

Unity Android SDK/NDK 俄罗斯方块砖 3D 小游戏

deepwaterooo

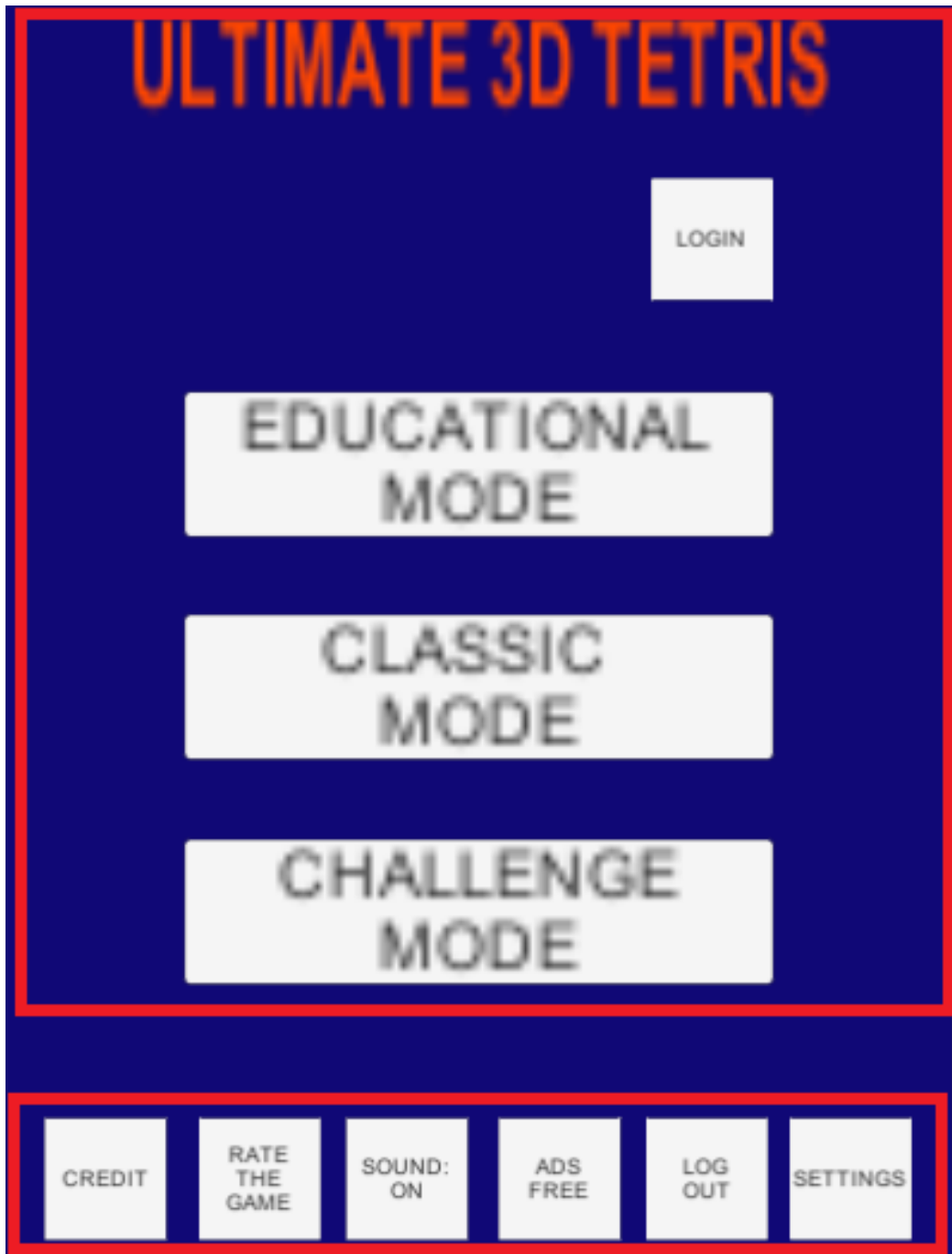
September 29, 2022

Contents

1 游戏基础框架的快速链接	1
2 把原理弄懂	2
3 环境弄得比较好的包括：	3
4 ILRuntime 库的系统再深入理解	3
4.1 ILRuntime 基本原理	3
4.2 ILRuntime 热更流程	4
4.3 ILRuntime 主要限制	5
4.4 ILRuntime 启动调试	7
4.5 线上项目和资料	7

1 游戏基础框架的快速链接

- 将游戏的绝大部分逻辑放到热更新程序包下执行，以便实现随时热更；
- 暂时不考虑服务器，所以先弄个本地服务器试运行着
- 把基本的 unity 游戏逻辑与 ILRuntime + MVVM 的唯一一个（如果我需要多个场景，那么我需要添加很多场景切换的逻辑，所以最简单的办法就是游戏自始至终只使用一个场景，其它所有变更的都只是 Panel View UI。）场景试着尽快运行起来
- 打包第一个场景：好像是两个前后相差几个的游戏引擎不太兼容，打包的预设不识别，会再看一看（先粗糙地封了一整个场景，会把可能热更新的拆下来拆小一点儿，晚些时候）



-
-

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

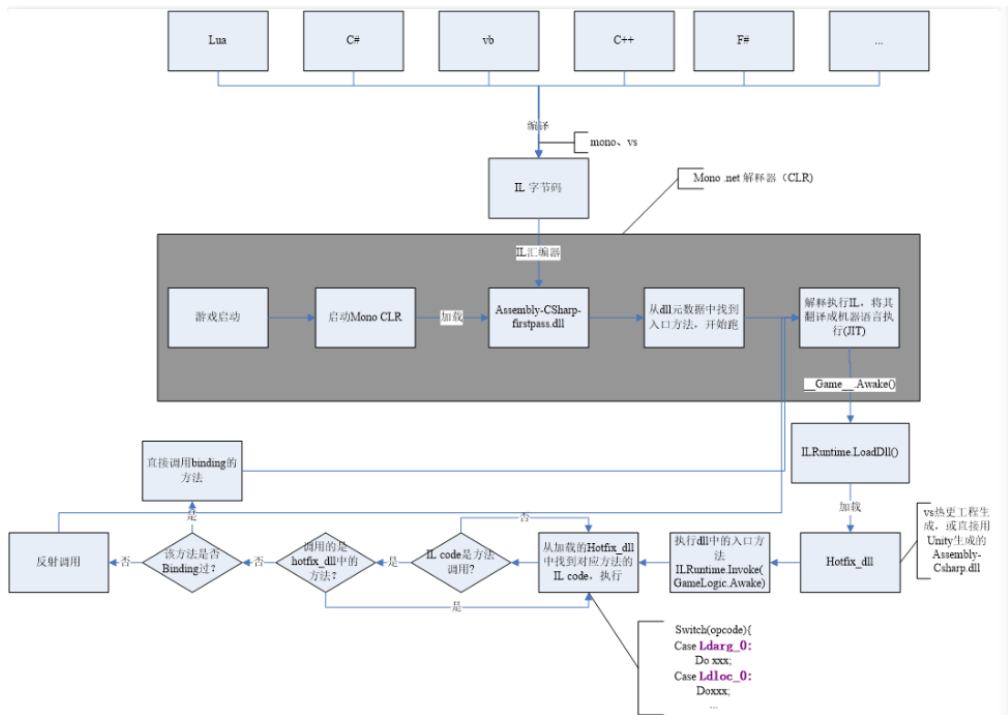
3 环境弄得比较好的包括：

- 电脑的配置有限，文件稍微大一点儿的时候已经不太好处理了；所以不得不分割成多个小文件
- 几年过去了，ILRuntime 已经不是最新最前沿的热更新技术，成为别人更新技术的一个子模块，所以还是自己再搜索找一下有没有更方便的热更新实现方法（若是不得，我就在自己游戏里实现 ILRuntime + MVVM 实现视图等的更新）
- 这一两天作必要的文献研究，确定哪个大的模块版块需要实现或是修改优化，列个大致计划，把它们一一完成；希望截止这个周末周六周日能够把这个部分确定得相对精确
- 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!!!

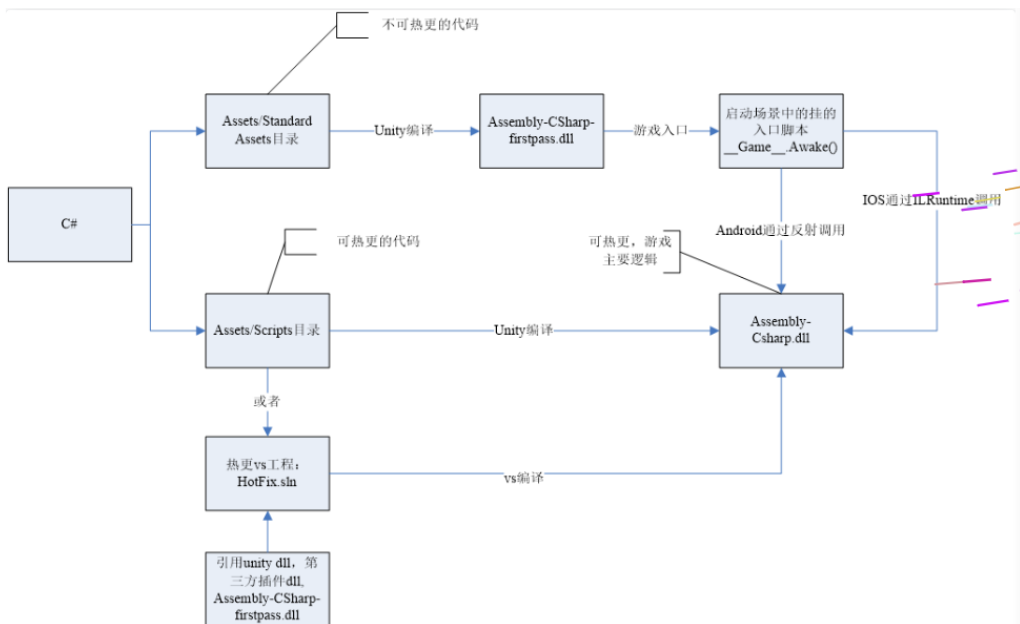
4 ILRuntime 库的系统再深入理解

4.1 ILRuntime 基本原理

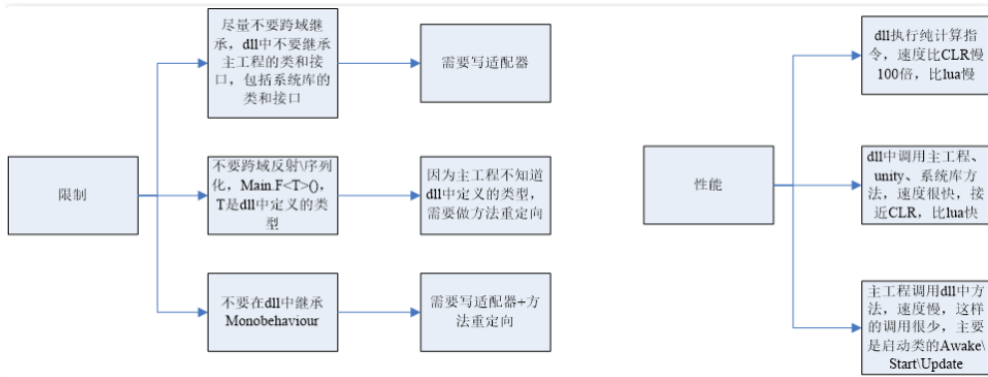
- ILRuntime 借助 Mono.Cecil 库来读取 DLL 的 PE 信息，以及当中类型的所有信息，最终得到方法的 IL 汇编码，然后通过内置的 IL 解译执行虚拟机来执行 DLL 中的代码。IL 解释器代码在 ILInterpreter.cs，通过 Opcode 来逐语句执行机器码，解释器的代码有四千多行。



4.2 ILRuntime 热更流程



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(ILIntepreter __intp, StackObject* __esp,
                                       List<object> __mStack, CLRMethod __method, bool isNewObj) {
    ILRuntime.Runtime.Enviorment.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 = ILIntepreter.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 = ILIntepreter.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>