMono<T> where T : MonoBehaviour 帮
    public static ILRuntime.Runtime.Enviorment.AppDomain appDomain;

    void Start() {
        appDomain = new ILRuntime.Runtime.Enviorment.AppDomain();
#if UNITY_EDITOR
        appDomain.UnityMainThreadID = System.Threading.Thread.CurrentThread.ManagedThreadId;
#endif
        // 调用资源管理器加载这两个程序集：HotFix.dll + HotFix.pdb
        TextAsset dllAsset = ResourceConstant.Loader.LoadAsset<TextAsset>("HotFix.dll", "HotFix.dll");  // 同步加载
        var msDll = new System.IO.MemoryStream(dllAsset.bytes); // 这里涉及到内存管理吗？以前我不曾涉及，现在长大了，应该把它理解
        if (GameApplication.Instance.usePDB) {
            ResourceConstant.Loader.LoadAssetAsyn<TextAsset>("HotFix.pdb", "HotFix.pdb", (pdbAsset) => {  // 异步加载
                var msPdb = new System.IO.MemoryStream(pdbAsset.bytes);
                appDomain.LoadAssembly(msDll, msPdb, new Mono.Cecil.Mdb.MdbReaderProvider());
                StartApplication();  // <<<<<<<<<<<<<<<<<<<
            }, EAssetBundleUnloadLevel.ChangeSceneOver);
        } else {
            appDomain.LoadAssembly(msDll, null, new Mono.Cecil.Mdb.MdbReaderProvider());
            StartApplication();
        }
    }
    void StartApplication() {
        InitializeILRunTimeHotFixSetting();
        DoStaticMethod("HotFix.HotFixMain", "Start");
    }
    void InitializeILRunTimeHotFixSetting() {
        InitializeDelegateSetting(); // 方法被我搬到了文件的最后，太长比较简单
        InitializeCLRBindSetting();
        InitializeAdapterSetting();
```

```
        InitializeValueTypeSetting();
    }
    unsafe void InitializeCLRBindSetting() {
        foreach (var i in typeof(System.Activator).GetMethods()) {
            //  找到名字为 CreateInstance, 并且是泛型方法的方法定义  ?
            if (i.Name == "CreateInstance" && i.IsGenericMethodDefinition)
                appDomain.RegisterCLRMethodRedirection(i, CreateInstance);  // 方法重定向？再理解消化一下，不太懂还
        }
    }

    void InitializeAdapterSetting() {
        appDomain.RegisterCrossBindingAdaptor(new ViewModelBaseAdapter());
        appDomain.RegisterCrossBindingAdaptor(new UnityGuiViewAdapter());
        appDomain.RegisterCrossBindingAdaptor(new ModuleBaseAdapter());
        appDomain.RegisterCrossBindingAdaptor(new IEnumeratorObjectAdaptor());
        appDomain.RegisterCrossBindingAdaptor(new MonoBehaviourAdapter()); // <<<<<<<<<<<<<<<<<<< 临时补了一下，也有了
        appDomain.RegisterCrossBindingAdaptor(new InterfaceCrossBindingAdaptor());
    }
    void InitializeValueTypeSetting() {
        appDomain.RegisterValueTypeBinder(typeof(Vector3), new Vector3Binder());
        appDomain.RegisterValueTypeBinder(typeof(Vector2), new Vector2Binder());
        appDomain.RegisterValueTypeBinder(typeof(Quaternion), new QuaternionBinder());
    }

// 定义如何调用热更新程序集里的静态入口方法
    object DoStaticMethod(string type, string method) {
        var hotfixType = appDomain.GetType(type);
        var staticMethod = hotfixType.GetMethod(method, 0);
        return appDomain.Invoke(staticMethod, null, null);
    }

// 定义热更新的两个方法的特用实现
#region Override
    public Type LoadType(string typeName) {
        if (appDomain.LoadedTypes.ContainsKey(typeName)) {
            return appDomain.LoadedTypes[typeName].ReflectionType;
        }
        return null;
    }
    public object CreateInstance(string typeName) {
        ILType type = (ILType)appDomain.LoadedTypes[typeName];
        var instance = type.Instantiate();
        return instance;
    }
#endregion

    // 公用静态方法的实现
    public unsafe static StackObject* CreateInstance(ILIntepreter intp, StackObject* esp, IList<object> mStack, CLRMethod m
        // 获取泛型参数 <T> 的实际类型
        IType[] genericArguments = method.GenericArguments;
        if (genericArguments != null && genericArguments.Length == 1) {
            var t = genericArguments[0];
            if (t is ILType)// 如果 T 是热更 DLL 里的类型 {
                // 通过 ILRuntime 的接口来创建实例
                return ILIntepreter.PushObject(esp, mStack, ((ILType)t).Instantiate());
        } else
            return ILIntepreter.PushObject(esp, mStack, Activator.CreateInstance(t.TypeForCLR));// 通过系统反射接口创建实例
    } else
          throw new EntryPointNotFoundException();
}

// 不难猜测理解：所有需要热更新的方法类型（以不同的参数类型以及返回类型作区分），都要在这里向程序集的代理管理器注册相关方法回调代理
    void InitializeDelegateSetting() {
        appDomain.DelegateManager.RegisterMethodDelegate<int>();
        appDomain.DelegateManager.RegisterFunctionDelegate<int, string>();
        appDomain.DelegateManager.RegisterMethodDelegate<string>();
        appDomain.DelegateManager.RegisterMethodDelegate<int, int>();
        appDomain.DelegateManager.RegisterMethodDelegate<List<int>, List<int>>();
        appDomain.DelegateManager.RegisterMethodDelegate<string, string>();
        appDomain.DelegateManager.RegisterMethodDelegate<object, MessageArgs<object>>();
        appDomain.DelegateManager.RegisterMethodDelegate<object, MessageArgs<ILTypeInstance>>();
        appDomain.DelegateManager.RegisterMethodDelegate<GameObject>();
        appDomain.DelegateManager.RegisterMethodDelegate<UnityEngine.Networking.UnityWebRequest>();
        appDomain.DelegateManager.RegisterMethodDelegate<TMP_FontAsset>();
        appDomain.DelegateManager.RegisterMethodDelegate<Font>();
        appDomain.DelegateManager.RegisterMethodDelegate<AnimationClip>();
        appDomain.DelegateManager.RegisterMethodDelegate<AnimatorOverrideController>();
        appDomain.DelegateManager.RegisterMethodDelegate<RuntimeAnimatorController>();
        appDomain.DelegateManager.RegisterMethodDelegate<AudioClip>();
```

```
appDomain.DelegateManager.RegisterMethodDelegate<Material>();
appDomain.DelegateManager.RegisterMethodDelegate<TextAsset>();
appDomain.DelegateManager.RegisterMethodDelegate<Sprite>();
appDomain.DelegateManager.RegisterMethodDelegate<Texture2D>();
appDomain.DelegateManager.RegisterMethodDelegate<TapGesture>();
appDomain.DelegateManager.RegisterMethodDelegate<LongPressGesture>();
appDomain.DelegateManager.RegisterMethodDelegate<DragGesture>();
appDomain.DelegateManager.RegisterMethodDelegate<PinchGesture>();
appDomain.DelegateManager.RegisterMethodDelegate<Exception>();
appDomain.DelegateManager.RegisterFunctionDelegate<GameObject, GameObject>();
appDomain.DelegateManager.RegisterFunctionDelegate<ILTypeInstance, ILTypeInstance, int>();
// 在 Unity 的程序集与热更新的程序集跨域之间，这里给出了方法代理转换的实现逻辑，几座小桥流水人家，爱表哥爱生活
appDomain.DelegateManager.RegisterDelegateConvertor<UnityAction>((action) => {
    return new UnityAction(() => {
        ((Action)action)();
    });
});
appDomain.DelegateManager.RegisterDelegateConvertor<UnityAction<bool>>((action) => {
    return new UnityAction<bool>((b) => {
        ((Action<bool>)action)(b);
    });
});
appDomain.DelegateManager.RegisterDelegateConvertor<UnityAction<int>>((action) => {
    return new UnityAction<int>((b) => {
        ((Action<int>)action)(b);
    });
});
appDomain.DelegateManager.RegisterDelegateConvertor<UnityAction<long>>((action) => {
    return new UnityAction<long>((b) => {
        ((Action<long>)action)(b);
    });
});
appDomain.DelegateManager.RegisterDelegateConvertor<UnityAction<float>>((action) => {
    return new UnityAction<float>((b) => {
        ((Action<float>)action)(b);
    });
});
appDomain.DelegateManager.RegisterDelegateConvertor<UnityAction<string>>((action) => {
    return new UnityAction<string>((b) => {
        ((Action<string>)action)(b);
    });
});
appDomain.DelegateManager.RegisterDelegateConvertor<UnityAction<BaseEventData>>((action) => {
    return new UnityAction<BaseEventData>((b) => {
        ((Action<BaseEventData>)action)(b);
    });
});
appDomain.DelegateManager.RegisterDelegateConvertor<GestureRecognizerTS<TapGesture>.GestureEventHandler>((action) =
    return new GestureRecognizerTS<TapGesture>.GestureEventHandler((gesture) => {
        ((Action<TapGesture>)action)(gesture);
    });
});
appDomain.DelegateManager.RegisterDelegateConvertor<GestureRecognizerTS<LongPressGesture>.GestureEventHandler>((act
    return new GestureRecognizerTS<LongPressGesture>.GestureEventHandler((gesture) => {
        ((Action<LongPressGesture>)action)(gesture);
    });
});
appDomain.DelegateManager.RegisterDelegateConvertor<GestureRecognizerTS<DragGesture>.GestureEventHandler>((action)
    return new GestureRecognizerTS<DragGesture>.GestureEventHandler((gesture) => {
        ((Action<DragGesture>)action)(gesture);
    });
});
appDomain.DelegateManager.RegisterDelegateConvertor<GestureRecognizerTS<PinchGesture>.GestureEventHandler>((action)
    return new GestureRecognizerTS<PinchGesture>.GestureEventHandler((gesture) => {
        ((Action<PinchGesture>)action)(gesture);
    });
});
#if UNITY_IPHONE
        appDomain.DelegateManager.RegisterDelegateConvertor<com.mob.FinishedRecordEvent>((action) => {
    return new com.mob.FinishedRecordEvent((ex) => {
        ((Action<Exception>)action)(ex);
    });
});
#endif
appDomain.DelegateManager.RegisterDelegateConvertor<Comparison<ILTypeInstance>>((action) => {
    return new Comparison<ILTypeInstance>((x, y) => {
        return ((Func<ILTypeInstance, ILTypeInstance, System.Int32>)action)(x, y);
    });
});
```

```
    }
}
```

5.7 HotFixReflector : SingletonMono<HotFixReflector>, IHotFixMain

```
public class HotFixReflector : SingletonMono<HotFixReflector>, IHotFixMain {
    public static Assembly assembly;

    void Start() {
        ResourceConstant.Loader.LoadAssetAsyn<TextAsset>( // 这里说是异步加载
            "HotFix.dll", "HotFix.dll",
            LoadHotFixDllSuccess,
            EAssetBundleUnloadLevel.ChangeSceneOver);
    }

    void LoadHotFixDllSuccess(TextAsset dllAsset) {
        if (GameApplication.Instance.usePDB) {
            ResourceConstant.Loader.LoadAssetAsyn<TextAsset>( // 同样是异步加载
                "HotFix.pdb",
                "HotFix.pdb", (pdbAsset) => {
                    assembly = Assembly.Load(dllAsset.bytes, pdbAsset.bytes);
                    StartApplication(); // <<<<<<<<<<
                }, EAssetBundleUnloadLevel.ChangeSceneOver);
        } else {
            assembly = AppDomain.CurrentDomain.Load(dllAsset.bytes);
            StartApplication();
        }
    }

    void StartApplication() {
        try {
            Type hotfixMainType = assembly.GetType("HotFix.HotFixMain");
            MethodInfo startMethod = hotfixMainType.GetMethod("Start");
            startMethod.Invoke(null, null);
        } catch (Exception e) {
            string errorMessage = string.Empty;
            if (e.InnerException != null)
                errorMessage = e.InnerException.Message + e.InnerException.StackTrace;
            else errorMessage = e.Message + e.StackTrace;
            DebugHelper.LogError(errorMessage, true);
        }
    }
#region Override
    public Type LoadType(string typeName) {
        Type type = assembly.GetTypes().FirstOrDefault(t => t.FullName == typeName);
        if (type == null) {
            DebugHelper.LogError(string.Format("Cant't find Class by class name:'{0}'", typeName), true);
            throw new Exception(string.Format("Cant't find Class by class name:'{0}'", typeName));
        }
        return type;
    }
    public object CreateInstance(string typeName) {
        return Activator.CreateInstance(LoadType(typeName));
    }
#endregion
}
```

5.8 IHotFixMain interface

```
public interface IHotFixMain {
    Type LoadType(string typeName);
    object CreateInstance(string typeName);
}
```

6 Framework.MVVM: Unity 中定义好的 MVVM 架构；使用 ILRuntime 定义实现了必要的跨域跨程序集适配，以及数据观察回调等

6.1 DataBinding: MVVM 具备双向数据绑定功能；这里这个框架里通过代理观察者模式来实现数据的改变通知与回调

6.1.1 BindableProperty<T> ： 泛型基类，可绑定的属性

```
public class BindableProperty<T> { // 泛型基类
    private T _value;

    public Action<T, T> OnValueChanged; // 代理模式，观察者模式？
    public T Value {
        get {
            return _value;
        }
        set {
            if (!Equals(_value, value)) {
                T old = _value;
                _value = value;
                ValueChanged(old, _value);
            }
        }
    }
    void ValueChanged(T oldValue, T newValue) {
        if (OnValueChanged != null)
            OnValueChanged(oldValue, newValue);
    }

    public override string ToString() {
        return (Value != null ? Value.ToString() : "null");
    }
}
```

6.1.2 ObservableList<T> : IList<T> ： 可被观察的链表，同样包装成泛型基类

```
public class ObservableList<T> : IList<T> { // 可被观察的链表：同样包装成泛型基类

    public Action<List<T>, List<T>> OnValueChanged;
    public Action<T> OnAdd;
    public Action<int, T> OnInsert;
    public Action<T> OnRemove;
    private List<T> _value = new List<T>(); // <<<<<<<<<<

    public List<T> Value {
        get {
            return _value;
        }
        set {
            if (!Equals(_value, value)) {
                var old = _value;
                _value = value;
                ValueChanged(old, _value);
            }
        }
    }
    void ValueChanged(List<T> oldValue, List<T> newValue) {
        if (OnValueChanged != null)
            OnValueChanged(oldValue, newValue);
    }

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

// 覆写和定义链表的相关必要方法
    public void Add(T item) {
        _value.Add(item);
        if (OnAdd != null)
```

```
            OnAdd(item);
    }
    public void Clear() {
        _value.Clear();
    }
    public bool Contains(T item) {
        return _value.Contains(item);
    }
    public void CopyTo(T[] array, int arrayIndex) {
        _value.CopyTo(array, arrayIndex);
    }
    public bool Remove(T item) {
        if (_value.Remove(item)) {
            if (OnRemove != null)
                OnRemove(item);
            return true;
        }
        return false;
    }
    public int Count {
        get {
            return _value.Count;
        }
    }
    public bool IsReadOnly {
        get;
        private set;
    }
    public int IndexOf(T item) {
        return _value.IndexOf(item);
    }
    public void Insert(int index, T item) {
        _value.Insert(index, item);
        if (OnInsert != null) {
            OnInsert(index, item);
        }
    }
    public void RemoveAt(int index) {
        _value.RemoveAt(index);
    }
    public T this[int index] {
        get {
            return _value[index];
        }
        set {
            _value[index] = value;
        }
    }
}
```

6.1.3 PropertyBinder<ViewModelBase>：ViewModel(的基类以及继承类) 的绑定辅助相关方法定义类

```
// ViewModel(的基类以及继承类) 的绑定辅助相关方法定义类
public class PropertyBinder<ViewModelBase> {

    private delegate void BindHandler(ViewModelBase viewModel); // 这里才真正意义上的代理模式吧
    private delegate void UnBindHandler(ViewModelBase viewModel);

    private readonly List<BindHandler> binders = new List<BindHandler>();
    private readonly List<UnBindHandler> unBinders = new List<UnBindHandler>();

    public void Add<TProperty>(string name, string realTypeName, Action<TProperty, TProperty> valueChangedHandler) {
        var fieldInfo = GameApplication.Instance.HotFix.LoadType(realTypeName).GetField(name, BindingFlags.Instance | Bindi
        if (fieldInfo == null)
            throw new Exception(string.Format("Unable to find bindableproperty field '{0}.{1}'", realTypeName, name));
        binders.Add(viewModel => {
            GetPropertyValue<TProperty>(name, viewModel, realTypeName, fieldInfo).OnValueChanged += valueChangedHandler;
        });
        unBinders.Add(viewModel => {
            GetPropertyValue<TProperty>(name, viewModel, realTypeName, fieldInfo).OnValueChanged -= valueChangedHandler;
        });
    }
    private BindableProperty<TProperty> GetPropertyValue<TProperty>(string name, ViewModelBase viewModel,
                                                                    string realTypeName, FieldInfo fieldInfo) {
        var value = fieldInfo.GetValue(viewModel);
        BindableProperty<TProperty> bindableProperty = value as BindableProperty<TProperty>;
```

```
        if (bindableProperty == null)
            throw new Exception(string.Format("Illegal bindableproperty field '{0}.{1}' ", realTypeName, name));
        return bindableProperty;
    }

// 这里告诉一个常识说：任何一个 ViewModel，不管是蕨类还是继承后的子类，它们都有可能有好几个视图与其绑定
    public void Bind(ViewModelBase viewModel) {
        if (viewModel != null)
            for (int i = 0; i < binders.Count; i++)
                binders[i](viewModel);
    }
// 这里告诉一个常识说：任何一个 ViewModel，不管是蕨类还是继承后的子类，它们都有可能有好几个视图，需要与其解绑定
    public void UnBind(ViewModelBase viewModel) {
        if (viewModel != null)
            for (int i = 0; i < unBinders.Count; i++)
                unBinders[i](viewModel);
    }
}
```

6.2 Message: 自定义的消息机制吗？这里还没能联系上下文，完全不知道这里是在做什么？

6.2.1 MessageArgs<T>：跨域跨程序集的消息参数泛型类？

```
public class MessageArgs<T> {
    public T Item {
        get;
        private set;
    }
    public MessageArgs(T item) {
        Item = item;
    }
}
```

6.2.2 MessageAggregator<T>：Singleton<MessageAggregator<T>>：单例模式的泛型管理类？用一个字典来管理消息？

```
// 单例模式的泛型管理类？用一个字典来管理消息（可观察的属性、链表、ViewModel 等，以及它们各自对应的回调）？这里要再看不遍，还没有真正
public class MessageAggregator<T> : Singleton<MessageAggregator<T>> {

    private readonly Dictionary<string, Action<object, MessageArgs<T>>> messages =
        new Dictionary<string, Action<object, MessageArgs<T>>>();

    public void Subscribe(string name, Action<object, MessageArgs<T>> handler) {
        if (!messages.ContainsKey(name))
            messages.Add(name, handler);
        else
            messages[name] += handler;
    }
    public void Publish(string name, object sender, MessageArgs<T> args) {
        if (messages.ContainsKey(name) && messages[name] != null)
            messages[name](sender, args);
    }
}
```

6.3 Module: 这里是模块级别的定义；在模块层面级别上；在 ViewModel 层面级别上等不面层面上的跨域适配定义与实现

- 不能像最开始一样把这里误当作 Model；这是一个 framework, 数据是定义在各个不同的具体应用里

6.3.1 ModuleBase.cs

```
public abstract class ModuleBase {
    public abstract void OnInitialize();
    public abstract void Excute();
}
```

6.3.2 ModuleBaseAdapter : CrossBindingAdaptor：在模块层面上自定义实现了：模块层面级别的跨域跨程序集适配器

```
public class ModuleBaseAdapter : CrossBindingAdaptor { // 继承了 CrossBindingAdaptor 抽象基类

//  抽象蕨类里的三个抽象方法：需要实现
    public override Type BaseCLRType { // 继承了 CrossBindingAdaptor 抽象基类，就应该需要覆写里面定义过的相关方法，改造成自己需要
        get {
            return typeof(ModuleBase);
        }
    }
    public override Type AdaptorType {
        get {
            return typeof(ModuleBaseAdaptor);
        }
    }
    public override object CreateCLRInstance(ILRuntime.Runtime.Enviorment.AppDomain appdomain, ILTypeInstance instance) {
        return new ModuleBaseAdaptor(appdomain, instance);
    }

    // ModuleBase: Framework.MVVM 里定义的基类; CrossBindingAdaptorType 是 ILRuntime.Runtime.Environment 里定义的公共接口类 in
    class ModuleBaseAdaptor : ModuleBase, CrossBindingAdaptorType { // 好久没有写 cs 代码了，这里看得昏昏乎乎，类里定义类，还是

        ILTypeInstance instance;
        ILRuntime.Runtime.Enviorment.AppDomain appdomain;

// 实现对   ModuleBase 里的两个抽象方法的调控掌握
        IMethod _onInitialize;
        bool _onInitializeGot;
        IMethod _excute;
        bool _excuteGot;

        public ModuleBaseAdaptor() {  }
        public ModuleBaseAdaptor(ILRuntime.Runtime.Enviorment.AppDomain appdomain, ILTypeInstance instance) {
            this.appdomain = appdomain;
            this.instance = instance;
        }

        public ILTypeInstance ILInstance { get { return instance; } } // 返回类型的实体类

// 覆写 ModuleBase 里的两个抽象方法
        public override void OnInitialize() {
            if (!_onInitializeGot) {
                _onInitialize = instance.Type.GetMethod("OnInitialize");
                _onInitializeGot = true;
            }
            if (_onInitialize != null) {
                appdomain.Invoke(_onInitialize, instance, null);
            }
        }
        public override void Excute() {
            if (!_excuteGot) {
                _excute = instance.Type.GetMethod("Excute");
                _excuteGot = true;
            }
            if (_excute != null) {
                appdomain.Invoke(_excute, instance, null);
            }
        }
    }
}
```

6.4 View

6.4.1 IView<ViewModelBase>

```
public interface IView<ViewModelBase> {
    ViewModelBase BindingContext {
        get;
        set;
    }
    void Reveal(bool immediate = false, Action action = null);
    void Hide(bool immediate = false, Action action = null);
}
```

6.4.2 UnityGuiView: IView<ViewModelBase>, 实现或是覆写基类以及泛型里的各种定义过的方法

```
// 继承自抽象基类: 便需要实现公用接口类里面所定义的三个接口方法
// 泛型类型是 ViewModelBase, 便可以实现或是覆写里面定义的各种公用、抽象或是 protected 方法
public abstract class UnityGuiView : IView<ViewModelBase> {  //  仍然是抽象基类: 这个类比较重要, 明天早上再看一下

private bool _isInitialized; // ViewModelBase 里同共有的

// 自己再定义的两个方法, 供实现
    public virtual bool DestoryOnHide {
        get {
            return false;
        }
    }
    public virtual bool IsRoot {
        get {
            return false;
        }
    }

    public static Action SetDownRootIndex;
    public Action CloseOtherRootView;
    protected readonly PropertyBinder<ViewModelBase> binder = new PropertyBinder<ViewModelBase>();
    public readonly BindableProperty<ViewModelBase> viewModelProperty = new BindableProperty<ViewModelBase>();

// 实现了抽象接口类, 便需要实现里面的所有定义过的接口方法: 公用接口类里定义了这三个方法
    public Action RevealedAction {
        get;
        set;
    }
    public Action HiddenAction {
        get;
        set;
    }
    public ViewModelBase BindingContext { // 实现了抽象接口类, 便需要实现里面的所有定义过的接口方法: 公用接口类里定义了这三个方法
        get {
            return viewModelProperty.Value;
        }
        set {
            if (!_isInitialized) {
                OnInitialize();
                _isInitialized = true;
            }
            viewModelProperty.Value = value;
        }
    }
    protected virtual void OnInitialize() {  // 辅助帮助抽象方法, 可以随每个视图里的不同需求再具体定义
        GameObject = ResourceConstant.Loader.LoadClone(BundleName, AssetName, EAssetBundleUnloadLevel.Never);
        GameObject.AddComponent<CanvasGroup>();
        Transform.SetParent(GameObject.Find("ViewRoot").transform, false);
        viewModelProperty.OnValueChanged += OnBindingContextChanged;
    }
// <<<<<<<<<<<<<<<<<<<< 此公用方法关联四个方法: 代理模式的 RevealedAction + OnAppear() +  OnReveal() + OnRevealed()
// OnAppear() + OnRevealed(): 两个为公用方法, 提供给子视图来继承覆写
    public void Reveal(bool immediate = true, Action action = null) {
        if (action != null)
            RevealedAction += action;
        OnAppear();
        OnReveal(immediate);
        OnRevealed();
    }
    public void Hide(bool immediate = true, Action action = null) {  // <<<<<<<<<<<<<<<<<<<<
            if (action != null)
                HiddenAction += action;
        OnHide(immediate);
        OnHidden();
        OnDisappear();
    }
    public virtual void OnAppear() {
        GameObject.SetActive(true);
    }
    private void OnReveal(bool immediate) {
        BindingContext.OnStartReveal();
        if (immediate) {
            Transform.localScale = Vector3.one;
            CanvasGroup.alpha = 1;
        } else
```

```
            StartAnimatedReveal(); // <<<<<<<<<<
        }
        public virtual void OnRevealed() {
            BindingContext.OnFinishReveal();
            if (RevealedAction != null)
                RevealedAction();
            if (IsRoot) {
                if (CloseOtherRootView != null)
                    CloseOtherRootView();
            }
            if (SetDownRootIndex != null)
                SetDownRootIndex();
        }
        private void OnHide(bool immediate) {
            BindingContext.OnStartHide();
            if (immediate) {
                Transform.localScale = Vector3.zero;
                CanvasGroup.alpha = 0;
            } else
                StartAnimatedHide();
        }
        public virtual void OnHidden() {
            if (HiddenAction != null)
                HiddenAction();
        }
        public virtual void OnDisappear() {
            GameObject.SetActive(false);
            BindingContext.OnFinishHide();
            if (DestoryOnHide)
                UnityEngine.Object.Destroy(GameObject);
        }
    // OnInitialize() + OnDestory(): ViewModelBase 里定义的抽象方法实现, 实现必要的基类逻辑
        public virtual void OnDestory() {
            if (BindingContext.IsRevealed)
                Hide(true);
            BindingContext.OnDestory();
            BindingContext = null;
            viewModelProperty.OnValueChanged = null;
        }

    // 对于视图中需要使用动画的情况: 作出了考虑, 定义了可以调用的方法
        protected virtual void StartAnimatedReveal() {
            CanvasGroup.interactable = false;
            Transform.localScale = Vector3.one;
            //huandong
                 //CanvasGroup.DOFade(1, 0.2f).SetDelay(0.2f).OnComplete(() =>
                 //{
                 //    canvasGroup.interactable = true;
                 //});
                 }
        protected virtual void StartAnimatedHide() {
            CanvasGroup.interactable = false;
            //canvasGroup.DOFade(0, 0.2f).SetDelay(0.2f).OnComplete(() =>
            //{
            //    transform.localScale = Vector3.zero;
            //    canvasGroup.interactable = true;
            //});
        }

    // 当有用户行为等导致视图变更的时候, 需要调用的对所绑定的 ViewModel 的变更
        protected virtual void OnBindingContextChanged(ViewModelBase oldValue, ViewModelBase newValue) {
            binder.UnBind(oldValue);
            binder.Bind(newValue);
        }

        // 主要是针对热更新 AB(AssetBundle) 包的处理的相关函数的定义
        public virtual string BundleName {
            get {
                return string.Empty;
            }
        }
        public virtual string AssetName {
            get {
                return string.Empty;
            }
        }
        public virtual string ViewName {
            get {
                return string.Empty;
```

```
        }
    }
    public virtual string ViewModelTypeName {
        get {
            return string.Empty;
        }
    }
    public GameObject GameObject {
        get;
        set;
    }
    private Transform _transform;
    public Transform Transform {
        get {
            if (_transform == null) {
                _transform = GameObject.transform;
            }
            return _transform;
        }
    }
    private CanvasGroup _canvasGroup;
    public CanvasGroup CanvasGroup {
        get {
            if (_canvasGroup == null)
                _canvasGroup = GameObject.GetComponent<CanvasGroup>();
            return _canvasGroup;
        }
    }
}
```

6.4.3 UnityGuiViewAdapter : CrossBindingAdapto: 最鼻祖的实体基类

```
public class UnityGuiViewAdapter : CrossBindingAdaptor { // 最鼻祖的实体基类

    // 这里是 ModuleBaseAdapter 里提供的三个接口方法: 这里想一想, 为什么要实现 ModuleBaseAdapter 里所定义的三个方法呢, 为什么需要
    public override Type BaseCLRType {
        get {
            return typeof(UnityGuiView);
        }
    }
    public override Type AdaptorType {
        get {
            return typeof(UnityGuiViewAdaptor);
        }
    }
    public override object CreateCLRInstance(ILRuntime.Runtime.Enviorment.AppDomain appdomain, ILTypeInstance instance) {
        return new UnityGuiViewAdaptor(appdomain, instance);
    }

    class UnityGuiViewAdaptor : UnityGuiView, CrossBindingAdaptorType { // ILRuntime.Enviorment.CrossBindingAdaptorType
        ILTypeInstance instance;
        ILRuntime.Runtime.Enviorment.AppDomain appdomain;
        object[] param2 = new object[2];

        public UnityGuiViewAdaptor() { }
        public UnityGuiViewAdaptor(ILRuntime.Runtime.Enviorment.AppDomain appdomain, ILTypeInstance instance) {
            this.appdomain = appdomain;
            this.instance = instance;
        }
        public ILTypeInstance ILInstance {
            get { return instance; }
        }

// UnityGuiView 里所定义的所有公用方法的基类实现: 因为后来的继承类可以覆写, 但是也可以要求就请按照基类的实现去执行
        protected override void OnInitialize() { // 辅助帮助抽象方法, 可以随每个视图里的不同需求再具体定义
            if (!_onInitializeGot) {
                _onInitialize = instance.Type.GetMethod("OnInitialize");
                _onInitializeGot = true;
            }
            if (_onInitialize != null && !isOnInitializeInvoking) {
                isOnInitializeInvoking = true;
                appdomain.Invoke(_onInitialize, instance);
                isOnInitializeInvoking = false;
            } else
                base.OnInitialize();
        }
        public override void OnAppear() {
```

```
    if (!_onAppearGot) {
        _onAppear = instance.Type.GetMethod("OnAppear");
        _onAppearGot = true;
    }
    if (_onAppear != null && !isOnAppearInvoking) {
        isOnAppearInvoking = true;
        appdomain.Invoke(_onAppear, instance);
        isOnAppearInvoking = false;
    } else {
        base.OnAppear();
    }
}
public override void OnRevealed() {
    if (!_onRevealedGot) {
        _onRevealed = instance.Type.GetMethod("OnRevealed");
        _onRevealedGot = true;
    }
    if (_onRevealed != null && !isOnRevealedInvoking) {
        isOnRevealedInvoking = true;
        appdomain.Invoke(_onRevealed, instance);
        isOnRevealedInvoking = false;
    } else {
        base.OnRevealed();
    }
}
public override void OnHidden() {
    if (!_onHiddenGot) {
        _onHidden = instance.Type.GetMethod("OnHidden");
        _onHiddenGot = true;
    }
    if (_onHidden != null && !isOnHiddenInvoking) {
        isOnHiddenInvoking = true;
        appdomain.Invoke(_onHidden, instance);
        isOnHiddenInvoking = false;
    } else {
        base.OnHidden();
    }
}
public override void OnDisappear() {
    if (!_onDisappearGot) {
        _onDisappear = instance.Type.GetMethod("OnDisappear");
        _onDisappearGot = true;
    }
    if (_onDisappear != null && !isOnDisappearInvoking) {
        isOnDisappearInvoking = true;
        appdomain.Invoke(_onDisappear, instance);
        isOnDisappearInvoking = false;
    } else {
        base.OnDisappear();
    }
}
public override void OnDestory() {
    if (!_onDestoryGot) {
        _onDestory = instance.Type.GetMethod("OnDestory");
        _onDestoryGot = true;
    }
    if (_onDestory != null && !isOnDestoryInvoking) {
        isOnDestoryInvoking = true;
        appdomain.Invoke(_onDestory, instance);
        isOnDestoryInvoking = false;
    } else {
        base.OnDestory();
    }
}
protected override void StartAnimatedReveal() {
    if (!_startAnimatedRevealGot) {
        _startAnimatedReveal = instance.Type.GetMethod("StartAnimatedReveal");
        _startAnimatedRevealGot = true;
    }
    if (_startAnimatedReveal != null && !isStartAnimatedRevealInvoking) {
        isStartAnimatedRevealInvoking = true;
        appdomain.Invoke(_startAnimatedReveal, instance);
        isStartAnimatedRevealInvoking = false;
    } else {
        base.StartAnimatedReveal();
    }
}
protected override void StartAnimatedHide() {
    if (!_startAnimatedHideGot) {
```

```
            _startAnimatedHide = instance.Type.GetMethod("StartAnimatedHide");
            _startAnimatedHideGot = true;
        }
        if (_startAnimatedHide != null && !isStartAnimatedHideInvoking) {
            isStartAnimatedHideInvoking = true;
            appdomain.Invoke(_startAnimatedHide, instance);
            isStartAnimatedHideInvoking = false;
        } else {
            base.StartAnimatedHide();
        }
    }
    protected override void OnBindingContextChanged(ViewModelBase oldValue, ViewModelBase newValue) {
        if (!_onBindingContextChangedGot) {
            _onBindingContextChanged = instance.Type.GetMethod("OnBindingContextChanged");
            _onBindingContextChangedGot = true;
        }
        if (_onBindingContextChanged != null && !isOnBindingContextChangedInvoking) {
            isOnBindingContextChangedInvoking = true;
            appdomain.Invoke(_onBindingContextChanged, instance, param2);
            isOnBindingContextChangedInvoking = false;
        } else {
            base.OnBindingContextChanged(oldValue, newValue);
        }
    }

// 下面是处理热更新 AB 包相关的回调接口控制公用方法: 定义为基类实现, 因为此类为第一个实体的基类
    public override string BundleName {
        get {
            if (!_getBundleNameGot) {
                _getBundleName = instance.Type.GetMethod("get_BundleName", 0);
                _getBundleNameGot = true;
            }
            if (_getBundleName != null && !isGetBundleNameInvoking) {
                isGetBundleNameInvoking = true;
                var res = (string)appdomain.Invoke(_getBundleName, instance, null);
                isGetBundleNameInvoking = false;
                return res;
            } else {
                return base.BundleName;
            }
        }
    }
    public override string AssetName {
        get {
            if (!_getAssetNameGot) {
                _getAssetName = instance.Type.GetMethod("get_AssetName", 0);
                _getAssetNameGot = true;
            }
            if (_getAssetName != null && !isGetAssetNameInvoking) {
                isGetAssetNameInvoking = true;
                var res = (string)appdomain.Invoke(_getAssetName, instance, null);
                isGetAssetNameInvoking = false;
                return res;
            } else {
                return base.AssetName;
            }
        }
    }
    public override string ViewName {
        get {
            if (!_getViewNameGot) {
                _getViewName = instance.Type.GetMethod("get_ViewName", 0);
                _getViewNameGot = true;
            }
            if (_getViewName != null && !isGetViewNameInvoking) {
                isGetViewNameInvoking = true;
                var res = (string)appdomain.Invoke(_getViewName, instance, null);
                isGetViewNameInvoking = false;
                return res;
            } else {
                return base.ViewName;
            }
        }
    }
    public override string ViewModelTypeName {
        get {
            if (!_getViewModelTypeNameGot) {
                _getViewModelTypeName = instance.Type.GetMethod("get_ViewModelTypeName", 0);
                _getViewModelTypeNameGot = true;
```

```
                }
                if (_getViewModelTypeName != null && !isGetViewModelTypeNameInvoking) {
                    isGetViewModelTypeNameInvoking = true;
                    var res = (string)appdomain.Invoke(_getViewModelTypeName, instance, null);
                    isGetViewModelTypeNameInvoking = false;
                    return res;
                } else {
                    return base.ViewModelTypeName;
                }
            }
        }

// 覆写 UnityGuiView 里定义的两个公用抽象方法
        public override bool DestoryOnHide {
            get {
                if (!_getDestoryOnHideGot) {
                    _getDestoryOnHide = instance.Type.GetMethod("get_DestoryOnHide", 0);
                    _getDestoryOnHideGot = true;
                }
                if (_getDestoryOnHide != null && !isGetDestoryOnHideInvoking) {
                    isGetDestoryOnHideInvoking = true;
                    var res = (bool)appdomain.Invoke(_getDestoryOnHide, instance, null);
                    isGetDestoryOnHideInvoking = false;
                    return res;
                } else {
                    return base.DestoryOnHide;
                }
            }
        }
        public override bool IsRoot {
            get {
                if (!_getIsRootGot) {
                    _getIsRoot = instance.Type.GetMethod("get_IsRoot", 0);
                    _getIsRootGot = true;
                }
                if (_getIsRoot != null && !isGetIsRootInvoking) {
                    isGetIsRootInvoking = true;
                    var res = (bool)appdomain.Invoke(_getIsRoot, instance, null);
                    isGetIsRootInvoking = false;
                    return res;
                } else {
                    return base.IsRoot;
                }
            }
        }
    }

// 每个标记变量对应的三小变量
    IMethod _onInitialize;
    bool _onInitializeGot;
    bool isOnInitializeInvoking = false;

    IMethod _onAppear;
    bool _onAppearGot;
    bool isOnAppearInvoking = false;
    IMethod _onRevealed;
    bool _onRevealedGot;
    bool isOnRevealedInvoking = false;
    IMethod _onHidden;
    bool _onHiddenGot;
    bool isOnHiddenInvoking = false;
    IMethod _onDisappear;
    bool _onDisappearGot;
    bool isOnDisappearInvoking = false;
    IMethod _onDestory;
    bool _onDestoryGot;
    bool isOnDestoryInvoking = false;
    IMethod _startAnimatedReveal;
    bool _startAnimatedRevealGot;
    bool isStartAnimatedRevealInvoking = false;
    IMethod _startAnimatedHide;
    bool _startAnimatedHideGot;
    bool isStartAnimatedHideInvoking = false;
    IMethod _getBundleName;
    bool _getBundleNameGot;
    bool isGetBundleNameInvoking = false;
    IMethod _getAssetName;
    bool _getAssetNameGot;
    bool isGetAssetNameInvoking = false;
```

```
    IMethod _getViewName;
    bool _getViewNameGot;
    bool isGetViewNameInvoking = false;
    IMethod _getDestoryOnHide;
    bool _getDestoryOnHideGot;
    bool isGetDestoryOnHideInvoking = false;
    IMethod _getIsRoot;
    bool _getIsRootGot;
    bool isGetIsRootInvoking = false;
    IMethod _getViewModelTypeName;
    bool _getViewModelTypeNameGot;
    bool isGetViewModelTypeNameInvoking = false;
    IMethod _onBindingContextChanged;
    bool _onBindingContextChangedGot;
    bool isOnBindingContextChangedInvoking = false;
}
```

6.5 ViewModel

6.5.1 ViewModelBase.cs

```
public class ViewModelBase {
    private bool _isInitialize;
    public bool IsRevealInProgress {
        get;
        private set;
    }
    public bool IsRevealed {
        get;
        private set;
    }
    public bool IsHideInProgress {
        get;
        private set;
    }
    public ViewModelBase ParentViewModel {
        get;
        set;
    }
    public virtual void OnStartReveal() {
        IsRevealInProgress = true;
        if (!_isInitialize) {
            OnInitialize();
            _isInitialize = true;
        }
    }
    public virtual void OnFinishReveal() {
        IsRevealInProgress = false;
        IsRevealed = true;
    }
    public virtual void OnStartHide() {
        IsHideInProgress = true;
    }
    public virtual void OnFinishHide() {
        IsHideInProgress = false;
        IsRevealed = false;
    }

    public virtual void OnDestory() {}
    protected virtual void OnInitialize() {}
}
```

6.5.2 ViewModelBaseAdapter : CrossBindingAdaptor：自定义实现了 ViewModel 的跨域跨程序集适配

```
// 作为两个不同程序集中的 ViewModel 的桥梁适配器：
public class ViewModelBaseAdapter : CrossBindingAdaptor { // 这里需要再想一想：为什么外面大类，里面小类，继承的基类不一样，公用与

// 实现了基类 CrossBindingAdapter 里的其中三个抽象方法
    public override Type BaseCLRType {
        get {
            return typeof(ViewModelBase);
        }
    }
    public override Type AdaptorType {
```

```
        get {
            return typeof(ViewModelBaseAdaptor);
        }
    }
    public override object CreateCLRInstance(ILRuntime.Runtime.Enviorment.AppDomain appdomain, ILTypeInstance instance) {
        return new ViewModelBaseAdaptor(appdomain, instance);
    }

// 作为两个不同程序集中的 ViewModel 的桥梁适配器: 继承自 ViewModelBase, 需要实现里而定义过的甩有 6 个抽象方法
    class ViewModelBaseAdaptor : ViewModelBase, CrossBindingAdaptorType { // <<<<<<<<<<<<<<<<<<<
        ILTypeInstance instance; // 来自于 CrossBindingAdapterType ?
        ILRuntime.Runtime.Enviorment.AppDomain appdomain;

        public ViewModelBaseAdaptor() { }
        public ViewModelBaseAdaptor(ILRuntime.Runtime.Enviorment.AppDomain appdomain, ILTypeInstance instance) {
            this.appdomain = appdomain;
            this.instance = instance;
        }

// 来自于 CrossBindingAdapterType 接口的实体实现
        public ILTypeInstance ILInstance { get { return instance; } }

// 作为两个不同程序集中的 ViewModel 的桥梁适配器: 继承自 ViewModelBase, 需要实现里而定义过的甩有 6 个抽象方法
        public override void OnStartReveal() {
            if (!_onStartRevealGot) {
                _onStartReveal = instance.Type.GetMethod("OnStartReveal");
                _onStartRevealGot = true;
            }
            if (_onStartReveal != null && !_isOnStartRevealInvoking) {
                _isOnStartRevealInvoking = true;
                appdomain.Invoke(_onStartReveal, instance);
                _isOnStartRevealInvoking = false;
            } else
                base.OnStartReveal();
        }
        public override void OnFinishReveal() {
            if (!_onFinishRevealGot) {
                _onFinishReveal = instance.Type.GetMethod("OnFinishReveal");
                _onFinishRevealGot = true;
            }
            if (_onFinishReveal != null && !_isOnFinishRevealInvoking) {
                _isOnFinishRevealInvoking = true;
                appdomain.Invoke(_onFinishReveal, instance);
                _isOnFinishRevealInvoking = false;
            } else
                base.OnFinishReveal();
        }
        public override void OnStartHide() {
            if (!_onStartHideGot) {
                _onStartHide = instance.Type.GetMethod("OnStartHide");
                _onStartHideGot = true;
            }
            if (_onStartHide != null && !_isOnStartHideInvoking) {
                _isOnStartHideInvoking = true;
                appdomain.Invoke(_onStartHide, instance);
                _isOnStartHideInvoking = false;
            } else
                base.OnStartHide();
        }
        public override void OnFinishHide() {
            if (!_onFinishHideGot) {
                _onFinishHide = instance.Type.GetMethod("OnFinishHide");
                _onFinishHideGot = true;
            }
            if (_onFinishHide != null && !_isOnFinishHideInvoking) {
                _isOnFinishHideInvoking = true;
                appdomain.Invoke(_onFinishHide, instance);
                _isOnFinishHideInvoking = false;
            } else
                base.OnFinishHide();
        }
        public override void OnDestory() {
            if (!_onDestoryGot) {
                _onDestory = instance.Type.GetMethod("OnDestory");
                _onDestoryGot = true;
            }
            if (_onDestory != null && !_isOnDestoryInvoking) {
                _isOnDestoryInvoking = true;
                appdomain.Invoke(_onDestory, instance);
```

```
                _isOnDestoryInvoking = false;
            } else
                base.OnDestory();
        }
        protected override void OnInitialize() {
            if (!_onInitializeGot) {
                _onInitialize = instance.Type.GetMethod("OnInitialize");
                _onInitializeGot = true;
            }
            if (_onInitialize != null && !_isOnInitializeInvoking) {
                _isOnInitializeInvoking = true;
                appdomain.Invoke(_onInitialize, instance);
                _isOnInitializeInvoking = false;
            } else
                base.OnInitialize();
        }
    // _onStart/_onFinish: Reveal + Hide;
        IMethod _onStartReveal;
        bool _onStartRevealGot;
        bool _isOnStartRevealInvoking = false;
        IMethod _onFinishReveal;
        bool _onFinishRevealGot;
        bool _isOnFinishRevealInvoking = false;
        IMethod _onStartHide;
        bool _onStartHideGot;
        bool _isOnStartHideInvoking = false;
        IMethod _onFinishHide;
        bool _onFinishHideGot;
        bool _isOnFinishHideInvoking = false;
    // _onInitialize + _onDestroy
        IMethod _onInitialize;
        bool _onInitializeGot;
        bool _isOnInitializeInvoking = false;
        IMethod _onDestory;
        bool _onDestoryGot;
        bool _isOnDestoryInvoking = false;
    }
}
```

7 Framework.Util 各种帮助类

7.1

7.2

7.3

7.4

7.5 Singleton<T> + SingletonMono<T> : MonoBehaviour

```
public class Singleton<T> where T : class, new() {
    protected static T _instance;
    public static T Instance {
        get {
            if (_instance == null)
                _instance = new T();
            return _instance;
        }
    }
    public static T GetInstance() {
        return Instance;
    }
}

public class SingletonMono<T> : MonoBehaviour where T : MonoBehaviour {  // MonoBehaviour 类型的泛型基类
    protected static T _instance;
    public static T Instance {
        get {
            if (_instance == null) {  // 实例化一个游戏控件 (GameObject)，再将泛型类型以元件的形式挂上去，更将控件的名字
                GameObject obj = new GameObject();
                _instance = obj.AddComponent<T>();
                obj.name = _instance.GetType().Name;
```

```
            }
            return _instance; // 返回实例
        }
    }
    public static T GetInstance() {
        return Instance;
    }
    public static void DestoryInstance() {
        if (_instance == null)
            return;
        GameObject obj = _instance.gameObject;
// 想一下：下面这一行，这里为什么会被亚掉？应该是正常运行这行才对呀，还是说控件池相关呢
        //ResourceMgr.Instance.DestroyObject(obj);
    }
}
```

8 HotFix 中使用 MVVM 架构实现热更新的搭配与相关的链接

9 ILRuntime 类库里源码的基本理解，最重要的涉及到的相关的类与方法摘要

9.1 CrossBindingAdaptor： IType 跨域（程序集）绑定适配器 + CrossBindingAdaptorType interface 公用接口类（为什么要这个公用接口类？）

```
public interface CrossBindingAdaptorType {  // 公用接口类
    ILTypeInstance ILInstance { get; }
}

// This interface is used for inheritance and implementation of CLR Types or interfaces
public abstract class CrossBindingAdaptor : IType {
    IType type;

// 下面是定义的几个公用的抽象方法，供子类实现
    // This returns the CLR type to be inherited or CLR interface to be implemented
    public abstract Type BaseCLRType { get; }
    // If this Adaptor is capable to impelement multuple interfaces, use this Property, AND BaseCLRType should return null
    public virtual Type[] BaseCLRTypes {
        get {
            return null;
        }
    }
    public abstract Type AdaptorType { get; }
    public abstract object CreateCLRInstance(Enviorment.AppDomain appdomain, ILTypeInstance instance);

    internal IType RuntimeType { get { return type; } set { type = value; } }

// 反射机制的所有可能涉及的相关的方法定义；getters/setters
#region IType Members
    public IMethod GetMethod(string name, int paramCount, bool declaredOnly = false) {
        return type.GetMethod(name, paramCount, declaredOnly);
    }
    public IMethod GetMethod(string name, List<IType> param, IType[] genericArguments, IType returnType = null, bool declar
        return type.GetMethod(name, param, genericArguments, returnType, declaredOnly);
    }
    public List<IMethod> GetMethods() {
        return type.GetMethods();
    }
    public int GetFieldIndex(object token) {
        return type.GetFieldIndex(token);
    }
    public IMethod GetConstructor(List<IType> param) {
        return type.GetConstructor(param);
    }
    public bool CanAssignTo(IType type) {
        bool res = false;
        if (BaseType != null)
            res = BaseType.CanAssignTo(type);
        var interfaces = Implements;
        if (!res && interfaces != null) {
            for (int i = 0; i < interfaces.Length; i++) {
```

```
                var im = interfaces[i];
                res = im.CanAssignTo(type);
                if (res)
                    return true;
            }
        }
        return res;
    }
    public IType MakeGenericInstance(KeyValuePair<string, IType>[] genericArguments) {
        return type.MakeGenericInstance(genericArguments);
    }
    public IType MakeByRefType() {
        return type.MakeByRefType();
    }
    public IType MakeArrayType(int rank) {
        return type.MakeArrayType(rank);
    }
    public IType FindGenericArgument(string key) {
        return type.FindGenericArgument(key);
    }
    public IType ResolveGenericType(IType contextType) {
        return type.ResolveGenericType(contextType);
    }
    public IMethod GetVirtualMethod(IMethod method) {
        return type.GetVirtualMethod(method);
    }
    public void GetValueTypeSize(out int fieldCout, out int managedCount) {
        type.GetValueTypeSize(out fieldCout, out managedCount);
    }
// Getter / Setter s
    public bool IsGenericInstance {
        get {
            return type.IsGenericInstance;
        }
    }
    public KeyValuePair<string, IType>[] GenericArguments {
        get {
            return type.GenericArguments;
        }
    }
    public Type TypeForCLR {
        get {
            return type.TypeForCLR;
        }
    }
    public IType ByRefType {
        get {
            return type.ByRefType;
        }
    }
    public IType ArrayType {
        get {
            return type.ArrayType;
        }
    }
    public string FullName {
        get {
            return type.FullName;
        }
    }
    public string Name {
        get {
            return type.Name;
        }
    }
    public bool IsValueType {
        get {
            return type.IsValueType;
        }
    }
    public bool IsPrimitive {
        get {
            return type.IsPrimitive;
        }
    }
    public bool IsEnum {
        get {
            return type.IsEnum;
        }
```

```
    }
    public bool IsDelegate {
        get {
            return type.IsDelegate;
        }
    }
    public AppDomain AppDomain {
        get {
            return type.AppDomain;
        }
    }
    public Type ReflectionType {
        get {
            return type.ReflectionType;
        }
    }
    public IType BaseType {
        get {
            return type.BaseType;
        }
    }
    public IType[] Implements {
        get {
            return type.Implements;
        }
    }
    public bool HasGenericParameter {
        get {
            return type.HasGenericParameter;
        }
    }
    public bool IsGenericParameter {
        get {
            return type.IsGenericParameter;
        }
    }
    public bool IsArray {
        get { return false; }
    }
    public bool IsByRef {
        get {
            return type.IsByRef;
        }
    }
    public bool IsInterface {
        get { return type.IsInterface; }
    }
    public IType ElementType {
        get {
            return type.ElementType;
        }
    }
    public int ArrayRank {
        get { return type.ArrayRank; }
    }
    public int TotalFieldCount {
        get {
            return type.TotalFieldCount;
        }
    }
    public StackObject DefaultObject {
        get {
            return default(StackObject);
        }
    }
    public int TypeIndex {
        get {
            return -1;
        }
    }
#endregion
}
```

9.2

9.3

9.4

9.5