

Unity Android SDK/NDK 俄罗斯方块砖 3D 小游戏

deepwaterooo

September 29, 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	5
4.5 线上项目和资料	5
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	6
5.1 Adaptor	6
5.1.1 IDisposableAdaptor : CrossBindingAdaptor	6
5.1.2 IEnumeratorObjectAdaptor : CrossBindingAdaptor { // 叠代器适配器? 不是集合元素里轮询的叠代器, 而是协程	6
5.1.3 InterfaceCrossBindingAdaptor : CrossBindingAdaptor: 就是实现基类里的三个方法呀	7
5.1.4 MonoBehaviourAdapter : CrossBindingAdaptor: ./Plugins/ILRuntime/Adapters 包裹里的	7
5.2 Factory: 顾名思义, 就是工厂模式, 负责生产各种不同类型的实例 (生产人质)	8
5.2.1 enum FactoryType	8
5.2.2 interface IObjectFactory	9
5.2.3 PoolObjectFactory : Singleton<PoolObjectFactory>, IObjectFactory: 单例对象池: 结合工厂模式使用	9
5.2.4 SingletonObjectFactory : IObjectFactory: 单例对象工厂	10
5.2.5 TransientObjectFactory : IObjectFactory	10
5.3 Inject: 每个 (自动) 注入的背后, 都是强大优化过的逻辑在支撑! 爱表哥, 爱生活	10
5.3.1 ServiceLocatorContain	10
5.4 Wrap	11
5.4.1 BehaviourWrap : MonoBehaviour	11
5.5 GameApplication: 游戏入口类	12
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	13
5.7 HotFixReflector : SingletonMono<HotFixReflector>, IHotFixMain	16
5.8 IHotFixMain interface	17

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	17
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	17
6.1.1 BindableProperty<T> : 泛型基类，可绑定的属性	17
6.1.2 ObservableList<T> : IList<T> : 可被观察的链表，同样包装成泛型基类	17
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	19
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？	19
6.2.1 MessageArgs<T>: (跨域跨程序集的) 消息参数泛型类？是泛型类，做的事情就是对泛型基类逻辑的必要封装	20
6.2.2 MessageAggregator<T> : Singleton<MessageAggregator<T>>: 单例模式的泛型集合管理类，用一个字典来管理消息	20
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	21
6.3.1 ModuleBase.cs	21
6.3.2 ModuleBaseAdapter : CrossBindingAdapter: 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	21
6.4 View	22
6.4.1 IView<ViewModelBase>: 这个特型（自创的用语，相对于泛型来说）的接口，实现了个性化 MVVM 设计 View 与 ViewModel 的自动绑定	22
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	22
6.4.3 UnityGuiViewAdapter : CrossBindingAdapto: 最鼻祖的实体基类	25
6.5 ViewModel	28
6.5.1 ViewModelBase.cs	28
6.5.2 ViewModelBaseAdapter : CrossBindingAdapter: 自定义实现了 ViewModel 的跨域跨程序集适配	29
7 ResourceManager 包裹 ResMgr	31
7.1 AssetLoader: 资源加载器	31
7.1.1 abstract class AssetLoader: Asset 加载封装抽象类	31
7.1.2 GameObjectLoader : AssetLoader: GameObjectAsset 加载类	32
7.1.3 SceneLoader : AssetLoader: 场景 Asset 加载类	33
7.2 AssetBundleSpec: 热更新资源包的相关参数配置与读取，资源包的加载等	33
7.3 interface IResourceLoader: 不同类型的资源加载公用接口类：定义的是对于不同类型的资源的加载调用公用 API 接口	39
7.4 ResourceConstant; 静态常量类，分几个不同平台来处理平台相关的资源获取存储等相关操作	40
7.5 ResourceMap : MonoBehaviour, IResourceLoader: 是对前 IResourceLoader 公用接口类的实现	42
8 SDK: 第三方库的接入，比如想要使用微信登录，或是使用了 google 账户登录等	50
8.0.1 NativeHelper: 原生交互类	50
9 Framework.Util 各种帮助类	50
9.1 Singleton<T> + SingletonMono<T> : MonoBehaviour	50
9.2 static class JsonSerializer: 静态帮助类，帮助将网络下载的资源包解析为相应的封装类型	51

1 模块搭建

- # only include two levels in TOC

- ILRuntime 的消理解，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，了解当时设计的优缺点，由此改进更将

3 环境弄得比较好的包括：

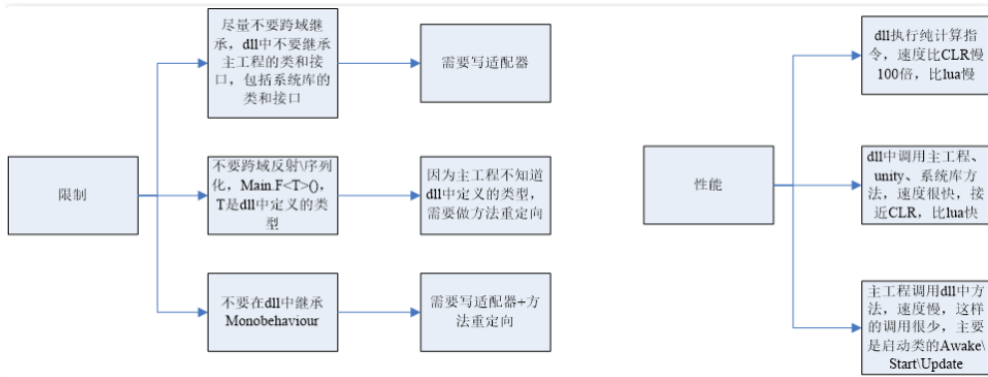
- 小笔记本电脑太慢了，会回家再读其它模块的源码，理解透彻。爱表哥，爱生活!!
- 输入法的搭建：终于用到了自己之前用过的好用的输入法
- 这两天开车疲累，最迟明天中午会去南湾找房间出租，尽快解决搬家的问题；昨天晚上回来得太晚了，一路辛苦，路上只差睡着，回到家里补觉补了好多个小时。
- 小电脑，笔记本电脑里的游戏环境搭建，今天下午去图书馆里弄（今天下午去图书馆里把需要借助快速网络来完成的事情都搭建好；家里被恶房东故意整了个腾腾慢的网，故意阻碍别人的发展，谁还愿意再这样的环境中继续住下去呢?!!!)
- 能够把程序源码读得比较懂，也并不代表把所有相关的原理就全部弄懂了；不是说还有多在的挑战，而是说要不断寻找更为有效的学习方法，快速掌握所有涉及到的相关原理；在理解得更为深入掌握了基本原理的基础上再去读源码，会不会更为有效事半功倍呢？这是一颗永远不屈服的心，爱表哥，爱生活!!!

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 ILInterpreter.cs，通过 Opcode 来逐语句执行机器码，解释器的代码有四千多行。

4.3 ILRuntime 主要限制



- **委托适配器 (DelegateAdapter)**: 将委托实例传出给 ILRuntime 外部使用, 将其转换成 CLR 委托实例。

由于 IL2CPP 之类的 AOT 编译技术无法在运行时生成新的类型, 所以在创建委托实例的时候 ILRuntime 选择了显式注册的方式, 以保证问题不被隐藏到上线后才发现。

```
// 同一参数组合只需要注册一次
delegate void SomeDelegate(int a, float b);
Action<int, float> act;
// 注册, 不带返回值, 最多支持五个参数传入
appDomain.DelegateManager.RegisterMethodDelegate<int, float>();

// 注册, 带参数返回值, 最后一个参数为返回值, 最多支持四个参数传入
delegate bool SomeFunction(int a, float b);
Func<int, float, bool> act;
```

- **委托转换器 RegisterDelegateConverter**: 需要将一个不是 Action 或者 Func 类型的委托实例传到 ILRuntime 外部使用, 需要写委托适配器和委托转换器。委托转换器将 Action 和 Func 转换为你真正需要的那个委托类型

```
app.DelegateManager.RegisterDelegateConverter<SomeFunction>((action) =>
{
    return new SomeFunction((a, b) =>
    {
        return ((Func<int, float, bool>)action)(a, b);
    });
});
```

- 为了避免不必要的麻烦, 以及后期热更出现问题, 建议: 1、尽量避免不必要的跨域委托调用 2、尽量使用 Action 以及 Func 委托类型
- **CLR 重定向**: ILRuntime 为了解决外部调用内部接口的问题, 引入了 CLR 重定向机制。原理就是当 IL 解释器发现需要调用某个指定 CLR 方法时, 将实际调用重定向到另外一个方法进行挟持, 再在这个方法中对 ILRuntime 的反射的用法进行处理
- 从代码中可以看出重定向的工作是把方法挟持下来后装到 ILIntepreter 的解释器里面实例化
- 不带返回值的重定向:

```
public static StackObject* CreateInstance(ILIntepreter intp, StackObject* esp,
                                          List<object> mStack, CLRMethod method, bool isNewObj) {
    // 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 // 通过系统反射接口创建实例
            return ILIntepreter.PushObject(esp, mStack, Activator.CreateInstance(t.TypeForCLR));
    }
```

```

    } else
        throw new EntryPointNotFoundException();
}
// 注册
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) {
        // RegisterCLRMethodRedirection: 通过 redirectMap 存储键值对 MethodBase-CLRRedirectionDelegate, 如果 i 不为空且 redirectMap 不为空
        appdomain.RegisterCLRMethodRedirection(i, CreateInstance);
    }
}
}

```

- 带返回值方法的重定向

```

public unsafe static StackObject* DLog(ILInteprerter __intp, StackObject* __esp,
                                       List<object> __mStack, CLRMethod __method, bool isNewObj) {
    ILRuntime.Runtime.Environment.AppDomain __domain = __intp.AppDomain;
    StackObject* ptr_of_this_method;
    // 只有一个参数, 所以返回指针就是当前栈指针 ESP - 1
    StackObject* __ret = ILInteprerter.Minus(__esp, 1);
    // 第一个参数为 ESP - 1, 第二个参数为 ESP - 2, 以此类推
    ptr_of_this_method = ILInteprerter.Minus(__esp, 1);
    // 获取参数 message 的值
    object message = StackObject.ToObject(ptr_of_this_method, __domain, __mStack);
    // 需要清理堆栈
    __intp.Free(ptr_of_this_method);
    // 如果参数类型是基础类型, 例如 int, 可以直接通过 int param = ptr_of_this_method->Value 获取值,
    // 关于具体原理和其他基础类型如何获取, 请参考 ILRuntime 实现原理的文档。

    // 通过 ILRuntime 的 Debug 接口获取调用热更 DLL 的堆栈
    string stackTrace = __domain.DebugService.GetStackTrace(__intp);
    Debug.Log(string.Format("{0}\n{1}", format, stackTrace));
    return __ret;
}

```

- **LitJson 集成:** Json 序列化是开发中非常经常需要用到的功能, 考虑到其通用性, 因此 ILRuntime 对 LitJson 这个序列化库进行了集成

```

//对 LitJson 进行注册, 需要在注册 CLR 绑定之前
LitJson.JsonMapper.RegisterILRuntimeCLRRedirection(appdomain);
//LitJson 使用
//将一个对象转换成 json 字符串
string json = JsonMapper.ToJson(obj);
//json 字符串反序列化成对象
JsonTestClass obj = JsonMapper.ToObject<JsonTestClass>(json);

```

- **ILRuntime 的性能优化**

- 值类型优化: 使用 ILRuntime 外部定义的值类型 (例如 UnityEngine.Vector3) 在默认情况下会造成额外的装箱拆箱开销。ILRuntime 在 1.3.0 版中增加了值类型绑定 (ValueTypeBinding) 机制, 通过对这些值类型添加绑定器, 可以大幅增加值类型的执行效率, 以及避免 GC Alloc 内存分配。
- 大规模数值计算: 如果在热更内需要进行大规模数值计算, 则可以开启 ILRuntime 在 2.0 版中加入的寄存器模式来进行优化
- 避免使用 foreach: 尽量避免使用 foreach, 会不可避免地产生 GC。而 for 循环不会。
- 加载 dll 并在逻辑后处理进行简单调用
- 整个文件流程: 创建 IEnumerator 并运行-> 用文件流判断并读入 dll 和 pdb-> 尝试加载程序集 dll-> (如果加载成功) 初始化脚本引擎 (InitializeILRuntime()) -> 执行脚本引擎加载后的逻辑处理 (OnHotFixLoaded()) -> 程序销毁 (在 OnDestroy 中关闭 dll 和 pdb 的文件流)
- MemoryStream: 为系统提供流式读写。MemoryStream 类封装一个字节数组, 在构造实例时可以使用一个字节数组作为参数, 但是数组的长度无法调整。使用默认无参数构造函数创建实例, 可以使用 Write 方法写入, 随着字节数据的写入, 数组的大小自动调整。参考博客: 传送门

- `appdomain.LoadAssembly`: 将需要热更的 dll 加载到解释器中。第一个填入 dll 以及 pdb, 这里的 pdb 应该是 dll 对应的一些标志符号。后面的 `ILRuntime.Mono.Cecil.Pdb.PdbReaderProvider` 是动态修改程序集, 它的作用是给 `ILRuntime.Mono.Cecil.Pdb.PdbReaderProvider()` 里的 `GetSymbolReader()` (传入两个参数, 一个是通过转化后的 `ModuleDefinition.ReadModule(stre` (即 dll)) 模块定义, 以及原来的 symbol (即 pdb) `GetSymbolReader` 主要的作用是检测其中的一些符号和标志是否为空, 不为空的话就进行读取操作。(这些内容都是 `ILRuntime` 中的文件来完成)
- Unity MonoBehaviour lifecycle methods callback execute orders:
- 还有一个看起来不怎么清楚的, 将就凑合着看一下: 这几个图因为文件地址错误丢了, 改天再补一下
- IL 热更优点:
 - 1、无缝访问 C# 工程的现成代码, 无需额外抽象脚本 API
 - 2、直接使用 VS2015 进行开发, `ILRuntime` 的解译引擎支持 .Net 4.6 编译的 DLL
 - 3、执行效率是 L# 的 10-20 倍
 - 4、选择性的 CLR 绑定使跨域调用更快速, 绑定后跨域调用的性能能达到 `slua` 的 2 倍左右 (从脚本调用 `GameObject` 之类的接口)
 - 5、支持跨域继承 (代码里的完美学演示)
 - 6、完整的泛型支持 (代码里的完美学演示)
 - 7、拥有 Visual Studio 的调试插件, 可以实现真机源码级调试。支持 Visual Studio 2015 Update3 以及 Visual Studio 2017 和 Visual Studio 2019
 - 8、最新的 2.0 版引入的寄存器模式将数学运算性能进行了大幅优化

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/wuxiongb/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.Environment.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.Environment.AppDomain appdomain;

        public Adaptor() { }
        public Adaptor(ILRuntime.Runtime.Environment.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？不是集合元素里轮询的叠代器，而是协程

```
public class IEnumeratorObjectAdaptor : CrossBindingAdaptor {
// 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.Environment.AppDomain appdomain, ILTypeInstance instance) {
        return new Adaptor(appdomain, instance);
    }

// 这些个 IXxxx IEnumerator<T> 之类的，是在哪里定义的，是 ILRuntime 自己定义出来方便使用的，在下面的包裹里：
// using ILRuntime.CLR.TypeSystem;
// using ILRuntime.CLR.Method;
    internal class Adaptor : IEnumerator<object>, CrossBindingAdaptorType {
        ILTypeInstance instance;
        ILRuntime.Runtime.Environment.AppDomain appdomain;

        public Adaptor() { }
        public Adaptor(ILRuntime.Runtime.Environment.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;
        }
    }

    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.Environment.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.Environment.AppDomain appdomain, ILTypeInstance instance) {
        return new Adaptor(appdomain, instance);
    }

// 为了完整实现 MonoBehaviour 的所有特性, 这个 Adaptor 还得扩展, 这里只抛砖引玉, 只实现了最常用的 Awake, Start 和 Update
// 像我熟悉的安卓 Activity/Fragment 的生命周期有很多回调方法一样, MonoBehaviour 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.Environment.AppDomain appdomain;

    public Adaptor() { }
    public Adaptor(ILRuntime.Runtime.Environment.AppDomain appdomain, ILTypeInstance instance) {
        this.appdomain = appdomain;
        this.instance = instance;
    }
    public ILTypeInstance ILInstance { get { return instance; } set { instance = value; } }
    public ILRuntime.Runtime.Environment.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: 顾名思义, 就是工厂模式, 负责生产各种不同类型的实例 (生产人质)

5.2.1 enum FactoryType

```

public enum FactoryType {
    Singleton,
    Transient,
    Pool
}

```

```
}
```

5.2.2 interface IObjectFactory

```
public interface IObjectFactory {  
    object AcquireObject(string classFullName);  
    void ReleaseObject(object obj);  
}
```

5.2.3 PoolObjectFactory : Singleton<PoolObjectFactory>, IObjectFactory: 单例对象池: 结合工厂模式使用

```
// 单例对象池: 结合工厂模式使用; 现接触过的对象池包括 ThreadPool, ObjectPool,  
public class PoolObjectFactory : Singleton<PoolObjectFactory>, IObjectFactory {
```

```
    public class ObjectPool {  
        public readonly List<PoolData> _pool; // readonly  
        public int Max { // 该对象池储存数据上限  
            get;  
            set;  
        }  
        public bool Limit { // 该对象池设限了吗?  
            get;  
            set;  
        }  
    }  
    public ObjectPool() {  
        Limit = false;  
        _pool = new List<PoolData>();  
    }  
}
```

```
    public class PoolData {  
        public bool InUse {  
            get;  
            set;  
        }  
        public object Obj {  
            get;  
            set;  
        }  
    }  
}
```

// 这里, 每种类型对应一个对象池 (因为每种类型根据游戏需要可能初始化数量等不同有不同的要求), 用一个字典管理不同类型的对象池

```
    private readonly Dictionary<Type, ObjectPool> pool;  
    public PoolObjectFactory() {  
        pool = new Dictionary<Type, ObjectPool>();  
    }  
  
    public object AcquireObject(string classFullName) {  
        Type type = GameApplication.Instance.HotFix.LoadType(classFullName);  
        lock (pool) { // 这里需要上锁  
            if (pool.ContainsKey(type)) {  
                if (pool[type]._pool.Count > 0) {  
                    for (int i = 0; i < pool[type]._pool.Count; i++) {  
                        var p = pool[type]._pool[i];  
                        if (!p.InUse) { // 从该类型对象池里的现有对象链表中遍历出第一个没被使用的对象拿来用  
                            p.InUse = true;  
                            return p.Obj;  
                        }  
                    }  
                    // 该类型对象池数量已经达到了上限  
                    if (pool[type].Limit && pool[type]._pool.Count >= pool[type].Max)  
                        throw new Exception("max limit is arrived.");  
                }  
                object obj = GameApplication.Instance.HotFix.CreateInstance(classFullName);  
                var poolData = new PoolData {  
                    InUse = true,  
                    Obj = obj  
                };  
                if (!pool.ContainsKey(type)) {  
                    ObjectPool objPool = new ObjectPool();  
                    pool.Add(type, objPool);  
                }  
                pool[type]._pool.Add(poolData);  
                return obj;  
            }  
        }  
    }  
}
```

```

    public void ReleaseObject(object obj) { }
}

```

5.2.4 SingletonObjectFactory : IObjectFactory: 单例对象工厂

// 单例对象工厂

```

public class SingletonObjectFactory : IObjectFactory {

    private static Dictionary<Type, object> cachedObjects = null;
    private static readonly object _lock = new object();

    private Dictionary<Type, object> CachedObjects {
        get {
            lock (_lock) {
                if (cachedObjects == null)
                    cachedObjects = new Dictionary<Type, object>();
                return cachedObjects;
            }
        }
    }

    public object AcquireObject(string classFullName) {
        Type type = GameApplication.Instance.HotFix.LoadType(classFullName);
        if (CachedObjects.ContainsKey(type))
            return CachedObjects[type];
        lock (_lock) {
            var instance = GameApplication.Instance.HotFix.CreateInstance(classFullName);
            CachedObjects.Add(type, instance);
            return instance;
        }
    }

    public void ReleaseObject(object obj) { }
}

```

5.2.5 TransientObjectFactory : IObjectFactory

```

public class TransientObjectFactory : IObjectFactory {

    public object AcquireObject(string classFullName) {
        var instance = GameApplication.Instance.HotFix.CreateInstance(classFullName);
        return instance;
    }

    public void ReleaseObject(object obj) { }
}

```

5.3 Inject: 每个（自动）注入的背后，都是强大优化过的逻辑在支撑！爱表哥，爱生活

- 每个看似此情只该天上有何在发生在人间的神仙着侣故事背后都有着别人三生三世的守候相亲相爱和相互理解与付出做支撑，爱表哥，爱生活!!!

5.3.1 ServiceLocatorContain

```

public class ServiceLocatorContain {
    public string TypeName {
        get;
        private set;
    }
    public Func<object> Function {
        get;
        set;
    }
}

public ServiceLocatorContain(string name, Func<object> func) {
    TypeName = name;
    Function = func;
}
}

// 对象定位（创建）工厂：要么去对象池里抓一下；（当没有或是所有对象均在用且达上限）要么创建一个新的
public class ServiceLocator {
    private static readonly Dictionary<Type, ServiceLocatorContain> Container = new Dictionary<Type, ServiceLocatorContain>();

    // 两种不同类型的工厂以及对象的定位方法（每种类型提供两种不同参数的定位方法）
    private SingletonObjectFactory _singletonObjectFactory = new SingletonObjectFactory();
}

```

```

private TransientObjectFactory _transientObjectFactory = new TransientObjectFactory();

// 两种不同类型的工厂以及对象的定位方法（每种类型提供两种不同参数的定位方法）
public void RegisterSingleton(string interfaceName, string typeName) {
    ServiceLocatorContain contain = new ServiceLocatorContain(typeName, Lazy(FactoryType.Singleton, typeName));
    Type type = GameApplication.Instance.HotFix.LoadType(interfaceName);
    if (!Container.ContainsKey(type)) {
        Container.Add(type, contain);
    } else {
        throw new Exception("Container contains key: " + type);
    }
}

public void RegisterSingleton(string typeName) {
    ServiceLocatorContain contain = new ServiceLocatorContain(typeName, Lazy(FactoryType.Singleton, typeName));
    Type type = GameApplication.Instance.HotFix.LoadType(typeName);
    if (!Container.ContainsKey(type)) {
        Container.Add(type, contain);
    } else {
        throw new Exception("Container contains key: " + type);
    }
}

public void RegisterTransient(string interfaceName, string typeName) {
    ServiceLocatorContain contain = new ServiceLocatorContain(typeName, Lazy(FactoryType.Transient, typeName));
    Type type = GameApplication.Instance.HotFix.LoadType(interfaceName);
    if (!Container.ContainsKey(type)) {
        Container.Add(type, contain);
    } else {
        throw new Exception("Container contains key: " + type);
    }
}

public void RegisterTransient(string typeName) {
    ServiceLocatorContain contain = new ServiceLocatorContain(typeName, Lazy(FactoryType.Transient, typeName));
    Type type = GameApplication.Instance.HotFix.LoadType(typeName);
    if (!Container.ContainsKey(type)) {
        Container.Add(type, contain);
    } else {
        throw new Exception("Container contains key: " + type);
    }
}

public void Clear() {
    Container.Clear();
}

public TInterface Resolve<TInterface>(string keyName) where TInterface : class {
    return Resolve(GameApplication.Instance.HotFix.LoadType(keyName)) as TInterface;
}

private static object Resolve(Type type) {
    if (!Container.ContainsKey(type))
        return null;
    return Container[type].Function();
}

private Func<object> Lazy(FactoryType factoryType, string typeFullName) {
    return () => {
        switch (factoryType) {
            case FactoryType.Singleton:
                return _singletonObjectFactory.AcquireObject(typeFullName);
            default:
                return _transientObjectFactory.AcquireObject(typeFullName);
        }
    };
}
}

```

5.4 Wrap

5.4.1 BehaviourWrap : MonoBehaviour

// 需要包装的原因是：Unity 程序集下的这些子类也需要能够被热更新程序集认得，
 // 所以对必要的回调函数进行包装改写

```

public class BehaviourWrap : MonoBehaviour {
    public string Description {
        get;
        set;
    }

    public object Data {
        get;
        set;
    }
}

```



```
#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 (本质上是程序脚本的实例化)；加载完毕自动触发 StartHotFix
IEnumerator Initialize() {
    // 游戏初始时，实例化获取一个单例资源 (管理类)，架起两个不同程序集之间沟通的桥梁
    ResourceMap resourceMap = gameObject.AddComponent<ResourceMap>(); // 整个游戏应用，全局唯一
    resourceMap.OnInitializeSuccess += StartHotFix; // 必备资源管理初始化好，自动触发游戏热更新程序模块集

    // 将 Unity 程序集中初始化好的实例 (通过公用接口) 桥接 refer 给热更新程序集引用，从而实现热更新程序集可以调用 unity 中的资源 (场景，
    ResourceConstant.Loader = resourceMap; // 可以重点看一下两个不同模块之间的资源管理的关系

    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.Environment.AppDomain appDomain;
```

```

void Start() {
    appDomain = new ILRuntime.Runtime.Environment.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.LoadAssetAsync<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 IEnumeratorObjectAdapter()); // 协程适配
    appDomain.RegisterCrossBindingAdaptor(new MonoBehaviourAdapter()); // <===== 临时补了一下, 也有了
    appDomain.RegisterCrossBindingAdaptor(new InterfaceCrossBindingAdapter());
}

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 ILInterpreter.PushObject(esp, mStack, ((ILType)t).Instantiate());
        } else
            return ILInterpreter.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
// 同一参数组合只需要注册一次

// 如果只在热更新的 DLL 项目中使用的委托，是不需要任何额外操作的，就跟在通常的 C# 里那样使用即可；
// 如果你需要将委托实例传给 ILRuntime 外部使用，那则根据情况，你需要额外添加适配器或者转换器，目的是为了适应外部的环境。
// 如果在运行时发现缺少注册某个指定类型的委托适配器或者转换器时，ILRuntime 会抛出相应的异常，根据提示添加注册即可。
void InitializeDelegateSetting() { // 注册，不带返回值，最多支持五个参数传入；注册，带参数返回值，最后一个参数为返回值，最多支持
    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
// 委托转换器：需要将一个不是 Action 或者 Func 类型的委托实例传到 ILRuntime 外部使用，需要写委托适配器和委托转换器。
// 委托转换器将 Action 和 Func 转换成你真正需要的那个委托类型
// 为了避免不必要的麻烦，以及后期热更出现问题，建议： 1、尽量避免不必要的跨域委托调用 2、尽量使用 Action 以及 Func 委托类型

// ILRuntime 内部是使用 Action，以及 Func 这两个系统自带委托类型来生成的委托实例，
// 所以如果你需要将一个不是 Action 或者 Func 类型的委托实例传到 ILRuntime 外部使用的话，
// 除了委托适配器，还需要额外写一个转换器，将 Action 和 Func 转换成你真正需要的那个委托类型。
// 在热更新的所有视图里，可能都会涉及到不同类型的比如控件的点击回调事件等。要在热更新程序包之外可以使用，以及使用得便利，就把所有不同
    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>((action) => {
        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.LoadAssetAsync<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.LoadAssetAsync<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.GetType().FirstOrDefault(t => t.FullName == typeName);
    if (type == null) {
        DebugHelper.LogError(string.Format("Can't find Class by class name:'{0}'", typeName), true);
        throw new Exception(string.Format("Can'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 | Bindings
        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, 不管是蒹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);
}

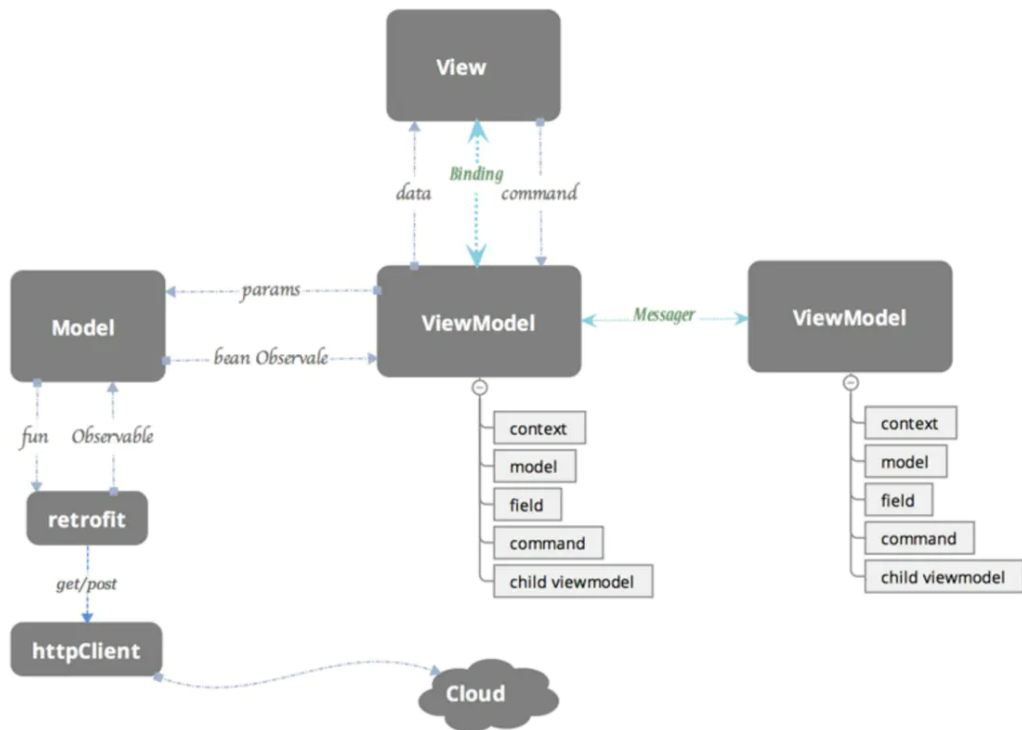
// 这里告诉一个常识说: 任何一个 ViewModel, 不管是蒹类还是继承后的子类, 它们都有可能有好几个视图, 需要与其解绑定
public void UnBind(ViewModelBase viewModel) {
    if (viewModel != null)
        for (int i = 0; i < unBinders.Count; i++)
            unBinders[i](viewModel);
}

}

```

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?

- 因为对这个框架里的这个模块不够理解, 先抓一个网上的图, 再把这一块儿补起来, 最主要的作用猜测应该还是跨域过程中对于不同回调 等的传递封装管理



- 那么上面的图是说，可以在不同的 ViewModel 之间传递消息？
- MVVM 可以双方数据绑定；这里应该是借用了 MVVM 的外套，但是里面基本所有必要的相关逻辑都是在这个框架中自己封装的。数据（不管是什么数据类型吧）的订阅与发布是被封装成一个一个的消息发送出去的。
- 这里消息桶的全局单例模式就像 Handler 里的 MessageQueue 一样是用来全权负责数据更改的变化通知，消息的订阅与发布等相关逻辑。
- 再具体的应用里的应用可以再找真正用到的地方具体理解消化一下。

6.2.1 MessageArgs<T>: (跨域跨程序集的) 消息参数泛型类？是泛型类，做的事情就是对泛型基类逻辑的必要封装

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

6.2.2 MessageAggregator<T> : Singleton<MessageAggregator<T>>: 单例模式的泛型集合管理类，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）？这里要再看一遍，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
- 这是一个架构设计，定义在 framework 里，但是仍然可以提供给热更新程序包使用
- 如果是全部在热更工程中的继承，用法跟普通继承是一样的。如果你想在热更 DLL 项目当中继承一个 Unity 主工程里的类，或者实现一个主工程里的接口，这就是跨域继承，你需要在 Unity 主工程中实现一个继承适配器。
- 在 ILRuntime 中，存在两种继承，一种就是普通的继承，和平时的继承的用法一样，这里我们着重介绍下第二种继承，跨域继承，
- 为 unity 工程和热更工程处于两个不同的程序域，是无法直接进行代码互通的。所以需要去跨域继承。
- 这里留个将来还可供参考的链接

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

// 抽象基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(ModuleBaseAdapter);
    }
}
public override object CreateCLRInstance(ILRuntime.Runtime.Environment.AppDomain appdomain, ILTypeInstance instance) {
    return new ModuleBaseAdapter(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 ModuleBaseAdapter : 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.Environment.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 ModuleBaseAdapter() { }
}

```

```

public ModuleBaseAdaptor(ILRuntime.Runtime.Environment.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>: 这个特型（自创的用语，相对于泛型来说）的接口，实现了个性化 MVVM 设计 View 与 ViewModel 的自动绑定

```

public interface IView<ViewModelBase> { // 泛型的写法不应该是 IView<T> where T : d1fkjkd 吗？
    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() + OnDestroy(): ViewModelBase 里定义的抽象方法实现，实现必要的基类逻辑

```

```

public virtual void OnDestroy() {
    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
// 去追这个方法：还是说，同一个视图可以绑定不同的其它的 ViewModelBase 呢？还是说只是在渐变过程中的过渡视图模型转换？去找实体类里的方
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.Environment.AppDomain appdomain, ILTypeInstance instance) {
    return new UnityGuiViewAdaptor(appdomain, instance);
}
```

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

    public UnityGuiViewAdaptor() { }
    public UnityGuiViewAdaptor(ILRuntime.Runtime.Environment.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 OnDestroy() {
    if (!_onDestroyGot) {
        _onDestroy = instance.Type.GetMethod("OnDestroy");
        _onDestroyGot = true;
    }
    if (_onDestroy != null && !isOnDestroyInvoking) {
        isOnDestroyInvoking = true;
        appdomain.Invoke(_onDestroy, instance);
        isOnDestroyInvoking = false;
    } else {
        base.OnDestroy();
    }
}
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 OnDestroy() {}
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 BaseType {
    get {
        return typeof(ViewModelBase);
    }
}
```

```
public override Type AdaptorType {
```

```
get {
    return typeof(ViewModelBaseAdaptor);
}
```

```
public override object CreateCLRInstance(ILRuntime.Runtime.Environment.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，需要实现里面定义过的共有 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):
```

```

    appdomain.Invoke(=onStartReveal,
    _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 OnDestroy() {
        if (!_onDestroyGot) {
            _onDestroy = instance.Type.GetMethod("OnDestroy");
            _onDestroyGot = true;
        }
        if (_onDestroy != null && !_isOnDestroyInvoking) {
            _isOnDestroyInvoking = true;
            appdomain.Invoke(_onDestroy, instance);
            _isOnDestroyInvoking = false;
        } else
            base.OnDestroy();
    }
    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 ResourceManager 包裹 ResMgr

7.1 AssetLoader: 资源加载器

7.1.1 abstract class AssetLoader: Asset 加载封装抽象类

- 这里定义的是一个公用接口类，供实现

```

// Asset 加载封装抽象类:
public abstract class AssetLoader {

    public GameObject root;
    public string bundleName;
    public string assetName;

    public EAssetBundleUnloadLevel unloadLevel = EAssetBundleUnloadLevel.Never;
    public AssetBundleSpec Spec {
        get;
        set;
    }
    public string FinalBundleName {
        get {
            if (!bundleName.EndsWith(ResourceConstant.bundleExtension))
                return bundleName + ResourceConstant.bundleExtension;
            return bundleName;
        }
    }
    public Action<AssetLoader> onAssetLoadEnd;
    public Action<string> onAssetLoadError;
    public abstract void LoadAssetFromBundle(AssetBundle assetBundle);
    public abstract void LoadError();
}

public class AssetLoader<T> : AssetLoader where T : UnityEngine.Object {
    AssetBundleRequest currentRequest;
    public Action<T> onLoadOver;
    public override void LoadAssetFromBundle(AssetBundle assetBundle) {
        CoroutineHelper.StartCoroutine(LoadAssetFromBundleAsync(assetBundle));
    }
    IEnumerator LoadAssetFromBundleAsync(AssetBundle assetBundle) {
        DateTime regTime = DateTime.Now;
        string name = string.IsNullOrEmpty(assetName) ? bundleName : assetName;
        var n = name.Split('/');
        try {
            string finalName = n[n.Length - 1];
            currentRequest = assetBundle.LoadAssetAsync<T>(finalName);
        }
        catch (Exception e) {
            DebugHelper.LogError(string.Format("LoadAssetAsync [{0}][{1}]Error,[{2}][{3}]", bundleName, assetName, e.Message), e);
        }
        yield return currentRequest;
        var costTime = (DateTime.Now - regTime).TotalMilliseconds;
        //DebugHelper.Log(bundleName + " LoadAssetAsync cost " + costTime, true);
        if (currentRequest != null) {
            LoadOver((T)currentRequest.asset);
        }
    }
    protected void LoadOver(T asset) {
        if (onLoadOver != null) {
            try {
                onLoadOver(asset);
                if (unloadLevel == EAssetBundleUnloadLevel.LoadOver) {
                    ResourceConstant.Loader.Unload(FinalBundleName, false);
                }
            }
            catch (Exception e) {
                DebugHelper.LogError(string.Format("Load [{0}][{1}]Error,[{2}][{3}]", bundleName, assetName, e.Message), e);
            }
        }
    }
}

```

```

        if (onAssetLoadEnd != null) {
            onAssetLoadEnd(this);
        }
    }
    public override void LoadError() {
        try {
            LoadOver(null);
        }
        catch (Exception e) {
            DebugHelper.LogError(FinalBundleName + " LoadError " + e.Message + e.StackTrace, true);
        }
        if (onAssetLoadEnd != null) {
            onAssetLoadEnd(this);
        }
    }
}

```

7.1.2 GameObjectLoader : AssetLoader: GameObjectAsset 加载类

// GameObjectAsset 加载类

```

public class GameObjectLoader : AssetLoader {

    AssetBundleRequest currentRequest;
    public Action<GameObject> onCloneOver;

    public override void LoadAssetFromBundle(AssetBundle assetBundle) {
        CoroutineHelper.StartCoroutine(LoadAssetFromBundleAsync(assetBundle));
    }

    IEnumerator LoadAssetFromBundleAsync(AssetBundle assetBundle) {
        var name = string.IsNullOrEmpty(assetName) ? bundleName : assetName;
        var n = name.Split('/');
        try { // 异步调用、跨域调用, 需要捕获异常?
            string finalName = n[n.Length - 1];
            currentRequest = assetBundle.LoadAssetAsync<GameObject>(finalName);
        } catch (Exception e) {
            DebugHelper.LogError(string.Format("LoadAssetAsync [{0}][{1}]Error,[{2}][{3}]",
                bundleName, assetName, e.Message, e.StackTrace), true);
        }
        yield return currentRequest;
        if (currentRequest != null)
            LoadGameObject(assetBundle, currentRequest.asset, name);
    }

    void LoadGameObject(AssetBundle ab, UnityEngine.Object asset, string aname) {
        var b = asset;
        if (b != null) {
            var clone = GameObject.Instantiate(b) as GameObject;
            CloneOver(clone);
        } else {
            DebugHelper.LogError((aname == null ? "aname==null" : aname)
                + " not find,use default" + (b == null ? " 返回空" : (b.GetType() + "--" + b.name)), true);
        }
    }

    protected void CloneOver(GameObject go) {
        if (!go) {
            return;
        }
        if (root)
            go.transform.parent = root.transform;
        //go.transform.localPosition = Vector3.zero;
        //go.transform.localScale = Vector3.one;
        //go.transform.localRotation = new Quaternion();
        try {
            if (onCloneOver != null) {
                onCloneOver(go);
                if (unloadLevel == EAssetBundleUnloadLevel.LoadOver)
                    ResourceConstant.Loader.Unload(FinalBundleName, false);
            }
        } catch (Exception e) {
            DebugHelper.LogError(string.Format("CloneOver [{0}][{1}]Error,[{2}][{3}]", bundleName, assetName, e.Message, e.
            ));
        }
        if (onAssetLoadEnd != null)
            onAssetLoadEnd(this);
    }

    public override void LoadError() {
        try {
            if (onCloneOver != null)

```

```

        onCloneOver(null);
    } catch (Exception e) {
        DebugHelper.LogError(FinalBundleName + " loadError " + e.Message + e.StackTrace, true);
    }
    if (onAssetLoadEnd != null)
        onAssetLoadEnd(this);
}
}

```

7.1.3 SceneLoader : AssetLoader: 场景 Asset 加载类

// 场景 Asset 加载类

```

public class SceneLoader : AssetLoader {
    public bool isAdditive;
    public Action onLoadOver;

    public override void LoadAssetFromBundle(AssetBundle assetBundle) {
        CoroutineHelper.StartCoroutine(LoadAssetFromBundleAsync(assetBundle));
    }
    IEnumerator LoadAssetFromBundleAsync(AssetBundle assetBundle) { // 使用协程异步加载
        if (isAdditive) {
            yield return SceneManager.LoadSceneAsync(assetName, LoadSceneMode.Additive);
        } else {
            yield return SceneManager.LoadSceneAsync(assetName);
        }
        LoadOver();
    }
    protected void LoadOver() {
        if (onLoadOver != null) {
            try {
                onLoadOver();
                if (unloadLevel == EAssetBundleUnloadLevel.LoadOver)
                    ResourceConstant.Loader.Unload(FinalBundleName, false);
            } catch (Exception e) {
                DebugHelper.LogError(string.Format("Load [{0}][{1}]Error,[{2}][{3}]", bundleName, assetName, e.Message, e.StackTrace), e);
            }
        }
        if (onAssetLoadEnd != null)
            onAssetLoadEnd(this);
    }

    public override void LoadError() {
        try {
            LoadOver();
        } catch (Exception e) {
            DebugHelper.LogError(FinalBundleName + " loadError " + e.Message + e.StackTrace, true);
        }
        if (onAssetLoadEnd != null)
            onAssetLoadEnd(this);
    }
}

```

7.2 AssetBundleSpec: 热更新资源包的相关参数配置与读取, 资源包的加载等

- 资源包的相关处理。热更新分代码热更新和资源热更新, 一个应用中可以只有某一类或是两者兼有。这里就是处理资源热更新中资源包相关的逻辑。

```

// 资源包数据结构类 (相关参数说明类)
public class AssetBundleSpec {
    // 资源名, get, private set, 下面 8 个属性都是如此
    public string Name {
        get;
        private set;
    }
    // // 资源文件 MD5
    // public string MD5 {}
    // // 文件长度
    // public int Length {}
    // // 资源来源分类
    // public EAssetBunbleSourceType AssetBundleSourceType {}
    // // 是否被缓存
    // public bool IsCached {}
    // // 是否强行中断加载
    // public bool IsForceInterruptLoad {}
    // public EAssetBundleUnloadLevel UnloadLevel {}
}

```

```

// public AssetBundle AssetBundle {}
// public byte[] Data {}

bool isLoading = false;
bool isDownloading = false;
bool IsFileExist {
    get {
        return FileHelp.IsFileExists(Name);
    }
}
List<AssetLoader> assetLoaders = new List<AssetLoader>();

public Action<AssetBundleSpec> onDownloadSuccess; // 委托设计模式: 这里自己明明是写过但是忘记了, 要补
public Action<AssetBundleSpec> onDownloadFail;

public AssetBundleSpec(string name, string md5, int length, EAssetBunbleSourceType sourceType) {
    Name = name;
    MD5 = md5;
    Length = length;
    AssetBundleSourceType = sourceType;
}

public void Check(string md5, int length, EAssetBunbleSourceType sourceType) {
    if (sourceType == EAssetBunbleSourceType.Server) {
        if (md5 == MD5) {
            //Debug.Log("Name: " + Name + " md5: " + md5 + " MD5: " + MD5);
            IsCached = true;
        }
        else {
            AssetBundleSourceType = sourceType;
            MD5 = md5;
            Length = length;
        }
    }
}

#region Unload
// 卸载
public void Unload(bool unloadAllLoadedObjects) {
    if (AssetBundle != null) {
        AssetBundle.Unload(unloadAllLoadedObjects);
        AssetBundle = null;
    }
    if (assetLoaders != null && IsForceInterruptLoad)
        assetLoaders.Clear();
}

#endregion
#region Download
// 开始从网上下载
public void TryDownloadAsset() {
    string url = Path.Combine(ResourceConstant.RemoteAssetBundleUrl, Name);
    url = url + "?timestamp=" + DateTime.Now.ToString();
    isDownloading = true;
    DateTime regTime = DateTime.Now;
    HttpHelper.Instance.Get(url, (request) => {
        var costTime = (DateTime.Now - regTime).TotalMilliseconds;
        DebugHelper.Log(Name + " TryDownloadAsset cost " + costTime + " " + url, true);
        DownloadSuccess(request, Name);
    }, (UnityWebRequest request) => {
        DownloadFail(request, TryDownloadAsset);
    }, 3);
}

public UnityWebRequest request;
public void DownloadAsset() {
    DateTime regTime = DateTime.Now;
    if (request == null) {
        string path = Path.Combine(ResourceConstant.RemoteAssetBundleUrl, Name);
        path = path + "?timestamp=" + DateTime.Now.ToString();
        isDownloading = true;
        request = HttpHelper.Instance.Get(path);
        request.timeout = 30;
    }
    request.SendWebRequest();
    while (!request.isDone) {
        continue;
    }
    var costTime = (DateTime.Now - regTime).TotalMilliseconds;
    DebugHelper.Log(Name + " DownloadAsset cost " + costTime, true);
    if (request.error == null && request.responseCode == 200) {
        DownloadSuccess(request, Name);
    }
    else {
        DownloadFail(request, DownloadAsset);
    }
}

```

```

    }
}
// 下载成功
void DownloadSuccess(UnityWebRequest request, string name) {
    isDownloading = false;
    string cachedAssetBundlePath = Path.Combine(ResourceConstant.AssetBundleCacheRoot, name);
    string dir = Path.GetDirectoryName(cachedAssetBundlePath);
    if (!Directory.Exists(dir)) {
        Directory.CreateDirectory(dir);
    }
    byte[] bytes = request.downloadHandler.data;
    MD5 = CryptoHelp.MD5(bytes, 0, bytes.Length);
    Length = bytes.Length;
    if (ResourceConstant.CacheAssetBundle) {
        FileStream fs = new FileStream(cachedAssetBundlePath, FileMode.Create);
        fs.Write(bytes, 0, bytes.Length);
        fs.Close();
        fs.Dispose();
    }
    DebugHelper.Log("CacheAsset: " + cachedAssetBundlePath, true);
    IsCached = true;
    AssetBundleSourceType = EAssetBunbleSourceType.Hotfix;
    if (Name.EndsWith(ResourceConstant.bundleExtension)) {
        if (AssetBundle == null) {
            DateTime regTime = DateTime.Now;
            AssetBundle = AssetBundle.LoadFromMemory(bytes);
            var costTime = (DateTime.Now - regTime).TotalMilliseconds;
            DebugHelper.Log(Name + " LoadFromMemory cost " + costTime, true);
        }
    } else {
        Data = bytes;
    }
    ResourceMap.Instance.OnCachedAssetBundleSpec(this);
    AssetBundleDownLoadReady();
    if (onDownloadSuccess != null) // 自动触发相关事件的回调
        onDownloadSuccess(this);
}
// 下载失败
void DownloadFail(UnityWebRequest request, Action retryAction) {
    isDownloading = false;
    DebugHelper.LogError("DownloadFail " + request.error + " uri: " + request.url, true);
    if (onDownloadFail != null) {
        onDownloadFail(this);
    }
    //retryAction();
}
void AssetBundleDownLoadReady() {
    foreach (var assetLoader in assetLoaders)
        assetLoader.LoadAssetFromBundle(AssetBundle);
    assetLoaders.Clear();
}
}
#endregion
#region Load
// 同步加载 Asset
public T LoadAsset<T>(string assetName, EAssetBundleUnloadLevel
    unloadLevel = EAssetBundleUnloadLevel.ChangeSceneOver) where T : UnityEngine.Object {
    TryReadyAssetBundle(unloadLevel);
    if (AssetBundle == null) {
        DebugHelper.LogError("LoadAsset " + typeof(T).Name + " " + Name + " " + assetName + " return null", true);
        return null;
    }
    var name = string.IsNullOrEmpty(assetName) ? Name : assetName;
    var n = name.Split('/');
    var asset = AssetBundle.LoadAsset<T>(n[n.Length - 1]);
    return asset;
}
public TMP_FontAsset LoadTMP_FontAsset(string assetName, EAssetBundleUnloadLevel unloadLevel = EAssetBundleUnloadLevel.ChangeSceneOver) {
    TryReadyAssetBundle(unloadLevel);
    if (AssetBundle == null) {
        DebugHelper.LogError("LoadAsset " + typeof(TMP_FontAsset).Name + " " + Name + " " + assetName + " return null");
        return null;
    }
    var name = string.IsNullOrEmpty(assetName) ? Name : assetName;
    var n = name.Split('/');
    var asset = AssetBundle.LoadAsset<TMP_FontAsset>(n[n.Length - 1]);
    return asset;
}
}
public Font LoadFont(string assetName, EAssetBundleUnloadLevel unloadLevel = EAssetBundleUnloadLevel.ChangeSceneOver) {
    TryReadyAssetBundle(unloadLevel);
}

```

```

        if (AssetBundle == null) {
            DebugHelper.LogError("LoadAsset " + typeof(Font).Name + " " + Name + " " + assetName + " return null", tr
            return null;
        }
        var name = string.IsNullOrEmpty(assetName) ? Name : assetName;
        var n = name.Split('/');
        var asset = AssetBundle.LoadAsset<Font>(n[n.Length - 1]);
        return asset;
    }
    public AnimationClip LoadAnimationClip(string assetName, EAssetBundleUnloadLevel unloadLevel = EAssetBundleUnload
    TryReadyAssetBundle(unloadLevel);
    if (AssetBundle == null) {
        DebugHelper.LogError("LoadAsset " + typeof(AnimationClip).Name + " " + Name + " " + assetName + " return
        return null;
    }
    var name = string.IsNullOrEmpty(assetName) ? Name : assetName;
    var n = name.Split('/');
    var asset = AssetBundle.LoadAsset<AnimationClip>(n[n.Length - 1]);
    return asset;
}
    public AnimatorOverrideController LoadAnimatorOverrideController(string assetName, EAssetBundleUnloadLevel unload
    TryReadyAssetBundle(unloadLevel);
    if (AssetBundle == null) {
        DebugHelper.LogError("LoadAsset " + typeof(AnimatorOverrideController).Name + " " + Name + " " + assetName
        return null;
    }
    var name = string.IsNullOrEmpty(assetName) ? Name : assetName;
    var n = name.Split('/');
    var asset = AssetBundle.LoadAsset<AnimatorOverrideController>(n[n.Length - 1]);
    return asset;
}
    public RuntimeAnimatorController LoadRuntimeAnimatorController(string assetName, EAssetBundleUnloadLevel unloadL
    TryReadyAssetBundle(unloadLevel);
    if (AssetBundle == null) {
        DebugHelper.LogError("LoadAsset " + typeof(RuntimeAnimatorController).Name + " " + Name + " " + assetName
        return null;
    }
    var name = string.IsNullOrEmpty(assetName) ? Name : assetName;
    var n = name.Split('/');
    var asset = AssetBundle.LoadAsset<RuntimeAnimatorController>(n[n.Length - 1]);
    return asset;
}
    public AudioClip LoadAudioClip(string assetName, EAssetBundleUnloadLevel unloadLevel = EAssetBundleUnloadLevel.Ch
    TryReadyAssetBundle(unloadLevel);
    if (AssetBundle == null) {
        DebugHelper.LogError("LoadAsset " + typeof(AudioClip).Name + " " + Name + " " + assetName + " return null
        return null;
    }
    var name = string.IsNullOrEmpty(assetName) ? Name : assetName;
    var n = name.Split('/');
    var asset = AssetBundle.LoadAsset<AudioClip>(n[n.Length - 1]);
    return asset;
}
    public Material LoadMaterial(string assetName, EAssetBundleUnloadLevel unloadLevel = EAssetBundleUnloadLevel.Chan
    TryReadyAssetBundle(unloadLevel);
    if (AssetBundle == null) {
        DebugHelper.LogError("LoadAsset " + typeof(Material).Name + " " + Name + " " + assetName + " return null
        return null;
    }
    var name = string.IsNullOrEmpty(assetName) ? Name : assetName;
    var n = name.Split('/');
    var asset = AssetBundle.LoadAsset<Material>(n[n.Length - 1]);
    return asset;
}
    public TextAsset LoadTextAsset(string assetName, EAssetBundleUnloadLevel unloadLevel = EAssetBundleUnloadLevel.Ch
    TryReadyAssetBundle(unloadLevel);
    if (AssetBundle == null) {
        DebugHelper.LogError("LoadAsset " + typeof(TextAsset).Name + " " + Name + " " + assetName + " return null
        return null;
    }
    var name = string.IsNullOrEmpty(assetName) ? Name : assetName;
    var n = name.Split('/');
    var asset = AssetBundle.LoadAsset<TextAsset>(n[n.Length - 1]);
    return asset;
}
    public Sprite LoadSprite(string assetName, EAssetBundleUnloadLevel unloadLevel = EAssetBundleUnloadLevel.ChangeS
    TryReadyAssetBundle(unloadLevel);
    if (AssetBundle == null) {
        DebugHelper.LogError("LoadAsset " + typeof(Sprite).Name + " " + Name + " " + assetName + " return null",

```

```

        return null;
    }
    var name = string.IsNullOrEmpty(assetName) ? Name : assetName;
    var n = name.Split('/');
    var asset = AssetBundle.LoadAsset<Sprite>(n[n.Length - 1]);
    return asset;
}
public Texture2D LoadTexture2D(string assetName, EAssetBundleUnloadLevel unloadLevel = EAssetBundleUnloadLevel.ChangeSceneOver)
{
    TryReadyAssetBundle(unloadLevel);
    if (AssetBundle == null) {
        DebugHelper.LogError("LoadAsset " + typeof(Texture2D).Name + " " + Name + " " + assetName + " return null");
        return null;
    }
    var name = string.IsNullOrEmpty(assetName) ? Name : assetName;
    var n = name.Split('/');
    var asset = AssetBundle.LoadAsset<Texture2D>(n[n.Length - 1]);
    return asset;
}
// 同步加载场景
public void LoadScene(string assetName, EAssetBundleUnloadLevel unloadLevel = EAssetBundleUnloadLevel.ChangeSceneOver)
{
    TryReadyAssetBundle(unloadLevel);
    var name = string.IsNullOrEmpty(assetName) ? Name : assetName;
    var n = name.Split('/');
    if (AssetBundle == null) {
        DebugHelper.LogError("LoadScene " + Name + " " + assetName + " return null", true);
        return;
    }
    SceneManager.LoadScene(assetName);
}
// 同步加载 GameObject
public GameObject LoadClone(string assetName, EAssetBundleUnloadLevel unloadLevel = EAssetBundleUnloadLevel.ChangeSceneOver)
{
    TryReadyAssetBundle(unloadLevel);
    var name = string.IsNullOrEmpty(assetName) ? Name : assetName;
    var n = name.Split('/');
    GameObject obj = null;
    obj = AssetBundle.LoadAsset<GameObject>(n[n.Length - 1]);
    UnityEngine.Object tempObject = null;
    if (obj == null) {
        tempObject = AssetBundle.mainAsset;
        if (tempObject == null) {
            var allAssets = AssetBundle.LoadAllAssets();
            foreach (var asset in allAssets) {
                DebugHelper.LogError(asset.name + asset.GetType(), true);
            }
            DebugHelper.LogError((name == null ? "aname==null" : name) + " not find,use default" + (tempObject == null ? "tempObject==null" : tempObject.name));
            return null;
        }
    }
    else {
        var cloneObj = GameObject.Instantiate(obj);
        return cloneObj;
    }
}
}
public void TryReadyAssetBundle(EAssetBundleUnloadLevel unloadLevel = EAssetBundleUnloadLevel.ChangeSceneOver) {
    //Debug.Log("AssetBundleName: " + Name + " IsCached: " + IsCached + " IsFileExist: " + IsFileExist);
    if (!IsCached || !IsFileExist) {
        //Debug.Log("AssetBundleName: " + Name + " DownloadAsset");
        DownloadAsset();
        //TryDownloadAsset();
        if (UnloadLevel != EAssetBundleUnloadLevel.Never) {
            UnloadLevel = unloadLevel;
        }
    }
    else if (AssetBundle == null) {
        //Debug.Log("AssetBundleName: " + Name + " LoadFromFile");
        AssetBundle = LoadAssetBundleFromFile(unloadLevel);
    }
}
}
AssetBundle LoadAssetBundleFromFile(EAssetBundleUnloadLevel unloadLevel = EAssetBundleUnloadLevel.ChangeSceneOver)
{
    try {
        byte[] datas = FileHelp.Read(Name);
        if (datas == null) {
            return null;
        }
        var assetBundle = AssetBundle.LoadFromMemory(datas);
        if (assetBundle != null) {
            return assetBundle;
        }
        else {
            DebugHelper.LogError("LoadAssetBundleFromFile return null " + Name, true);
            return null;
        }
    }
}

```

```

    }
    catch (Exception e) {
        DebugHelper.LogError("LoadAssetBundleFromFile AESDecrypt Error finalBundleName: " + Name + " " + e, true);
        return null;
    }
}
#endregion
#region LoadAsync
public void TryLoadAsync(AssetLoader assetLoader, EAssetBundleUnloadLevel unloadLevel, bool isForceInterruptLoad = false)
{
    isForceInterruptLoad = isForceInterruptLoad;
    if (AssetBundle != null) {
        assetLoader.LoadAssetFromBundle(AssetBundle);
    } else {
        if (unloadLevel != EAssetBundleUnloadLevel.Never) {
            unloadLevel = unloadLevel;
        }
        //DebugHelper.Log(Name + "TryLoadAsync ", true);
        if (isDownloading || isLoading) {
            assetLoaders.Add(assetLoader);
        } else if (!IsCached || !IsFileExist) {
            assetLoaders.Add(assetLoader);
            TryDownloadAsset();
        } else {
            ReadyLoadAsync(assetLoader);
        }
    }
}

Coroutine asynLoadCoroutine;
WWW thisWWW;
// 开始异步加载
void ReadyLoadAsync(AssetLoader assetLoader) {
    bool isInCacheFile = false;
    string url = ResourceConstant.GetFileUrl(Name, out isInCacheFile);
    if (asynLoadCoroutine != null) {
        CoroutineHelper.StopCoroutine(asynLoadCoroutine);
    }
    if (string.IsNullOrEmpty(url)) {
        DebugHelper.LogError("LoadAssetBundleAsync url IsNullOrEmpty " + Name, true);
        LoadError();
        return;
    }
    isLoading = true;
    asynLoadCoroutine = CoroutineHelper.StartCoroutine(LoadAsync(url, assetLoader));
}

IEnumerator LoadAsync(string url, AssetLoader assetLoader) {
    try {
        thisWWW = new WWW(url);
    }
    catch (Exception e) {
        DebugHelper.LogError("DownloadAsync " + url + " error" + e.Message + e.StackTrace, true);
    }
    yield return thisWWW;
    if (thisWWW == null) {
        DebugHelper.LogError("DownloadAsync " + url + " thisWWW==null", true);
        LoadError();
    } else if (!string.IsNullOrEmpty(thisWWW.error)) {
        DebugHelper.LogError("DownloadAsync " + thisWWW.error + "URL: " + url, true);
        yield return new WaitForEndOfFrame();
        if (thisWWW != null) {
            thisWWW.Dispose();
            thisWWW = null;
        }
        LoadError();
    } else {
        LoadAsyncSuccess(url, assetLoader);
    }
}

void LoadAsyncSuccess(string url, AssetLoader assetLoader) {
    if (AssetBundle == null) {
        AssetBundle = thisWWW.assetBundle;
    }
    isLoading = false;
    if (AssetBundle != null) {
        assetLoader.LoadAssetFromBundle(AssetBundle);
        AssetBundle.DownloadReady();
    } else {
        DebugHelper.LogError("DownloadAsyncSuccess thisWWW.assetBundle==null " + url, true);
        assetLoader.LoadError();
    }
}

```



```

    }
    public void LoadError() {
        isLoading = false;
    }
}
#endregion
}

```

7.3 interface IResourceLoader: 不同类型的资源加载公用接口类: 定义的是对于不同类型的资源的加载调用公用 API 接口

// 公用接口类: 定义的是对于不同类型的资源的加载调用公用 API 接口

```

public interface IResourceLoader {
    #region Load
    T LoadAsset<T>(string bundleName, string assetName,
        EAssetBundleUnloadLevel unloadLevel = EAssetBundleUnloadLevel.ChangeSceneOver)
        where T : UnityEngine.Object;
    TMP_FontAsset LoadTMP_FontAsset(string bundleName, string assetName,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver);
    Font LoadFont(string bundleName, string assetName,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver);
    AnimationClip LoadAnimationClip(string bundleName, string assetName,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver);
    AnimatorOverrideController LoadAnimatorOverrideController(string bundleName, string assetName,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver);
    RuntimeAnimatorController LoadRuntimeAnimatorController(string bundleName, string assetName,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver);
    AudioClip LoadAudioClip(string bundleName, string assetName,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver);
    Material LoadMaterial(string bundleName, string assetName,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver);
    TextAsset LoadTextAsset(string bundleName, string assetName,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver);
    Sprite LoadSprite(string bundleName, string assetName,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver);
    Texture2D LoadTexture2D(string bundleName, string assetName,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver);
    void LoadScene(string bundleName, string assetName,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver, bool isAdditive = false);
    GameObject LoadClone(string bundleName, string assetName,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver);
    #endregion
    #region LoadAsync
    void LoadAssetAsync<T>(string bundleName, string assetName, Action<T> onSuccess,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver, bool isForceInterruptLoad = false)
        where T : UnityEngine.Object;
    void LoadTMP_FontAssetAsync(string bundleName, string assetName, Action<TMP_FontAsset> onSuccess,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver, bool isForceInterruptLoad = false);
    void LoadFontAsync(string bundleName, string assetName, Action<Font> onSuccess,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver, bool isForceInterruptLoad = false);
    void LoadAnimationClipAsync(string bundleName, string assetName, Action<AnimationClip> onSuccess,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver, bool isForceInterruptLoad = false);
    void LoadAnimatorOverrideControllerAsync(string bundleName, string assetName, Action<AnimatorOverrideController> onSuccess,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver, bool isForceInterruptLoad = false);
    void LoadRuntimeAnimatorControllerAsync(string bundleName, string assetName, Action<RuntimeAnimatorController> onSuccess,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver, bool isForceInterruptLoad = false);
    void LoadAudioClipAsync(string bundleName, string assetName, Action<AudioClip> onSuccess,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver, bool isForceInterruptLoad = false);
    #endregion
}

```

```

void LoadMaterialAsyn(string bundleName, string assetName, Action<Material> onSuccess,
    EAssetBundleUnloadLevel unloadLevel =
    EAssetBundleUnloadLevel.ChangeSceneOver, bool isForceInterruptLoad = false);
void LoadTextAssetAsyn(string bundleName, string assetName, Action<TextAsset> onSuccess,
    EAssetBundleUnloadLevel unloadLevel =
    EAssetBundleUnloadLevel.ChangeSceneOver, bool isForceInterruptLoad = false);
void LoadSpriteAsyn(string bundleName, string assetName, Action<Sprite> onSuccess,
    EAssetBundleUnloadLevel unloadLevel =
    EAssetBundleUnloadLevel.ChangeSceneOver, bool isForceInterruptLoad = false);
void LoadTexture2DAsyn(string bundleName, string assetName, Action<Texture2D> onSuccess,
    EAssetBundleUnloadLevel unloadLevel =
    EAssetBundleUnloadLevel.ChangeSceneOver, bool isForceInterruptLoad = false);
void LoadSceneAsyn(string bundleName, string assetName, Action onSuccess,
    EAssetBundleUnloadLevel unloadLevel =
    EAssetBundleUnloadLevel.ChangeSceneOver, bool isAdditive = false);
void LoadCloneAsyn(string bundleName, string assetName, Action<GameObject> onSuccess,
    EAssetBundleUnloadLevel unloadLevel =
    EAssetBundleUnloadLevel.ChangeSceneOver, bool isForceInterruptLoad = false);

#endregion

#region Unload
void Unload(string keyName, bool allObjects);
void UnloadAll();
#endregion

void LoadTexture2DAsyn(string name, Action<Texture2D> onSuccess, Action onFail, bool needCache);
}

```

7.4 ResourceConstant; 静态常量类，分几个不同平台来处理平台相关的资源获取存储等相关操作

```

// AssetBundle 来源枚举： 将其分为三类
public enum EAssetBunbleSourceType {
    Primary = 0,
    Hotfix = 1,
    Server = 2
}

// AssetBundle 包卸载层级
public enum EAssetBundleUnloadLevel {
    None = -1,
    Never = 0,
    LoadOver = 1,
    ChangeSceneOver = 2
}

namespace Framework.ResMgr {
    public static class ResourceConstant {
        public static IResourceLoader Loader {
            get;
            set;
        }

        public static readonly string bundleExtension = ".ab"; // assetBundle

        // static string _resourceWebRoot = "https://hotfix.server.address/";
        static string _resourceWebRoot = "http://localhost/"; // 先尝试建一个本地的服务器运行起来再说
        // 资源服务器路径: setter/getter
        public static string ResourceWebRoot {
            get { return _resourceWebRoot; }
            set { _resourceWebRoot = value; }
        }

        // 是否缓存 AssetBundle
        public static bool CacheAssetBundle = true;
        // 只读目录
        public static string AssetBundleReadOnlyRoot {
            get {
                string path = string.Empty;

                #if UNITY_EDITOR
                    path = GetEditorAssetPath();
                #elif UNITY_STANDALONE_WIN
                    path = Application.dataPath + "/StreamingAssets/";
                #elif UNITY_ANDROID
                    path = Application.dataPath + "!/assets/";
                #elif UNITY_IPHONE
                    path = Application.dataPath + "/Raw/";
                #endif
            }
        }
    }
}

```

```

        return path;
    }
}
// 可写更新目录
public static string AssetBundleCacheRoot {
    get {
        string path = string.Empty;
#if UNITY_EDITOR
        path = GetEditorAssetPath();
#elif UNITY_STANDALONE_WIN
        path = Application.dataPath + "/StreamingAssets/";
#elif UNITY_ANDROID
        path = Application.persistentDataPath;
#elif UNITY_IPHONE
        path = Application.persistentDataPath;
#endif
        return path;
    }
}

public static string AssetBundleUrl {
    get {
        string path = string.Empty;
#if UNITY_EDITOR
        path = GetEditorAssetBundleUrl();
#elif UNITY_STANDALONE_WIN
        path = "file://" + Application.dataPath + "/TempStreamingAssets/";
#elif UNITY_ANDROID
        path = Application.streamingAssetsPath;
#elif UNITY_IPHONE
        path = "file://" + Application.dataPath + "/Raw/";
#endif
        return path;
    }
}

public static string AssetBundleCacheUrl {
    get {
        string path = string.Empty;
#if UNITY_EDITOR
        path = GetEditorAssetBundleUrl();
#elif UNITY_STANDALONE_WIN
        path = "file://" + Application.dataPath + "/StreamingAssets/";
#elif UNITY_ANDROID
        path = "file://" + Application.persistentDataPath + "/";
#elif UNITY_IPHONE
        path = "file://" + Application.persistentDataPath + "/";
#endif
        return path;
    }
}

// 资源服务器路径
public static string RemoteAssetBundleUrl {
    get {
        string path = ResourceWebRoot;
#if UNITY_EDITOR
        if (!GameApplication.Instance.useLocal) {
            if (UnityEditor.EditorUserBuildSettings.activeBuildTarget == UnityEditor.BuildTarget.Android) {
                path += "Android/";
            } else if (UnityEditor.EditorUserBuildSettings.activeBuildTarget == UnityEditor.BuildTarget.iOS) {
                path += "iOS/";
            } else if (UnityEditor.EditorUserBuildSettings.activeBuildTarget == UnityEditor.BuildTarget.StandaloneWindows) {
                path += "Windows/";
            }
        } else {
            path = GetEditorAssetPath();
        }
#elif UNITY_STANDALONE_WIN
        path += "Windows/";
#elif UNITY_ANDROID
        path += "Android/";
#elif UNITY_IPHONE
        path += "iOS/";
#endif
        return path;
    }
}

// 获取文件路径
public static string GetFileUrl(string name, out bool isInCacheFile) {
    var cachePath = Path.Combine(AssetBundleCacheRoot, name);

```

```

    if (File.Exists(cachePath)) {
        isInCacheFile = true;
        return Path.Combine(AssetBundleCacheUrl, name);
    }
    if (Application.platform == RuntimePlatform.Android) {
        string filePath = Path.Combine(AssetBundleUrl, name);
        var www = new WWW(filePath);
        if (string.IsNullOrEmpty(www.error)) {
            isInCacheFile = false;
            return Path.Combine(AssetBundleUrl, name);
        }
    } else if (Application.platform == RuntimePlatform.IPhonePlayer) {
        var filePath = Path.Combine(AssetBundleReadOnlyRoot, name);
        if (File.Exists(filePath)) {
            isInCacheFile = true;
            return Path.Combine(AssetBundleUrl, name);
        }
    } else {
        var filePath = Path.Combine(AssetBundleReadOnlyRoot, name);
        if (File.Exists(filePath)) {
            isInCacheFile = false;
            return Path.Combine(AssetBundleUrl, name);
        }
    }
    isInCacheFile = false;
    return string.Empty;
}
#region EditorCode
#if UNITY_EDITOR
    static string GetEditorAssetBundleUrl() {
        string path = "file://" + GetEditorAssetPath();
        return path;
    }
    // 获取编辑器运行时资源路径
    static string GetEditorAssetPath() {
        string path = Path.Combine(Application.dataPath, "../TempStreamingAssets/" + GetPlatformFolderName());
        return path;
    }
    // 获取平台文件夹路径
    static string GetPlatformFolderName() {
        if (UnityEditor.EditorUserBuildSettings.activeBuildTarget == UnityEditor.BuildTarget.Android) {
            return "Android";
        } else if (UnityEditor.EditorUserBuildSettings.activeBuildTarget == UnityEditor.BuildTarget.iOS) {
            return "iOS";
        } else {
            return "Windows";
        }
    }
}
#endif
#endregion
}
}

```

7.5 ResourceMap : MonoBehaviour, IResourceLoader: : 是对前 IResourceLoader 公用接口类的实现

```

// 按需加载: 是对前 IResourceLoader 公用接口类的实现
public class ResourceMap : MonoBehaviour, IResourceLoader {
    private const string TAG = "ResourceMap";

    static ResourceMap _instance;
    public static ResourceMap Instance {
        get {
            return _instance;
        }
    }

    // ResourceList 初始化成功
    public Action OnInitializeSuccess;
    // Assetbundle 资源包集合: 管理字典
    // Key: AssetBundleName
    public Dictionary<string, AssetBundleSpec> assetBundleSpecs = new Dictionary<string, AssetBundleSpec>();
    // 正在下载的 AssetBundle
    public AssetBundleSpec downloadingAssetBundleSpec;
    // AssetBundle 下载队列
    public Queue<AssetLoader> downloadingAssets = new Queue<AssetLoader>();
}

```



```

// 显示加载进度条
void InitializeLoadingPanel() {}
// 分析资源文件
void AnalysisResourceList(string text, EAssetBunbleSourceType type) {
    string[] fileInfos = text.Split('\n');
    int fileInfosLength = fileInfos.Length;
    for (int i = 0; i < fileInfosLength - 1; i++) {
        string[] fileInfoParams = fileInfos[i].Split(',');
        if (fileInfoParams.Length >= 2) {
            if (!assetBundleSpecs.ContainsKey(fileInfoParams[0])) {
                AssetBundleSpec asset = new AssetBundleSpec(fileInfoParams[0],
                                                            fileInfoParams[1], int.Parse(fileInfoParams[2]), type);
                assetBundleSpecs.Add(fileInfoParams[0], asset);
            } else
                assetBundleSpecs[fileInfoParams[0]].Check(fileInfoParams[1], int.Parse(fileInfoParams[2]), type);
        } else
            DebugHelper.LogError("fileInfo error! fileInfo: " + fileInfos[i]);
    }
}
#endregion
#region Util
// 获得处理过的正确 Bundle 名
string GetFinalBundleName(string bundleName) {
    string finalBundleName = bundleName.ToLower();
    if (!bundleName.EndsWith(ResourceConstant.bundleExtension))
        finalBundleName = bundleName.ToLower() + ResourceConstant.bundleExtension;
    return finalBundleName;
}
#endregion
#region Load
// 同步加载资源
public T LoadAsset<T>(string bundleName, string assetName,
                    EAssetBundleUnloadLevel unloadLevel =
                    EAssetBundleUnloadLevel.ChangeSceneOver) where T : UnityEngine.Object {
    string finalBundleName = GetFinalBundleName(bundleName);
    if (assetBundleSpecs.ContainsKey(finalBundleName)) {
        return assetBundleSpecs[finalBundleName].LoadAsset<T>(assetName, unloadLevel);
    } else {
        DebugHelper.LogError("ResourceMap not contains " + finalBundleName, true);
        return null;
    }
}
public TMP_FontAsset LoadTMP_FontAsset(string bundleName, string assetName,
                                       EAssetBundleUnloadLevel unloadLevel =
                                       EAssetBundleUnloadLevel.ChangeSceneOver) {
    string finalBundleName = GetFinalBundleName(bundleName);
    if (assetBundleSpecs.ContainsKey(finalBundleName)) {
        return assetBundleSpecs[finalBundleName].LoadTMP_FontAsset(assetName, unloadLevel);
    } else {
        DebugHelper.LogError("ResourceMap not contains " + finalBundleName, true);
        return null;
    }
}
public Font LoadFont(string bundleName, string assetName,
                    EAssetBundleUnloadLevel unloadLevel =
                    EAssetBundleUnloadLevel.ChangeSceneOver) {
    string finalBundleName = GetFinalBundleName(bundleName);
    if (assetBundleSpecs.ContainsKey(finalBundleName)) {
        return assetBundleSpecs[finalBundleName].LoadFont(assetName, unloadLevel);
    } else {
        DebugHelper.LogError("ResourceMap not contains " + finalBundleName, true);
        return null;
    }
}
public AnimationClip LoadAnimationClip(string bundleName, string assetName,
                                       EAssetBundleUnloadLevel unloadLevel =
                                       EAssetBundleUnloadLevel.ChangeSceneOver) {
    string finalBundleName = GetFinalBundleName(bundleName);
    if (assetBundleSpecs.ContainsKey(finalBundleName)) {
        return assetBundleSpecs[finalBundleName].LoadAnimationClip(assetName, unloadLevel);
    } else {
        DebugHelper.LogError("ResourceMap not contains " + finalBundleName, true);
        return null;
    }
}
public AnimatorOverrideController LoadAnimatorOverrideController(string bundleName, string assetName,
                                                                    EAssetBundleUnloadLevel unloadLevel =
                                                                    EAssetBundleUnloadLevel.ChangeSceneOver) {

```

```

        string finalBundleName = GetFinalBundleName(bundleName);
        if (assetBundleSpecs.ContainsKey(finalBundleName)) {
            return assetBundleSpecs[finalBundleName].LoadAnimatorOverrideController(assetName, unloadLevel);
        } else {
            DebugHelper.LogError("ResourceMap not contains " + finalBundleName, true);
            return null;
        }
    }
}

public RuntimeAnimatorController LoadRuntimeAnimatorController(string bundleName, string assetName,
                                                                EAssetBundleUnloadLevel unloadLevel =
                                                                EAssetBundleUnloadLevel.ChangeSceneOver) {

    string finalBundleName = GetFinalBundleName(bundleName);
    if (assetBundleSpecs.ContainsKey(finalBundleName)) {
        return assetBundleSpecs[finalBundleName].LoadRuntimeAnimatorController(assetName, unloadLevel);
    } else {
        DebugHelper.LogError("ResourceMap not contains " + finalBundleName, true);
        return null;
    }
}

public AudioClip LoadAudioClip(string bundleName, string assetName,
                                EAssetBundleUnloadLevel unloadLevel =
                                EAssetBundleUnloadLevel.ChangeSceneOver) {

    string finalBundleName = GetFinalBundleName(bundleName);
    if (assetBundleSpecs.ContainsKey(finalBundleName)) {
        return assetBundleSpecs[finalBundleName].LoadAudioClip(assetName, unloadLevel);
    } else {
        DebugHelper.LogError("ResourceMap not contains " + finalBundleName, true);
        return null;
    }
}

public Material LoadMaterial(string bundleName, string assetName,
                              EAssetBundleUnloadLevel unloadLevel =
                              EAssetBundleUnloadLevel.ChangeSceneOver) {

    string finalBundleName = GetFinalBundleName(bundleName);
    if (assetBundleSpecs.ContainsKey(finalBundleName)) {
        return assetBundleSpecs[finalBundleName].LoadMaterial(assetName, unloadLevel);
    } else {
        DebugHelper.LogError("ResourceMap not contains " + finalBundleName, true);
        return null;
    }
}

public TextAsset LoadTextAsset(string bundleName, string assetName,
                                EAssetBundleUnloadLevel unloadLevel =
                                EAssetBundleUnloadLevel.ChangeSceneOver) {

    string finalBundleName = GetFinalBundleName(bundleName);
    if (assetBundleSpecs.ContainsKey(finalBundleName)) {
        return assetBundleSpecs[finalBundleName].LoadTextAsset(assetName, unloadLevel);
    } else {
        DebugHelper.LogError("ResourceMap not contains " + finalBundleName, true);
        return null;
    }
}

public Sprite LoadSprite(string bundleName, string assetName,
                           EAssetBundleUnloadLevel unloadLevel =
                           EAssetBundleUnloadLevel.ChangeSceneOver) {

    string finalBundleName = GetFinalBundleName(bundleName);
    if (assetBundleSpecs.ContainsKey(finalBundleName)) {
        return assetBundleSpecs[finalBundleName].LoadSprite(assetName, unloadLevel);
    } else {
        DebugHelper.LogError("ResourceMap not contains " + finalBundleName, true);
        return null;
    }
}

public Texture2D LoadTexture2D(string bundleName, string assetName,
                                EAssetBundleUnloadLevel unloadLevel =
                                EAssetBundleUnloadLevel.ChangeSceneOver) {

    string finalBundleName = GetFinalBundleName(bundleName);
    if (assetBundleSpecs.ContainsKey(finalBundleName)) {
        return assetBundleSpecs[finalBundleName].LoadTexture2D(assetName, unloadLevel);
    } else {
        DebugHelper.LogError("ResourceMap not contains " + finalBundleName, true);
        return null;
    }
}

// 同步加载场景
public void LoadScene(string bundleName, string assetName,
                       EAssetBundleUnloadLevel unloadLevel =
                       EAssetBundleUnloadLevel.ChangeSceneOver, bool isAdditive = false) {

    string finalBundleName = GetFinalBundleName(bundleName);

```

```

        if (assetBundleSpecs.ContainsKey(finalBundleName)) {
            assetBundleSpecs[finalBundleName].LoadScene(assetName, unloadLevel);
        } else {
            DebugHelper.LogError("ResourceMap not contains " + finalBundleName, true);
        }
    }
    // 同步加载 GameObject
    public GameObject LoadClone(string bundleName, string assetName,
        EAssetBundleUnloadLevel unloadLevel =
            EAssetBundleUnloadLevel.ChangeSceneOver) {
        string finalBundleName = GetFinalBundleName(bundleName);
        if (assetBundleSpecs.ContainsKey(finalBundleName)) {
            return assetBundleSpecs[finalBundleName].LoadClone(assetName, unloadLevel);
        } else {
            DebugHelper.LogError("ResourceMap not contains " + finalBundleName
                + " " + StackTraceUtility.ExtractStackTrace(), true);
            return null;
        }
    }
}
#endregion

#region LoadAsync
// 异步加载资源
public void LoadAssetAsync<T>(string bundleName, string assetName, Action<T> loadOver,
    EAssetBundleUnloadLevel unloadLevel,
    bool isForceInterrupLoad = false) where T : UnityEngine.Object {
    AssetLoader<T> assetLoader = new AssetLoader<T>();
    assetLoader.bundleName = bundleName.ToLower();
    assetLoader.assetName = assetName;
    assetLoader.onLoadOver = loadOver;
    assetLoader.unloadLevel = unloadLevel;
    AddAssetLoaderToLoadQueue(assetLoader, unloadLevel, isForceInterrupLoad);
}

public void LoadTMP_FontAssetAsync(string bundleName, string assetName, Action<TMP_FontAsset> loadOver,
    EAssetBundleUnloadLevel unloadLevel, bool isForceInterrupLoad = false) {
    AssetLoader<TMP_FontAsset> assetLoader = new AssetLoader<TMP_FontAsset>();
    assetLoader.bundleName = bundleName.ToLower();
    assetLoader.assetName = assetName;
    assetLoader.onLoadOver = loadOver;
    assetLoader.unloadLevel = unloadLevel;
    AddAssetLoaderToLoadQueue(assetLoader, unloadLevel, isForceInterrupLoad);
}

public void LoadFontAsync(string bundleName, string assetName, Action<Font> loadOver,
    EAssetBundleUnloadLevel unloadLevel, bool isForceInterrupLoad = false) {
    AssetLoader<Font> assetLoader = new AssetLoader<Font>();
    assetLoader.bundleName = bundleName.ToLower();
    assetLoader.assetName = assetName;
    assetLoader.onLoadOver = loadOver;
    assetLoader.unloadLevel = unloadLevel;
    AddAssetLoaderToLoadQueue(assetLoader, unloadLevel, isForceInterrupLoad);
}

public void LoadAnimationClipAsync(string bundleName, string assetName, Action<AnimationClip> loadOver,
    EAssetBundleUnloadLevel unloadLevel, bool isForceInterrupLoad = false) {
    AssetLoader<AnimationClip> assetLoader = new AssetLoader<AnimationClip>();
    assetLoader.bundleName = bundleName.ToLower();
    assetLoader.assetName = assetName;
    assetLoader.onLoadOver = loadOver;
    assetLoader.unloadLevel = unloadLevel;
    AddAssetLoaderToLoadQueue(assetLoader, unloadLevel, isForceInterrupLoad);
}

public void LoadAnimatorOverrideControllerAsync(string bundleName, string assetName,
    Action<AnimatorOverrideController> loadOver,
    EAssetBundleUnloadLevel unloadLevel,
    bool isForceInterrupLoad = false) {
    AssetLoader<AnimatorOverrideController> assetLoader = new AssetLoader<AnimatorOverrideController>();
    assetLoader.bundleName = bundleName.ToLower();
    assetLoader.assetName = assetName;
    assetLoader.onLoadOver = loadOver;
    assetLoader.unloadLevel = unloadLevel;
    AddAssetLoaderToLoadQueue(assetLoader, unloadLevel, isForceInterrupLoad);
}

public void LoadRuntimeAnimatorControllerAsync(string bundleName, string assetName,
    Action<RuntimeAnimatorController> loadOver,
    EAssetBundleUnloadLevel unloadLevel,
    bool isForceInterrupLoad = false) {
    AssetLoader<RuntimeAnimatorController> assetLoader = new AssetLoader<RuntimeAnimatorController>();
    assetLoader.bundleName = bundleName.ToLower();
    assetLoader.assetName = assetName;
    assetLoader.onLoadOver = loadOver;
}

```



```

        assetLoader.unloadLevel = unloadLevel;
        AddAssetLoaderToLoadQueue(assetLoader, unloadLevel, isForceInterruptLoad);
    }
    public void LoadAudioClipAsync(string bundleName, string assetName, Action<AudioClip> loadOver,
        EAssetBundleUnloadLevel unloadLevel, bool isForceInterruptLoad = false) {
        AssetLoader<AudioClip> assetLoader = new AssetLoader<AudioClip>();
        assetLoader.bundleName = bundleName.ToLower();
        assetLoader.assetName = assetName;
        assetLoader.onLoadOver = loadOver;
        assetLoader.unloadLevel = unloadLevel;
        AddAssetLoaderToLoadQueue(assetLoader, unloadLevel, isForceInterruptLoad);
    }
    public void LoadMaterialAsync(string bundleName, string assetName, Action<Material> loadOver,
        EAssetBundleUnloadLevel unloadLevel, bool isForceInterruptLoad = false) {
        AssetLoader<Material> assetLoader = new AssetLoader<Material>();
        assetLoader.bundleName = bundleName.ToLower();
        assetLoader.assetName = assetName;
        assetLoader.onLoadOver = loadOver;
        assetLoader.unloadLevel = unloadLevel;
        AddAssetLoaderToLoadQueue(assetLoader, unloadLevel, isForceInterruptLoad);
    }
    public void LoadTextAssetAsync(string bundleName, string assetName, Action<TextAsset> loadOver,
        EAssetBundleUnloadLevel unloadLevel, bool isForceInterruptLoad = false) {
        AssetLoader<TextAsset> assetLoader = new AssetLoader<TextAsset>();
        assetLoader.bundleName = bundleName.ToLower();
        assetLoader.assetName = assetName;
        assetLoader.onLoadOver = loadOver;
        assetLoader.unloadLevel = unloadLevel;
        AddAssetLoaderToLoadQueue(assetLoader, unloadLevel, isForceInterruptLoad);
    }
    public void LoadSpriteAsync(string bundleName, string assetName, Action<Sprite> loadOver,
        EAssetBundleUnloadLevel unloadLevel, bool isForceInterruptLoad = false) {
        AssetLoader<Sprite> assetLoader = new AssetLoader<Sprite>();
        assetLoader.bundleName = bundleName.ToLower();
        assetLoader.assetName = assetName;
        assetLoader.onLoadOver = loadOver;
        assetLoader.unloadLevel = unloadLevel;
        AddAssetLoaderToLoadQueue(assetLoader, unloadLevel, isForceInterruptLoad);
    }
    public void LoadTexture2DAsync(string bundleName, string assetName, Action<Texture2D> loadOver,
        EAssetBundleUnloadLevel unloadLevel, bool isForceInterruptLoad = false) {
        AssetLoader<Texture2D> assetLoader = new AssetLoader<Texture2D>();
        assetLoader.bundleName = bundleName.ToLower();
        assetLoader.assetName = assetName;
        assetLoader.onLoadOver = loadOver;
        assetLoader.unloadLevel = unloadLevel;
        AddAssetLoaderToLoadQueue(assetLoader, unloadLevel, isForceInterruptLoad);
    }
}
// 异步加载场景
public void LoadSceneAsync(string bundleName, string assetName, Action success,
    EAssetBundleUnloadLevel unloadLevel, bool isAdditive = false) {
    Debug.Log(TAG + ": LoadSceneAsync()");
    SceneLoader sceneLoader = new SceneLoader();
    sceneLoader.bundleName = bundleName.ToLower();
    sceneLoader.assetName = assetName;
    sceneLoader.onLoadOver = success;
    sceneLoader.unloadLevel = unloadLevel;
    sceneLoader.isAdditive = isAdditive;
    AddAssetLoaderToLoadQueue(sceneLoader, unloadLevel, false);
}
// 异步加载 GameObject
public void LoadCloneAsync(string bundleName, string assetName, Action<GameObject> success,
    EAssetBundleUnloadLevel unloadLevel, bool isForceInterruptLoad = false) {
    GameObjectLoader gameObjectLoader = new GameObjectLoader();
    gameObjectLoader.bundleName = bundleName.ToLower();
    gameObjectLoader.assetName = assetName;
    gameObjectLoader.unloadLevel = unloadLevel;
    gameObjectLoader.onCloneOver = success;
    AddAssetLoaderToLoadQueue(gameObjectLoader, unloadLevel, isForceInterruptLoad);
}
// 添加 AssetLoader 到加载队列中
void AddAssetLoaderToLoadQueue(AssetLoader assetLoader,
    EAssetBundleUnloadLevel unloadLevel, bool isForceInterruptLoad) {
    Debug.Log(TAG + ": AddAssetLoaderToLoadQueue()");
    Debug.Log(TAG + " isForceInterruptLoad: " + isForceInterruptLoad);
    Debug.Log(TAG + " assetLoader.FinalBundleName: " + assetLoader.FinalBundleName);
    Debug.Log(TAG + " (assetBundleSpecs.ContainsKey(assetLoader.FinalBundleName)): "
        + (assetBundleSpecs.ContainsKey(assetLoader.FinalBundleName)));
    if (assetBundleSpecs.ContainsKey(assetLoader.FinalBundleName)) {

```

```

        assetLoader.onAssetLoadEnd = LoadNextAsset;
        var assetLoaderSpec = assetBundleSpecs[assetLoader.FinalBundleName];
        assetLoader.Spec = assetLoaderSpec;
        if (downloadingAssetBundleSpec == null) {
            downloadingAssetBundleSpec = assetLoaderSpec;
            downloadingAssetBundleSpec.TryLoadAsyn(assetLoader, unloadLevel);
        } else
            downloadingAssets.Enqueue(assetLoader);
        if (!downloadingAssetBundleSpecs.ContainsKey(assetLoaderSpec.Name)) {
            downloadingAssetBundleSpecs[assetLoaderSpec.Name]
                = new AssetBundleDownloadReference() { spec = assetLoaderSpec, referenceCount = 1 };
        } else {
            downloadingAssetBundleSpecs[assetLoaderSpec.Name].referenceCount++;
        }
    } else {
        DebugHelper.LogError("ResourceMap not contains " + assetLoader.FinalBundleName, true);
    }
}
// 加载下一个 AssetLoader
void LoadNextAsset(AssetLoader al) {
    if (downloadingAssetBundleSpec != null && downloadingAssetBundleSpecs.ContainsKey(downloadingAssetBundleSpec.Name)) {
        AssetBundleDownloadReference r = downloadingAssetBundleSpecs[downloadingAssetBundleSpec.Name];
        r.referenceCount--;
        if (r.referenceCount <= 0)
            downloadingAssetBundleSpecs.Remove(downloadingAssetBundleSpec.Name);
    }
    if (downloadingAssets.Count > 0) {
        AssetLoader assetLoader = downloadingAssets.Dequeue();
        downloadingAssetBundleSpec = assetLoader.Spec;
        downloadingAssetBundleSpec.TryLoadAsyn(assetLoader,
                                                downloadingAssetBundleSpec.UnloadLevel,
                                                downloadingAssetBundleSpec.IsForceInterruptLoad);

        if (onNextAssetLoader != null)
            onNextAssetLoader(assetLoader);
    } else
        downloadingAssetBundleSpec = null;
}
}
#endregion
#region GetVideoClipURL
public void GetFileURLAsyn(string name, Action<string> onSuccess, Action onFail, bool needCache = true) {
    if (!FileHelp.IsFileExists("VideoClip/" + name)) {
        string url = Path.Combine(ResourceConstant.RemoteAssetBundleUrl + "VideoClip", name);
        Debug.Log("url: " + url);
        HttpHelper.Instance.Get(url, (request) => {
            if (needCache) {
                if (!Directory.Exists(ResourceConstant.AssetBundleCacheRoot + "/VideoClip")) {
                    Directory.CreateDirectory(ResourceConstant.AssetBundleCacheRoot + "/VideoClip");
                }
                FileStream fileStream = File.Create(Path.Combine(ResourceConstant.AssetBundleCacheRoot + "/VideoClip",
                                                                    byte[] data = request.downloadHandler.data;
                fileStream.Write(data, 0, data.Length);
                fileStream.Close();
                fileStream.Dispose();
                if (onSuccess != null)
                    onSuccess("VideoClip/" + name);
            }
        }, (request) => {
            if (onFail != null)
                onFail();
        });
    } else {
        if (onSuccess != null)
            onSuccess("VideoClip/" + name);
    }
}
}
#endregion
#region LoadTexture
// 异步从网上加载图片
public void LoadTexture2DAsyn(string name, Action<Texture2D> onSuccess, Action onFail, bool needCache = true) {
    if (FileHelp.IsFileExists("DownloadTexture/" + name)) {
        CoroutineHelper.StartCoroutine(LoadTexture2DFromCacache(name, onSuccess, onFail));
    } else {
        string url = Path.Combine(ResourceConstant.RemoteAssetBundleUrl, name);
        HttpHelper.Instance.GetTexture(url, (texture, request) => {
            if (needCache) {
                if (!Directory.Exists(ResourceConstant.AssetBundleCacheRoot + "/DownloadTexture")) {
                    Directory.CreateDirectory(ResourceConstant.AssetBundleCacheRoot + "/DownloadTexture");
                }
            }
        });
    }
}
}

```

```

        FileStream fileStream = File.Create(Path.Combine(ResourceConstant.AssetBundleCacheRoot + "/DownloadTexture2D",
        byte[] data = request.downloadHandler.data;
        fileStream.Write(data, 0, data.Length);
        fileStream.Close();
        fileStream.Dispose();
    }
    if (onSuccess != null)
        onSuccess(texture);
}, (request) => {
    if (onFail != null)
        onFail();
});
}
}
WWW thisWWW;
IEnumerator LoadTexture2DFromCacache(string name, Action<Texture2D> onSuccess, Action onFail) {
    string url = Path.Combine(ResourceConstant.AssetBundleCacheUrl + "/DownloadTexture", name);
    thisWWW = new WWW(url);
    yield return thisWWW;
    if (thisWWW == null) {
        DebugHelper.LogError("Download Texture2D " + url + " thisWWW == null", true);
        if (onFail != null) {
            onFail();
        }
    } else if (!string.IsNullOrEmpty(thisWWW.error)) {
        DebugHelper.LogError("Download Texture2D " + thisWWW.error + " url: " + url, true);
        yield return new WaitForEndOfFrame();
        if (thisWWW != null) {
            thisWWW.Dispose();
            thisWWW = null;
            if (onFail != null)
                onFail();
        }
    } else {
        if (onSuccess != null)
            onSuccess(thisWWW.texture);
    }
}
}
#endregion
#region Unload
// 卸载
public void Unload(string keyName, bool allObjects) {
    string finalBundleName = GetFinalBundleName(keyName);
    if (assetBundleSpecs.ContainsKey(finalBundleName))
        assetBundleSpecs[finalBundleName].Unload(allObjects);
}
// 全部卸载
public void UnloadAll() {
    var tempData = downloadingAssets.Where(p => p.Spec.UnloadLevel ==
        EAssetBundleUnloadLevel.Never);
    downloadingAssets = new Queue<AssetLoader>(tempData);
    downloadingAssetBundleSpecs.Clear();
    foreach (var assetBundle in assetBundleSpecs) {
        if (assetBundle.Value.UnloadLevel ==
            EAssetBundleUnloadLevel.ChangeSceneOver) {
            assetBundle.Value.Unload(false);
        }
    }
    if (downloadingAssetBundleSpecs == null)
        LoadNextAsset(null);
}
}
#endregion

void Update() {
    checkSaveTimes += Time.deltaTime;
    if (checkSaveTimes > 5) {
        checkSaveTimes = 0;
        if (haveCachedChanged) {
            haveCachedChanged = false;
            SaveHotFixResourceList();
        }
    }
}

public void SaveHotFixResourceList() {
    if (ResourceConstant.CacheAssetBundle) {
        DateTime regTime = DateTime.Now;
        StringBuilder stringBuilder = new StringBuilder();
        string hotFixListPath = ResourceConstant.AssetBundleCacheRoot + "/HotFixList.txt";
        FileStream fileStream;
    }
}

```

```

        if (!File.Exists(hotFixListPath)) {
            fileStream = File.Create(hotFixListPath);
        } else {
            fileStream = new FileStream(hotFixListPath, FileMode.Truncate);
        }
        string ts = string.Empty;
        var ec = new UTF8Encoding();
        foreach (var assetBundleSpec in assetBundleSpecs) {
            if (assetBundleSpec.Value.IsCached && assetBundleSpec.Value.AssetBundleSourceType == EAssetBunbleSourceType)
                ts = assetBundleSpec.Value.Name + "," + assetBundleSpec.Value.MD5 + "," + assetBundleSpec.Value.Length;
            var bytes = ec.GetBytes(ts);
            fileStream.Write(bytes, 0, bytes.Length);
        }
        fileStream.Close();
        fileStream.Dispose();
        DebugHelper.LogError("Save HotFixList.txt cost " + (DateTime.Now - regTime).TotalMilliseconds, true);
    }
}

public void OnCachedAssetBundleSpec(AssetBundleSpec spec) {
    haveCachedChanged = true;
}

// IOS 平台设置
void IOSSetting() {
#if UNITY_IOS
    UnityEngine.iOS.Device.SetNoBackupFlag(Application.persistentDataPath);
#endif
}
}

```

8 SDK: 第三方库的接入, 比如想要使用微信登录, 或是使用了 google 账户登录等

8.0.1 NativeHelper: 原生交互类

```

// 原生交互类
public class NativeHelper {
    // 微信登录, google/facebook/linkedin 登录方式一致吗? 需要分别接入他们各方提供的 SDK
    public static void WeChatLogin() {
        if (Application.platform == RuntimePlatform.WindowsPlayer) {
        } else if (Application.platform == RuntimePlatform.Android) {
        } else if (Application.platform == RuntimePlatform.IPhonePlayer) {
            //IOSNativeInterface.WeChatLogin();
        }
    }
}

```

9 Framework.Util 各种帮助类

9.1 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 {
            // 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, 更新控件的名字
            // 是因为控件具备生命周期, 所以实例就是 MonoBehaviour 的继承类了吗? 好像是这样
        }
    }
}

```

```

        if (_instance == null) {
            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);
}
}

```

9.2 static class JsonSerializer: 静态帮助类，帮助将网络下载的资源包解析为相应的封装类型

```

public static class JsonSerializer {
    public static JsonValue Deserialize(string jsonString) {
        try {
            if (string.IsNullOrEmpty(jsonString))
                return new JsonObject();
            var re = JsonValue.Parse(jsonString);
            return re;
        } catch (Exception ex) {
            DebugHelper.LogError(string.Format("{0}{1}", jsonString, ex), true);
            return null;
        }
    }
}
}

```