

Unity Android SDK/NDK 俄罗斯方块砖 3D 小游戏

deepwaterooo

March 15, 2023

Contents

1 ET-EUI 服务器 + ET-Network-Module 非 ET 框架客户端的网络模块 + Unity ILRuntime + MVVM 热更新程序域逻辑: 项目链接	1
1.1 目前项目存在的问题: 需要各种版本升级: LangVersion .Csharp 语言版本, .NET-Framework 版本, 第三方库版本【包括 ILRuntime UniRx 等等】	1
1.2 版本升级步骤: Unity 游戏端, 项目主工程四个项目, 编译成功	2
1.2.1 UniRx 第三方库:	2
1.2.2 FingerGestures 和其它	3
1.3 Hotfix.sln 里一个热更新项目的构建	3
1.4 文件服务器: ET 框架中文件服务器是一个用 go 语言实现的最简单的静态文件服务器	3
1.4.1 其它版本中的实现, 将来可以参考的项目及源码	3
1.4.2 斗地主游戏文件服务器参考	4
1.5 游戏项目的后期优化结尾	4
2 SDK flow 的基本流程设计:	5
3 游戏基本场景设计	6
4 大致进展计划	28
5 进展过程与基本问题	31
6 把原理弄懂	32
7 几种不同热更新模式的探讨	33
7.1 HybridCLR——划时代的 Unity 原生 C# 热更新技术: IL2CPP 与热更新	33
8 环境弄得比较好的包括:	34
9 ILRuntime 库的系统再深入理解	35
9.1 ILRuntime 基本原理	35
9.2 ILRuntime 热更流程	36
9.3 ILRuntime 主要限制	36
9.4 ILRuntime 启动调试	39
9.5 线上项目和资料	39
10 remember necessary positoins	39

11 ILRuntime 的研究	39
11.1 工程运行的入口	39
11.1.1 HotFixILRuntime	39
11.1.2 LoadAssembly(System.IO.Stream stream, System.IO.Stream symbol, ISymbolReaderProvider symbolReader)	40
11.1.3 ReadModule(stream)	40
12 热更新资源加载的过程	47
12.1 AssetBundleList.txt	47

1 ET-EUI 服务器 + ET-Network-Module 非 ET 框架客户端的网络模块 + Unity ILRuntime + MVVM 热更新程序域逻辑：项目链接

- 前两天测试过：ET-Network-Module 客户端可以与 ET-EUI 服务器通信成功；
- 现链接 ET-Network-Module 客户端的网络模块与客户端 ILRuntime 热更新程序域逻辑。【初步版本先这样，网络模块的功能比较少：主要是方便下载热更新程序域的热更新资源包；顺便要求用户必须注册登录而已。不带其它网络模块功能】
- 现只把游戏的客户端还原回所有的主要逻辑，全部整合入 ILRuntime 热更新程序域。Unity 域只保留了网络通信模块与游戏入口类。
- 【整合登录注册文件服务器逻辑】：这个整合的注册登录服，与文件服务器，服务器本身很简单。那么现在客户端整合的逻辑也很简单：
 - 可以在 GameApplication 中加入这个登录注册的界面，界面提示用户，要注册登录，会写入数据库，否则无法真正进入游戏。当用户点击登录按钮，客户端就发出了向服务器的登录注册申请
 - 服务器端的逻辑是 ET 框架中完备的：注册登录用检查 MongoDB 数据库用户帐户管理中，是否有当前用户名的纪录：
 - * 有用户名密码匹配就登录成功，Realm 注册登录服为客户端申请分配到一个网关服，并把网关服的相关住处返回给当前客户端，当前客户端再向网关服建立连接。当收到网关服认证，就连接成功，**客户端【调用热更新程序域入口函数，进入游戏】**；
 - * 若数据库里没有当前用户名，就自动注册一个用户帐户，注册登录成功。接下来其它逻辑如上。
 - * 若数据库里记有当前用户为黑名单，就返回错误，当前用户无法进入热更新程序域，就无法游戏玩耍了

1.1 目前项目存在的问题：需要各种版本升级：LangVersion .Csharp 语言版本，.NETFramework 版本，第三方库版本【包括 ILRuntime UniRx 等等】

- 感觉去年再次接触这个项目的时候，还比较弱小，很多地方，应该是一目了然，应该使用最新的 Unity Visual studio 的新版本来实现和完成。但当时的自己相对来说，很多地方理解不够，觉得能够将就就把项目运行起来就不错了。对自己的要求比较低，项目完成在相对陈旧版本的基础上，Visual Studio 用了 2019，Unity 2018.3.7 .NETFramework v3.5。
- 现在因为要整合使用 ET 框架至少 6.0 吧，就需要使用 LangVersion 8.0,.NET 6.0，所以现在需要把客户端项目工程，热更新项目工程升给版本，必要的时候必要情况下，Unity 里某些场景或是预设等可能会有部分需要重做。这是去年使用相对古老一点儿版本开发项目遗留的

问题，但这次需要解决。要不然，ET 框架 6.0 版本的服务器，与 ET 框架网络通信的客户端 ET-Network-Module 网络模块，与客户端游戏项目工程，游戏项目热更新程序域整合不起来。但现在连接，就是需要，必须得使这几个模块能够链接整合。

- 把 windows 下的 `pyim:toggle-input-method` 和光标给调出来【终于调出来了】再也不受这个困扰了。爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!
- 那么现在的整合步骤是：【今天下面的这个步骤，能够完成多少就完成多少，但是应该还是会有不错的进展的】如下章节

1.2 版本升级步骤：Unity 游戏端，项目主工程四个项目，编译成功

- 使用与 ET-Network-Module 同样的 Unity 版本【确保到时服务器与客户端是可以正常成功通信的】来先把项目能在新的 unity 版本下运行起来，将主工程中四个项目编译成功【这个步骤完成】

1.2.1 UniRx 第三方库：

- UniRx 的版本升级：感觉这个库有点儿过时了，基本可以不用它了。人们现在似乎已经不再提它了。要不要把它直接去掉呢？
 - UniRx 是 Unity Reactive Extensions 的缩写，意思是 Unity 的响应式扩展集。重写了 .Net 的响应式扩展。
 - .Net 官方的 Rx 很棒，但是在 Unity 中无法使用，并且与 IOS 的 IL2CPP 有兼容性问题。这个库解决了这些问题并且添加了一些 Unity 专属的工具类。
 - 支持的平台有：PC/Mac/Android/iOS/WP8/WindowsStore/等等，并且支持 Unity4.6 之后的所有版本。
 - Reactive Extensions 指的就是：观察者模式 + Linq。
- 响应式
 - 响应式是什么意思：一旦 A 做了事情，B 就得到了通知。
 - 为什么 **UniRx 是响应式**：因为 UniRx 通过观察者模式实现了这种响应。
- 为什么使用 Rx：
 - 拿网络操作进行举例：网络操作需要用到 WWW 和 Coroutine。但是使用 Coroutine 会遇到下面列问题
 - * 1. 协程不能有返回值，因为返回类型必须是 IEnumerator
 - * 2. 协程不能处理异常，因为 yield return 语句没办法被 try-catch
 - * 这样就会造成代码大面试的强耦合。
 - 游戏循环 (every Update, OnCollisionEnter, etc), 传感器数据 (Kinect, Leap Motion, VR Input, etc.) 都是事件。Rx 将事件转化为响应式的序列，通过 LINQ 操作可以很简单地组合起来，还支持时间操作。
 - Unity 通常是单线程，但是 UniRx 可以让多线程更容易。
 - UniRx 可以简化 uGUI 的编程，所有的 UI 事件 (clicked, valuechanged, etc) 可以转化为 UniRx 的事件流。
- 当把 Assets/Plugins/ILRuntime/Generated 去掉之后，所有的错误消失了。再解决下一个项目中存在的编译问题。【亲爱的表哥，活宝妹一定要嫁的亲爱的表哥!!! 活宝妹就是一定要嫁给亲爱的表哥!!!】

1.2.2 FingerGestures 和其它

Assets\Plugins\FingerGestures\Editor\FingerGestures\PointCloudGestureTemplateInspector.cs(100,25): error CS0117: 'Handles' does not contain a definition for 'CircleCap'
Assets\Plugins\FingerGestures\Editor\FingerGestures\PointCloudGestureEditor.cs(120,25): error CS0117: 'Handles' does not contain a definition for 'CircleCap'

- 这两个错误，是第三方库包 FingerGestures 里面的，可能与新版本 Unity 之间的适配问题。下午来解决这些问题。【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!!】
- 这里的做法简单粗暴：直接将出错的两行去掉了，因为项目中未必真的用到这两行。等出错的时候，再 debug 回来。
- 这样，游戏主工程中的四个项目就可以在最新版本，确保可以运行的版本下编译成功了。
- trunk.sln 里主程序域里四个项目的整合完整

1.3 Hotfix.sln 里一个热更新项目的构建

- 以前使用的是.NETFrameworkv3.5 现在需要将它升级。升级到了 v6.0. 居然是不费吹灰之力!!!! 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥】
- 为什么几个月前的自己，会觉得升级版本那么困难？为什么现在升级一下就不费吹灰之力了？K，一个人的强大也是这么一点儿一点儿累积起来的吗？这些时时与过去前几个月的自己的对比，或是几周前，或是几天前的自己的对比，深切感受自己学习专业上的成长进步，都是支持支撑自己努力学习达到自己的目标【活宝妹从来都是为能够嫁给亲爱的表哥，而努力学习读书】的永不丢失的动力。
- 那么现在的工作就是：把自己的想到的，很容易实现的与 ET 服务器对接的 ET-Network-Module 网络模块整合进来，进行测试。【希望今天能够快速地做完这一块儿，不难】
- 网络模块的登录注册，客户端与网关服务器链接成功后，自动由 unity 程序域调用热更新域入口程序，进入游戏热更新进入游戏，链接完成，轻松完成任务。【亲爱的表哥，活宝妹一定要嫁的亲爱的表哥!!!】

1.4 文件服务器：ET 框架中文件服务器是一个用 go 语言实现的最简单的静态文件服务器

1.4.1 其它版本中的实现，将来可以参考的项目及源码

- 把框架中这块的逻辑理解透彻。需要自己的游戏可以从文件服务器中下载资源包，热更新程序包，来实现客户端与服务器的动态交互，实时热更新。
- 本来是想要拉取 ET 框架的最新版本【现只拉取到 ET 框架的 release7.2 版本在本地，不曾推送到仓库】来运行，并阅读能够构建好源码来深入理解文件服务器中与热更新资源包程序包相关的原理。【但目前因为我所使用的 Unity 使用渲染管道与原项目 Unity 所使用版本的不同，目前没能安装和构建 Unity 客户端项目成功】
- 但是这个 ET 框架的问题是：先前我至少是用 master 分支，示例场景 Init 在自己的机制上运行成功过。那么也就是可能，后来源仓库的提交删除了某些相关文件，当自己的仓库与尖仓库同步之后，这个项目无法再次运行成功。我可以简单去 checkout 一个不包含他们最近几次提交的分支来运行。但暂时就先放一下。
- 暂时就仍去读【ET-EUI 框架中的文件服务器相关的逻辑】。若能够感觉到明显不同或是有欠缺，再回去查 ET-release7.2 分支中是否有作更新。接下来进一步完成自己项目中文件服务器相关的工作。【ET-EUI 是个裁剪了的前端 UI 框架，可能没有文件服务器】

1.4.2 斗地主游戏文件服务器参考

- 【斗地主游戏文件服务器参考】这个游戏在自己所台式机上也是可以运行成功，并且自己玩过好多次的。那就先看一下它的文件服务器。
- FileServer/FileServer.exe 可惜好像只是用 go 语言写的一个极简单的静态文件服务器。**【极简实现：功能极为简单，但是满足需求】**
- 既然可以用 go 语言一个文件一行代码搞定一个文件服务器，那么其实，我自己的极简文件服务器也同样应该可以用一个文件一行代码来搞定自己所需要的这个极简热更新资源包文件服务器。可能就是过程中涉及一点儿程序集什么之类的管理
- 没有理解的是 **http.ListenAndServe()** 是如何应对客户端索要相关文件的下载的？这个 go 语言指定说指定文件目录位置里的文件都充当是服务器文件；可是客户端想要下载文件时的逻辑，去找出来看一下 **【整理客户端的起始逻辑，当程序集等加载好，什么地方需要资源包，需要比对的地方，就是与服务器端交互的地方，从客户端的 Init.cs 起始步骤开始找】**
- **【FileServer/FileServer.go】** 这个文件，就是使用 go 语言的极简编程，将一个服务器端的文件目录地址，变成了一个静态文件服务器。其它文件服务器相关的资源包的上载与下传，都是由 C# 语言客户端实现的。
- **【热更新资源包的上载上传到服务器的逻辑：class BuildEditor 类】** 这个好像是游戏客户端的 Editor 里的一键打包上传服务器什么之类的工具类定义里面。再看一遍。
- **【客户端下载资源包的逻辑：class BundleDownloaderComponent 类】** ET 框架中是一个资源包下载的专用组件，负责与文件服务器 MD5 码比对以及资源包的下载等。自己的项目，不是这种专用组件，若真使用一个 go 语言一个文件一行代码搞定的个文件服务器，可能需要必要的比对适配，以满足自己游戏逻辑的需要。感觉框架里的这个类，看得比较懂了。
- 客户端的下载服务器码表文件并比对，标记需要下载的资源包，并异步下载到本地。这个过程大致懂了。晚上会再看下 Editor 工具里上传资源包到服务器的实现细节。晚上回来若有时间会更改自己的并测试。
- **【接下来测试】** 进入热更新程序域时，当前先前的资源包从来都是用本地的资源包测试的。现在需要服务器端添加一个热更新资源包文件服务器配置，需要客户端从服务器下载更新资源包，进入游戏。这里的逻辑需要测试。会今天下午或是晚上来测试这个环境。
- 下午的时间，会主要用来整理和优化自己的游戏。**【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥，爱表哥，爱生活!!!】**

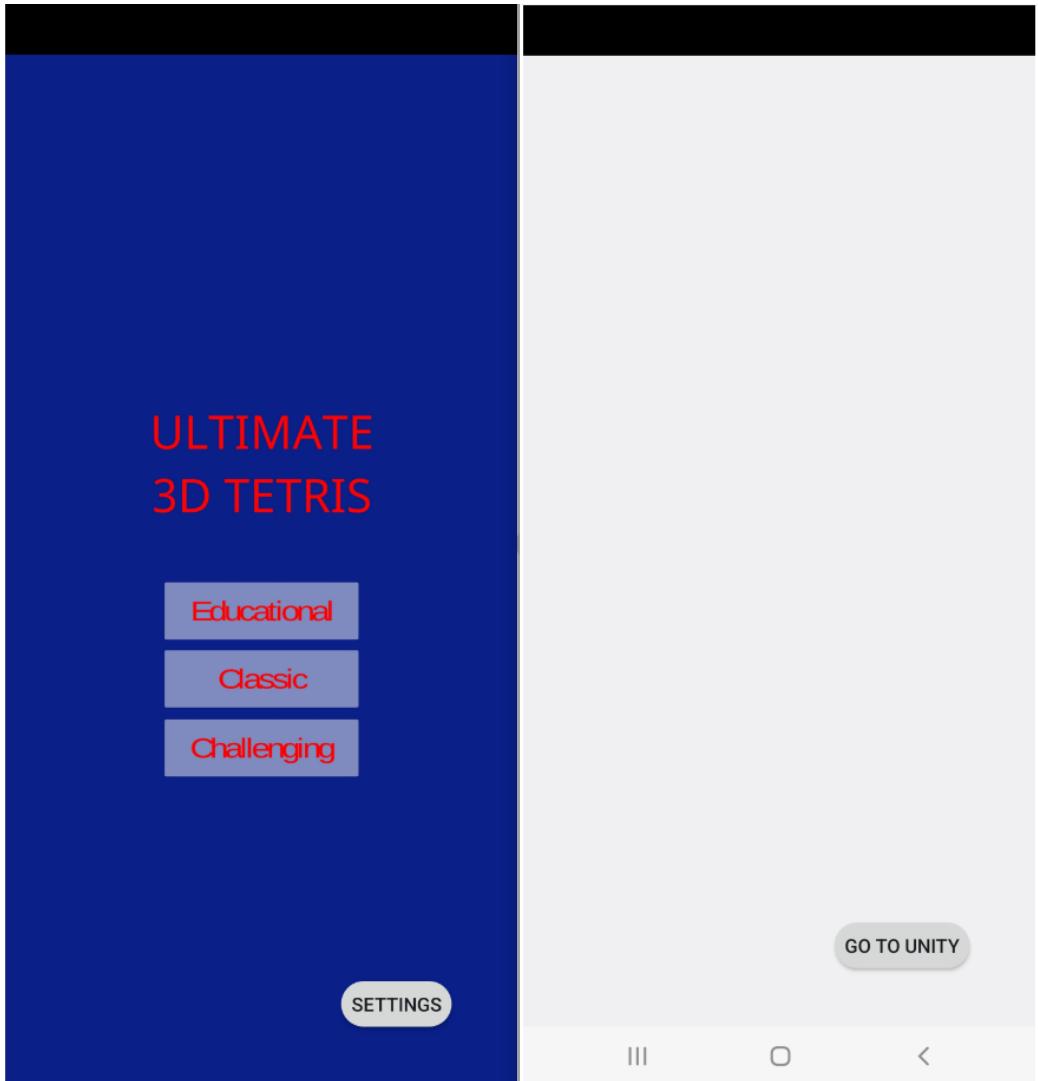
1.5 游戏项目的后期优化结尾

- 感觉现在是具备再进行最后一次大型重构，适配 ET 双端热更新框的能力。但游戏本身并非多人网络游戏，可以带来的好处是辅助功能：用户配置同步到云端；用户各个模式的游戏进展数据同步到云端等。但因为游戏本身是单人单机游戏，觉得这个游戏就要一个最简单的服务器，再进化一下后期的优化，准备结束这个游戏。下一个游戏的时候，可以考虑写一个多人网络游戏。
- 那么接下来，该如何优化这个游戏，考虑如何上线呢？会利用接下来下午和晚上的时间查阅相关步骤，绝大部分的功能模块就这么定下来，准备最后的工作，放上线。

2 SDK flow 的基本流程设计:

- 在这里起参考作用的是这个链接:<https://blog.csdn.net/u014361280/article/details/91888091>

- 把里面的几种方法想透，也有试过导出游戏工程在安卓中构建，但是因为一个自己还不是很能理解的 bug (就是说运行时，它找不到 AOT 的相关编译码，有试用 mono 导和用 il2cpp 导，都出现同样的 bug). 后来把这里面的思路想透，直接实现在 unity 引擎中打.apk 包，运行出上面的效果，实现运行出来，感觉很开心.

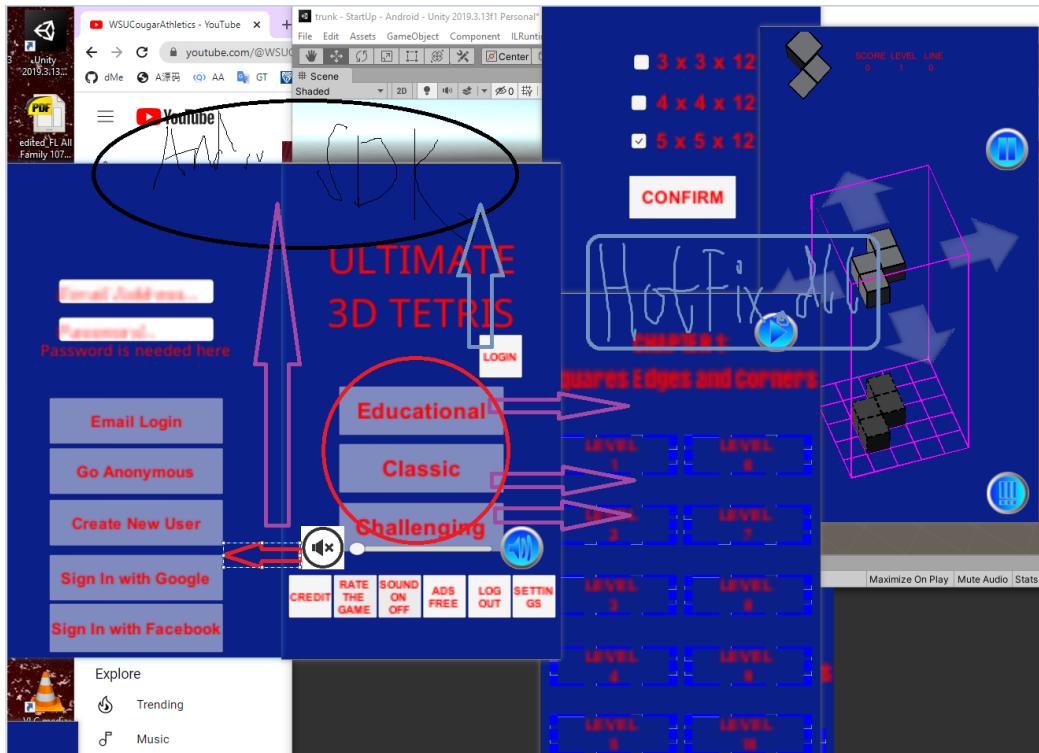


- 这里面的小问题包括：当用户正在玩游戏，但感觉音量大了一点儿，想直接去应用中调音量，但是进入安卓 SDK，游戏会丢失。这里有个像上份工作中电动车，当用户出去购物时，希望车的配置是被保存保留的。也就是说，(如果可以,) 通过操作安卓活动的启动方式，或者是游戏本身为用户帮助用户保存游戏状态，当从安卓 SDK 中切回来的时候，不是回到游戏的初始界面，而是回到用户离开前的界面。想想这个有什么比较好的实现设计思路
- 还没有想明白，为什么模仿上上份工作中的项目，不知道是否是因为 com.unity3d.UnityPlayerActivity 和 UnityPlayer 的改变，没能连通。会花时间再想想原因

- 现在这个大框架好了之后，就可以顺理成章地去实现小 SDK 中的各个功能，登录登出，网络请求服务器等
- 现在 所有的挑战就剩对自己来说最难的游戏热更新服务器的设计与实现了
- 我可以，也已经绝大部分实现了网络上现有的将 unity 游戏界面作为安卓界面的一部分等相关简单游戏导出安卓后再构建，但涉及到自己项目中的不同是：项目使用了热更新，会比普通游戏项目增加几个难度
- 现仍想按照前工作中的一个项目，直接将安卓 SDK 接入到游戏中，并从游戏端直接构建，不再导出安卓使用 Android Studio 来构建
 - 这里需要想的一个问题是：游戏中如何从热更新域中退出来，也就是实现游戏中热更新域的多次重入与多次从热更新域中退出来

3 游戏基本场景设计

- 安卓 SDK 设计：这个项目可以略去 SDK 的设计与实现，因为加入一个 SDK，将不得不需要将主菜单界面移出热更新域，放在 unity 程序域里，会改变现有的游戏架构（绝大部分 ((99.99% 的游戏逻辑都放在热更新域里可以随时热更新))：程序的逻辑只有一个热更新域的入口，且只进不出。会不会造成什么问题，可能还有 20% 感觉还没有想透，但是有一个安卓 SDK 是很方便，也很帮忙的，很想要实现一下

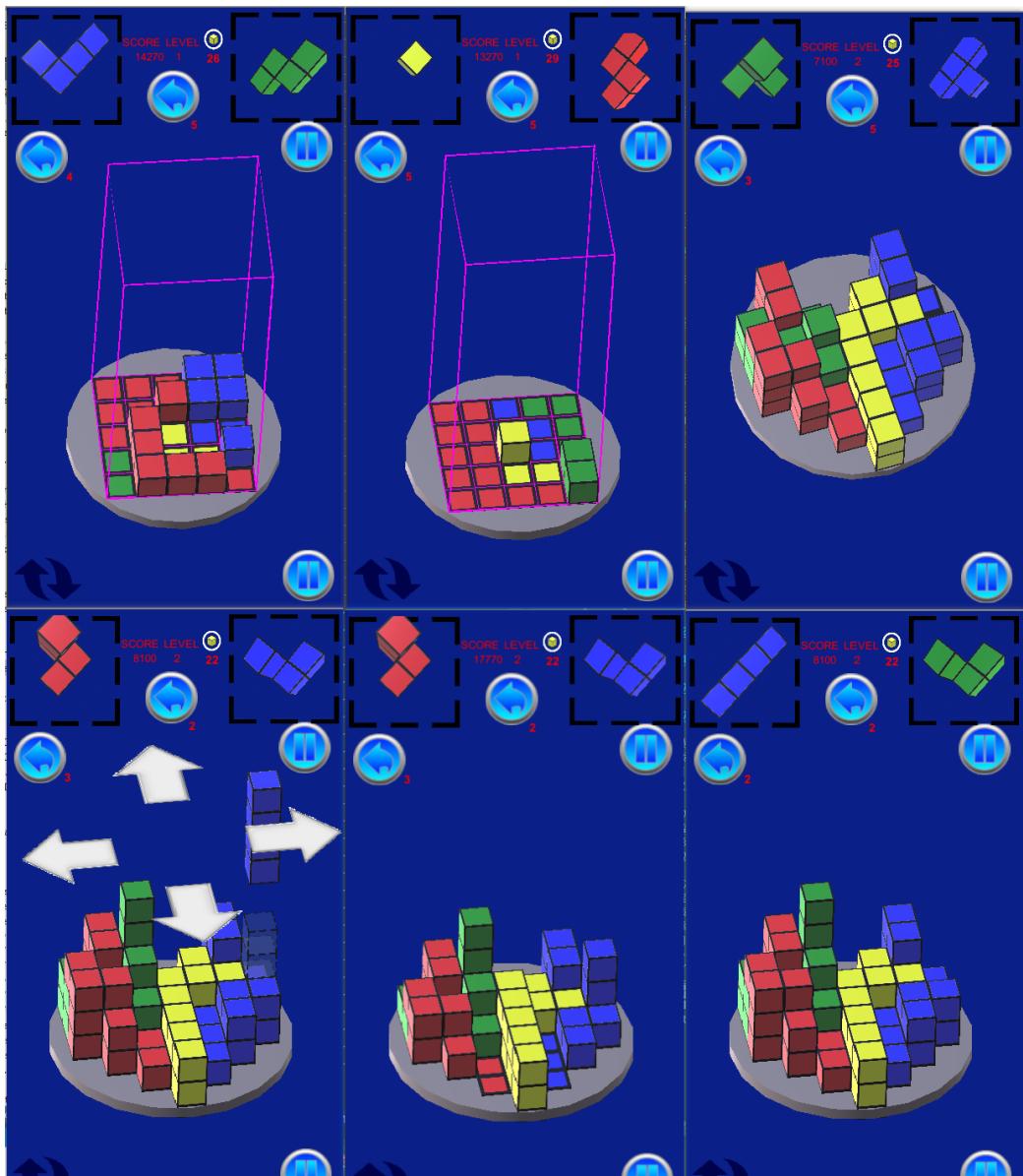


- 如上图所示，需要把热更新域里的菜单主办面提出热更新域，放主程序里，作为安卓 SDK 与热更新域交互的桥梁
- 安卓相关的，或普通游戏逻辑相关的，包装入安卓 SDK，比如安卓音量控制，电池电量提醒，登录登出，给游戏打分等，网络访问？

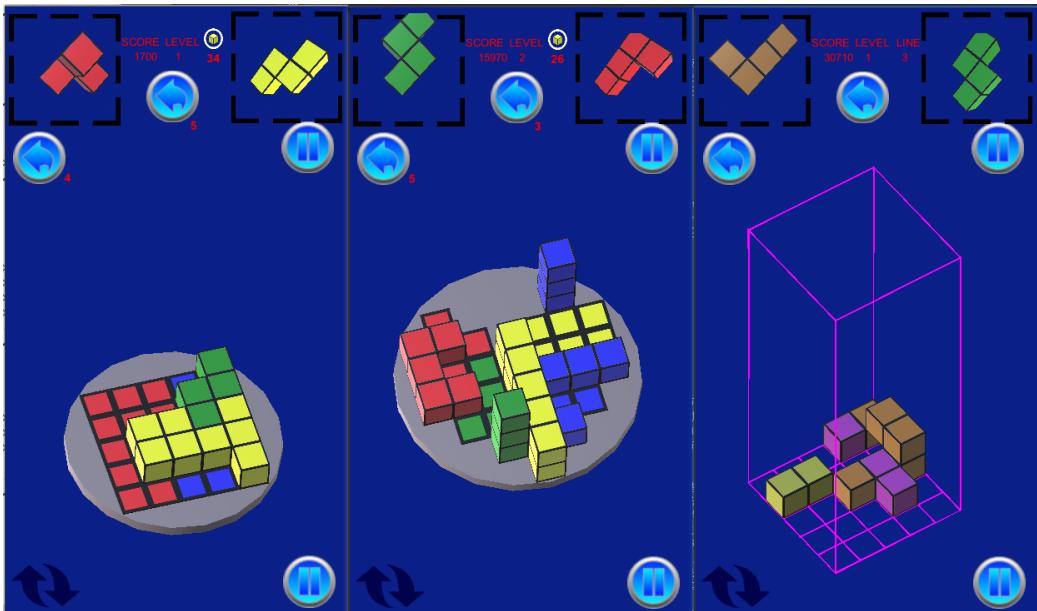
- 其实游戏逻辑仍放在热更新域里
- 从这次打包进安卓来看，其实很简单，但自己仍然走了很多的弯路才最终可以打出一个可运行的安卓包来，某些自己不够熟悉的可能需要花点儿时间
- 也就是说，接下来自己最不熟悉领域，热更新的服务器，将会成为自己最大的挑战，我会倾向于上午上网查些相关资料，理清逻辑，希望能够实现一个至少逻辑功能上相对完整的热更新服务器，但要花点时间.. 这个过程可能会比较长
- 所以，这个过程中，下午和晚上的时间希望自己还是能够好好实现一个安卓 SDK，并将现游戏按照上面的大体设计完成
- 从今天晚上把游戏菜单界面连接等相关的改 bug 开始，就开始打一个安卓 SDK 好了. 服务器有没有思路，过一个周左右再更新. 从今天晚上开始，接下来就写安卓 SDK 的封装
- SDK 封装设计：

f:/dwSDK/	157
<-> app	158
<+> libs	159
<+> src	160
<+> androidTest	161
<+> main	162
<-> java	163
<-> com	164
<-> deepwaterooo	165
<-> dw	166
[+] GameActivity.java	167
[+] MainActivity.java	168
<+> res	169
[+] AndroidManifest.xml	170
<+> test	171
[+] build.gradle	172
<-> dwsdk	173
<+> libs	174
<+> src	175
<+> androidTest	176
<+> main	177
<-> java	178
<-> com	179
<-> deepwaterooo	180
<-> dwsdk	181
<-> activities	182
<-> authentication	183
[+] DWActivateAccountActivity.java	184
[+] DWDIALOGActivity.java	185
[+] DWForgotPasswordActivity.java	186
[+] <u>DWLoginActivity.java</u>	187
[+] DWParentalCheckActivity.java	188
[+] DWSignUpActivity.java	189
[+] BaseActivity.java	190
[+] CustomWebViewActivity.java	191
[+] DWAllSetForGameActivity.java	192
[+] DWHaveAccountActivity.java	193
[+] DW SplashScreenActivity.java	194
<-> networklayer	195
[+] ApiClient.java	196
[+] Activity.java	197

- debug current problems: 现从 unity 直接安装的应用, 手机上是空屏, 感觉是游戏的第一个场景加载不成功, 原因包括:
 - 我用 android 26 构建工具运行在 android 12 api 31 上. 没有用 31 的构建工具是因为项目中引用了众多的库, 现在测试知道可行的, 必须用 java 8 和安卓 26 的构建工具
 - 另一个需要深入理解和测试的方案是 unity 导出为安卓工程, 第一个场景仍是空场景
 - 那么我热更新域项目中没有真正涉及和管理的关于场景加载的部分, 我需要把这部分再弄懂弄透. 在 editor 中运行不涉及这个部分, 我想当然地以为 unity 打包安装后也会自动运行这第一个主场景
 - 等弄出一个可以在我安卓手机上真正运行的游戏之后, 会赶快实现一个安卓 SDK, 同时搜索寻找服务器的可行解决方案 (这个服务器对自己来说是了最难的知识版块, 因为不曾涉及过)
- 等把接各种 SDK 做熟了, 可以把现有的录屏功能加进游戏里
- 安卓 SDK: 对我来说, 现在这个相对比较容易, 因为以前做过, 现在有机会再自己从头到尾做一遍, 是整体上巩固知识的过程;
- finally I am able to build unity apk using android-api level 26 and install in my Android 12 phone. but Not running well yet. Adding android-31, and will try to get it runnable on my phone tomorrow, test basic Volume android calls, work on Android SDK implementation. meanwhile, try to get hotfix server ideas clear.
- 到目前为止, 游戏热更新程序域里的逻辑 (除了挑战模式最后一关 level 11 还没怎么测试以外) 算是基本连通, 扫除了所有能够轻易察觉的 bug
- 接下来需要进行热更新域之外的工作: SettingsView 用户帐户管理, 登录等, 热更新的服务器, 安卓 SDK 打安卓包等, 以及游戏上架 google play store 所有必须的准备工作.
- 剩余的部分对自己还说才算是真正的挑战, 基本都不曾自己亲自来尾地做过一遍. 但所有的过程都有一第一遍. 会每天逐步完成一些这个模块的内容, 希望能够在一两个周之内把这所有剩余的工作做完. 最难的是现在剩余不曾做过的这个部分, 但最喜欢的也必须是这个学习的过程. 爱表哥, 爱生活!!!
- 对游戏的整体 UI 看起来的效果感觉不理想, 除了 moveCanvas 四个按钮的那个图片可以做得好一点儿, rotateCanvas 上六个按钮很不满意, 不知道是否要考虑底层 opengl c++ 再连一个 dll 域来画册 (会要好好考虑一下怎么把那 6 个按键摆放得更体会玩家心态.)



- fix all identified bugs during modes switches and ILRuntime Hotfix callback delays.

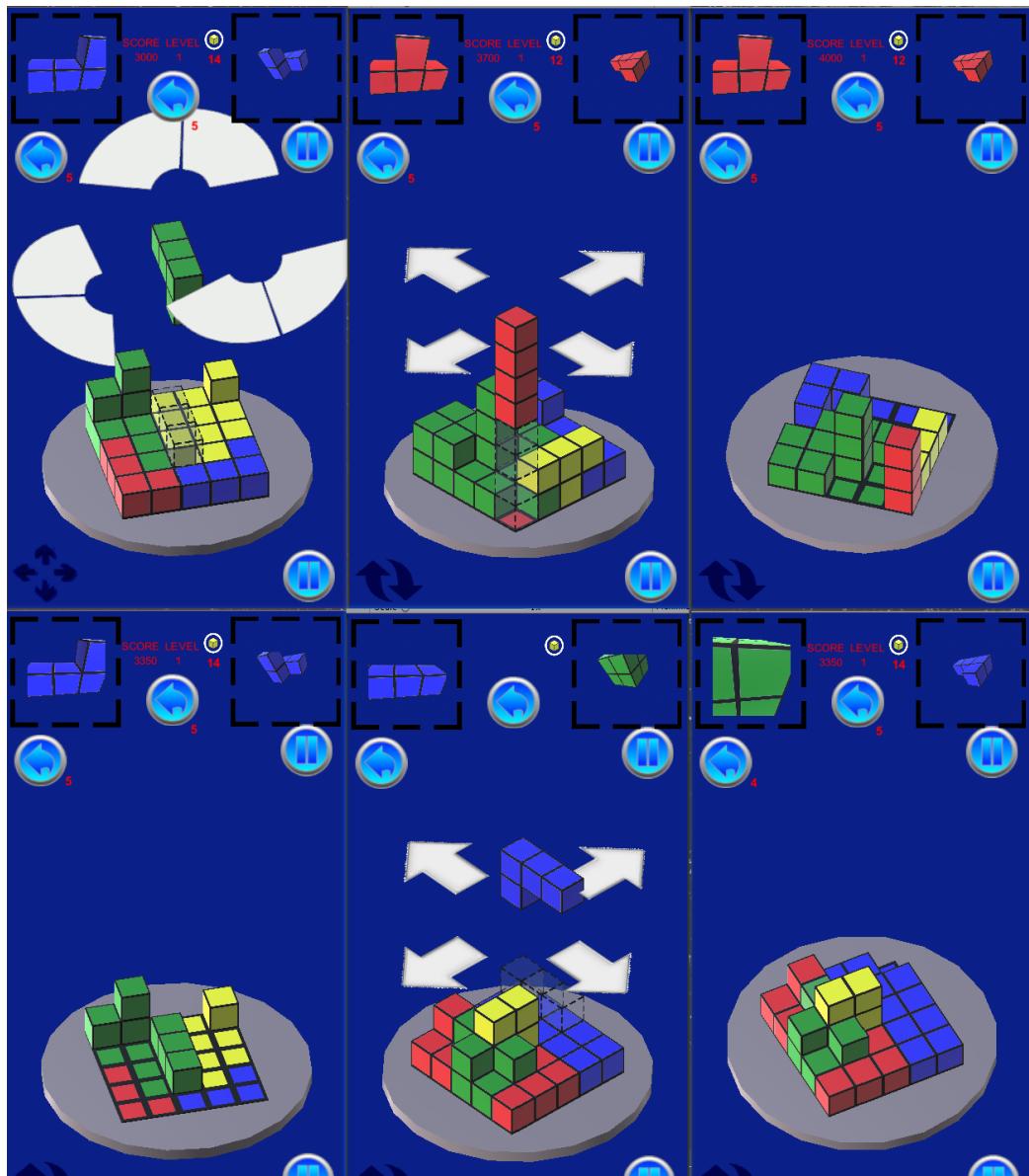


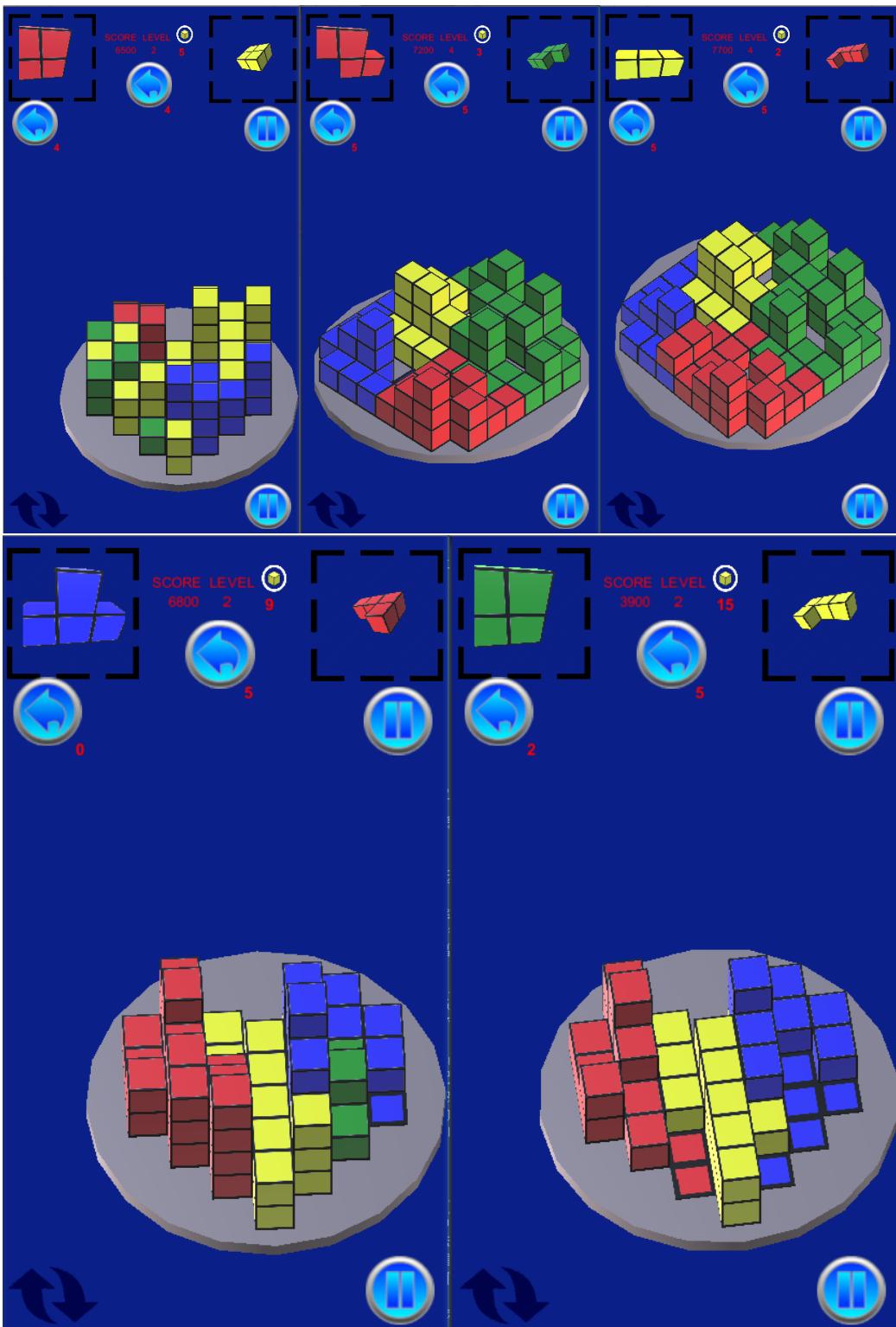
- CHALLENGING MODE: ALL kinds of bugs awaiting to be fixed. will work on them tomorrow.
- working on identified bugs. 前几天估算时间的时候没有意识到之前自己的游戏逻辑并没有写完整, 还是想尽快把挑战模式下的游戏逻辑按照原计划实现出相对理想的状态, 这几天会把游戏逻辑补充完整, 并尽量同步修复意识到的 bug
- **TODO:**
 - 要想个办法怎么把材质等相关加载到热更新程序域里去
 - CHALLENGING mode particle system NOT integrated yet; materials and colors for CHALLENGING mode to be integrated.
 - 现在前两个模式的 BUG 已经修得差不多了, 现在打很久可能才会遇见一个 bug
 - 有时候, 某个方块砖在某自旋转画面上延那个轴的两个按钮转不动, 现加必要的日志, 改天再遇到, 希望能把它们再遇到的 BUG 都改正过来
 - Integrating challenging levels modes.
 - 今天修改了几个 BUG 后能够意识到现从保存文件加载游戏进度中 (主要是由启蒙和挑战模式下的撤销最后一块即使已经消除了的方块砖所引起的) 所存在的诸多问题, 想要重新走最原始的, 就是用户想要撤销最后一块方块砖的时候, 如果不曾有消除, 只削除最后一块方块砖就可以了, 如果有, 就清理面板, 再由保存文件从头加载
 - 因为几年前自己考虑游戏的逻辑的时候可能想得并不是很完整, 所有会有很多逻辑上不合理的地方. 现在游戏过程中能够意识到的逻辑问题, 会尽力重构使游戏逻辑更合理.
 - 考虑热更新服务器该如何处理

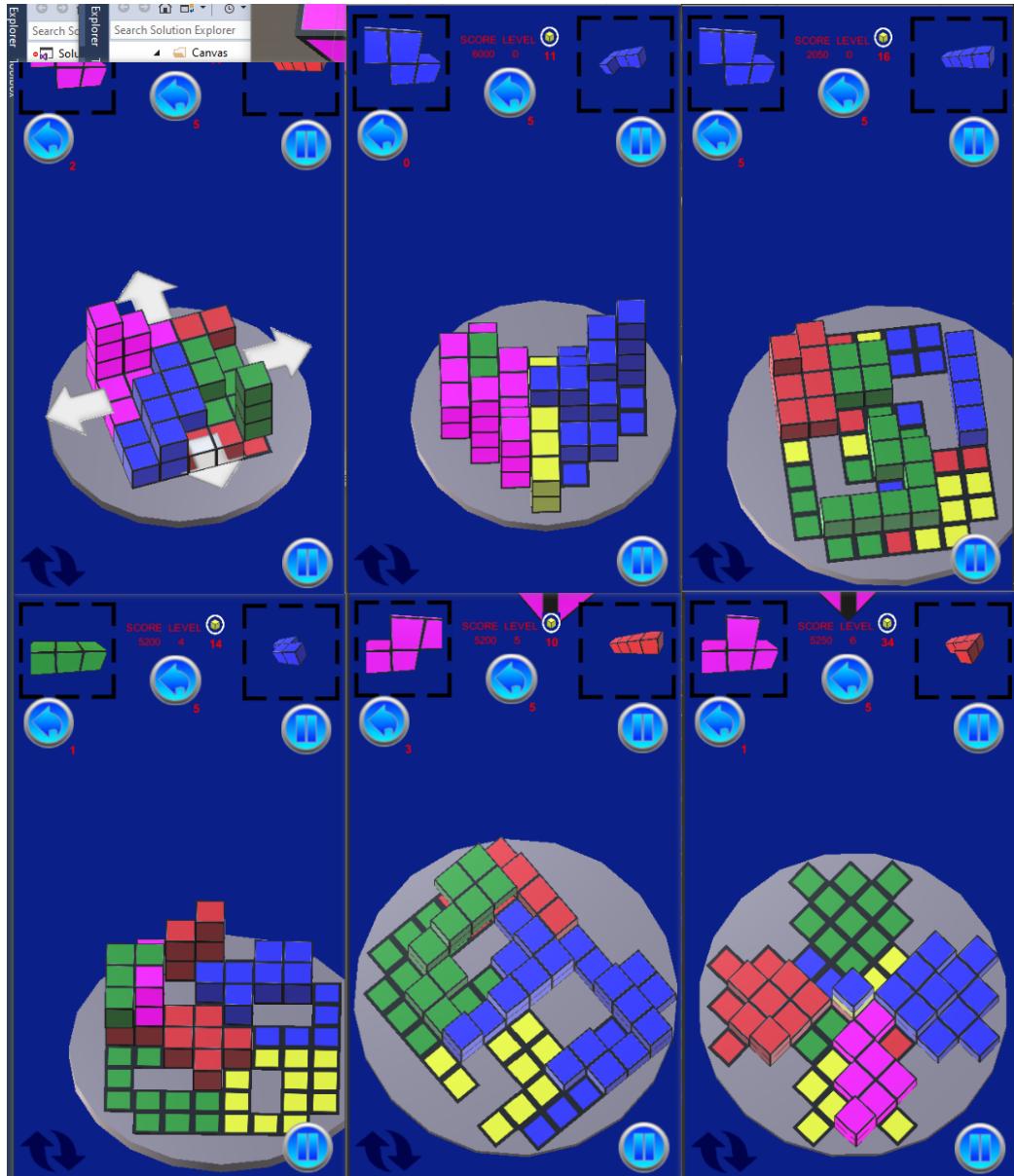


- debugging game flow, 现游戏的主场景如下 (热更新程序域里的主要逻辑功能模块化管理主要工作完成, 还没有 debug 游戏逻辑):

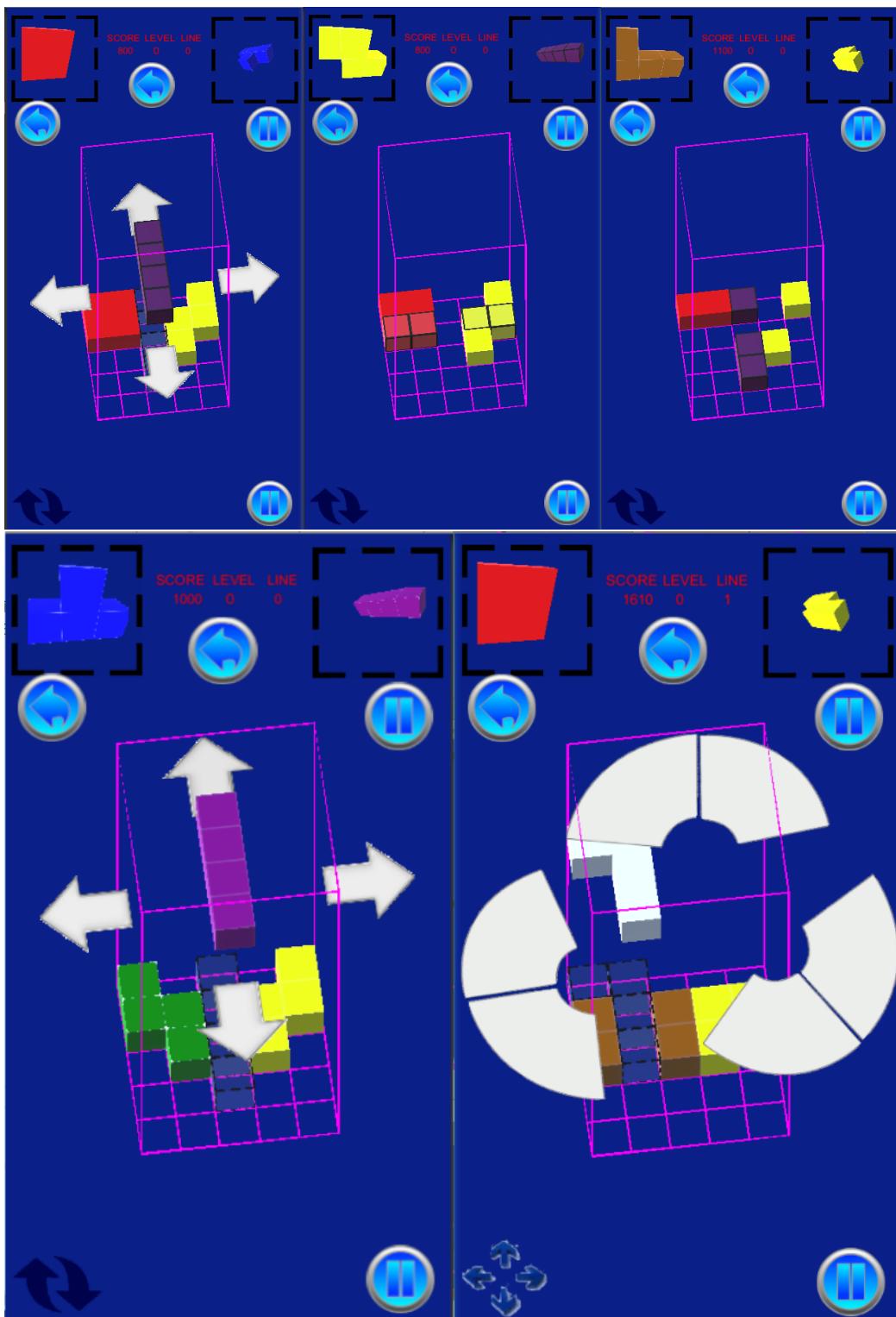
现游戏框架与实现细节, 除了将来的热更新服务器的服务端, 其它任何部分都不再有任何挑战, 只是时间问题, 需要时间来整合持战模式和修改所有遇到过的 BUG 与细节. 爱表哥, 爱生活!!!

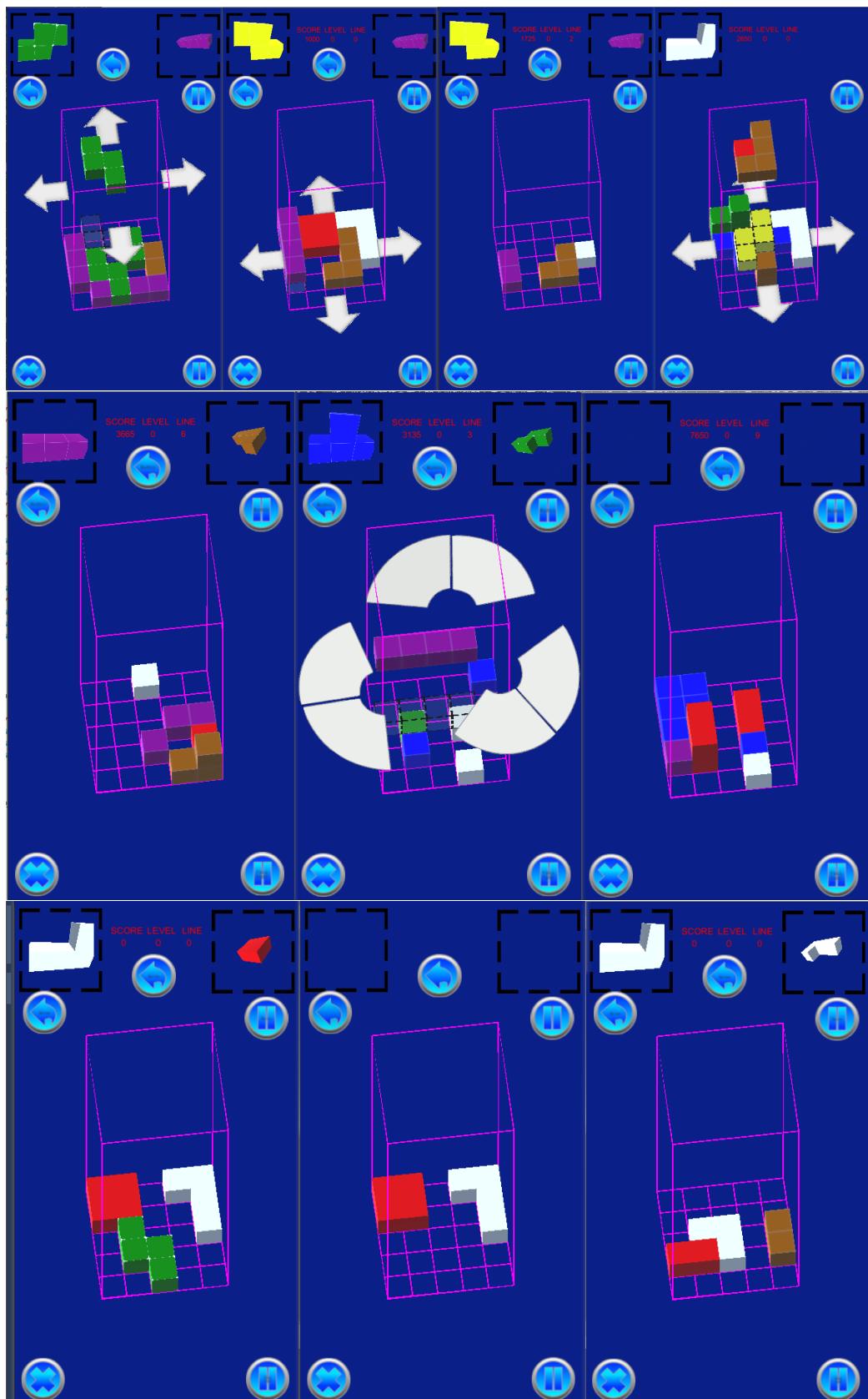


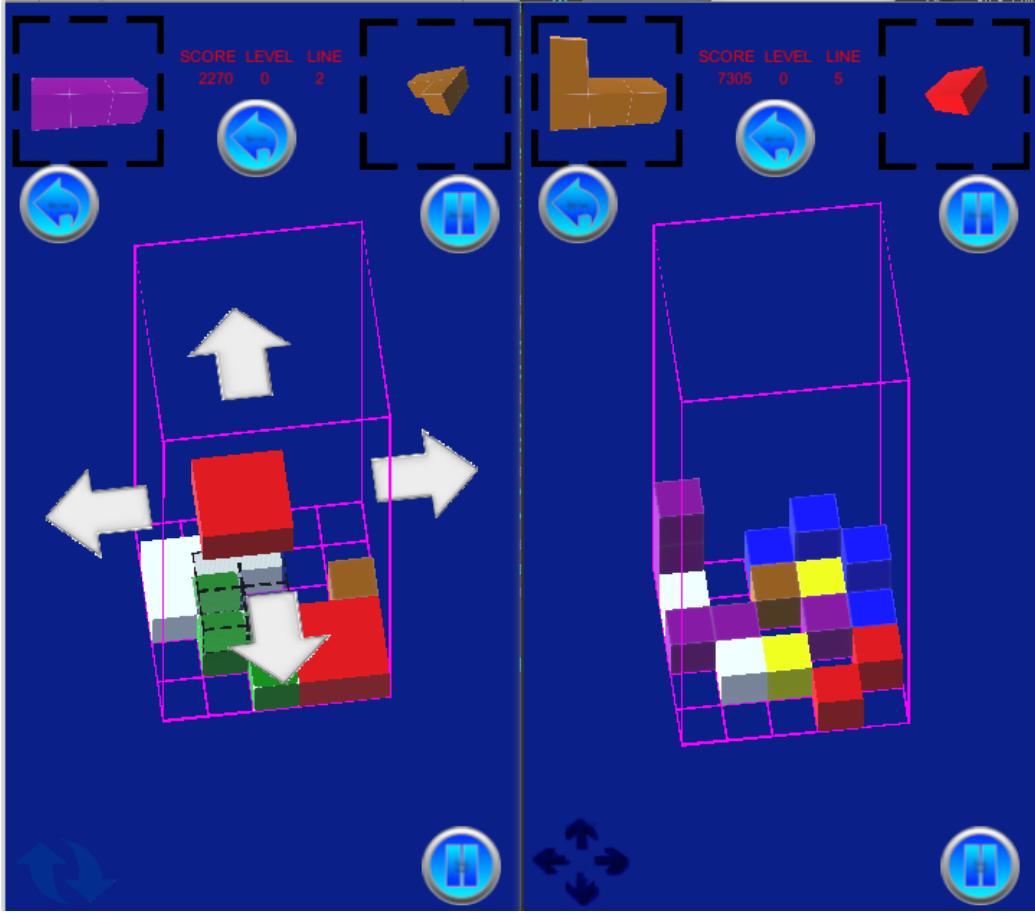




- 在整合了更为优化方便读取的日志文件之后，游戏里的 BUG 已经基本上被自己清理干净了，只有两三个不容易重复的，希望下次再遇见能将它们一一解决掉。
- 从明天开始，将精中精力整合挑战模式下的 11 关，最开始的一两关因为挑战模式下的诸多整合问题，可能会进展相对迟缓一点儿，也需要在最开始的一两关解决掉整合过程中重构过程中所造成的绝大部分相关 BUG，等这一步走完，接下来的将会容易很多。希望能够用几天的时间把这块儿做完，都是琐碎的细节，并不存在技术上的挑战/



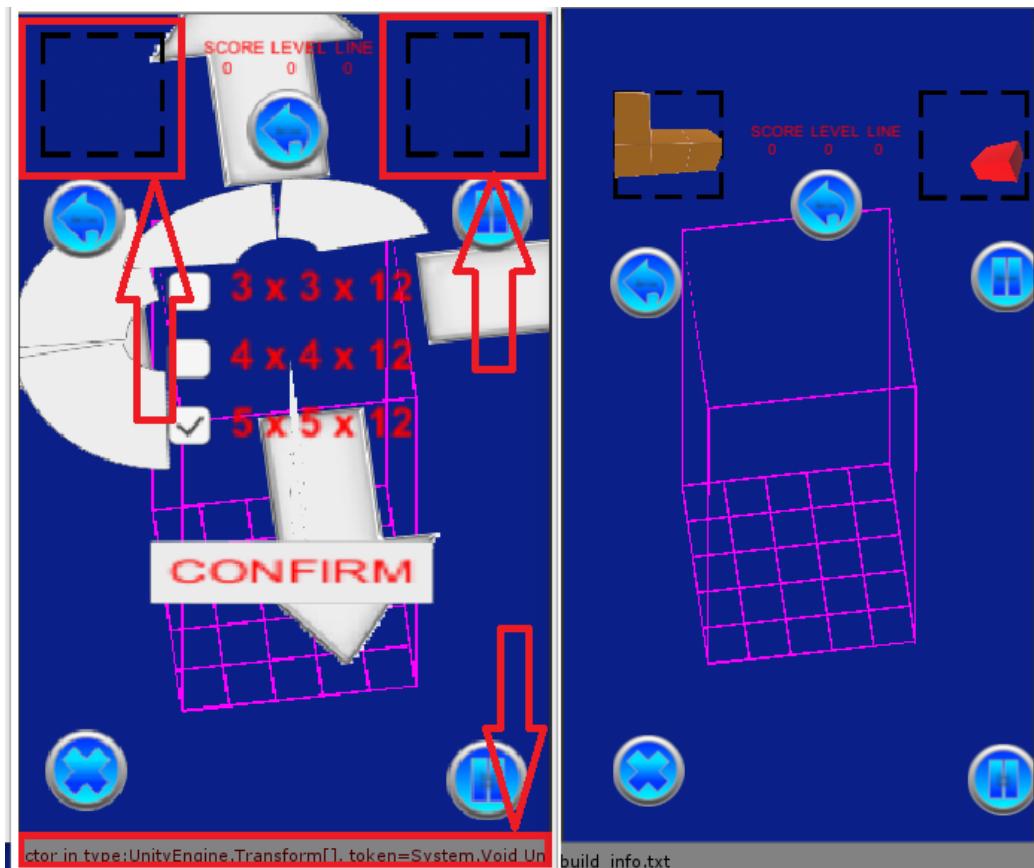




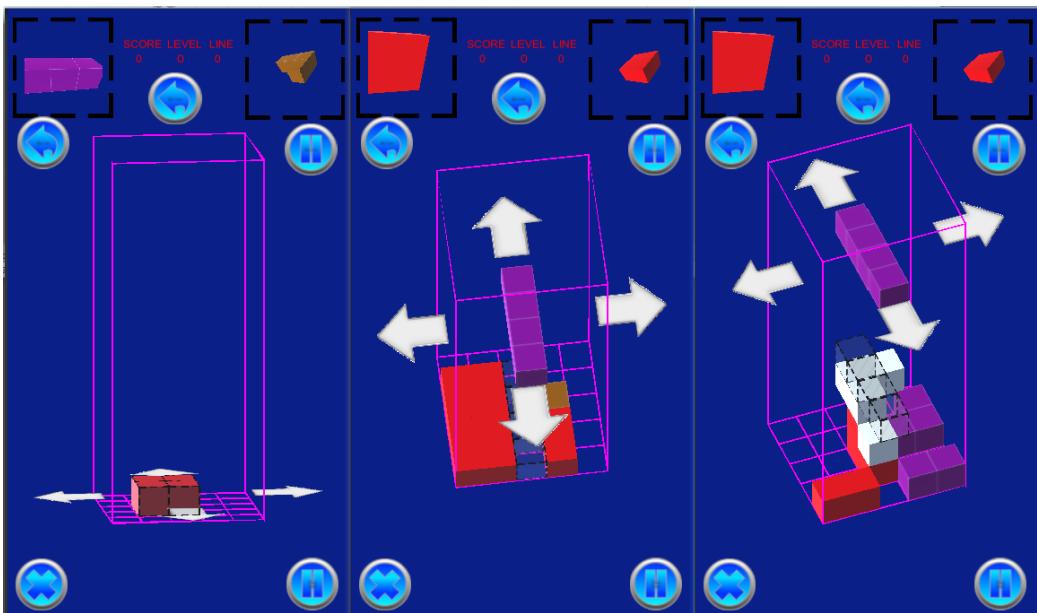
- 对于启蒙模式，是允许撤销最后一块落地了的方块砖的，即使它的落地造成了某些行或列的消除。现 debug 这个撤销最后一块方块砖过程中的 bug
- 游戏视图切换的时候，从游戏主场景出来，场景没能清理干净，某些方块砖可以缓存或是销毁，清理面板
- 现在解决了几乎所有遇到的问题或是 BUG，现在就是不管看着那堆破烂的游戏逻辑，感觉它们搅在一起太混乱，一边拆解到不同的模块里去，一边 debug 游戏逻辑（感觉现在没有很挑战的 bug，琐碎的游戏逻辑快要打不起精神来了，感觉学习和解决比较难的 BUG，等解决了问题的时候会比较开心，爱表哥，爱生活!!!）
- BUG：阴影方块砖最开始的位置还是对的，可是后来的位置不对，会跑到方块砖的上面
- 现逻辑支持但凡前一步有个方块砖落地，就可以撤销。但仍然存在一个方块砖落地后同时消除多行时的撤销时可能会还有 bug to be fixed
- 上面的这个 bug 前几天在整合 CLASSIC CHALLENGING MODE 之前，已经修改好了，但是因为重新整合了一套系统，所以昨天今天又把这套新系统里的这些类似的 BUG 又改了一遍。并且这套新系统里会有更多的 BUG，会需要考虑更多的因素，但是对于自己写出来的源码，改这些都只是时间问题
- CHALLENGING MODE 下小立方体是带黑边的，现部分预设安装了黑边，部分没有，因为一定会整合 CHALLENGING MODE，所以最终都会变成带黑边的，但目前混合存在的就让它们暂时如此

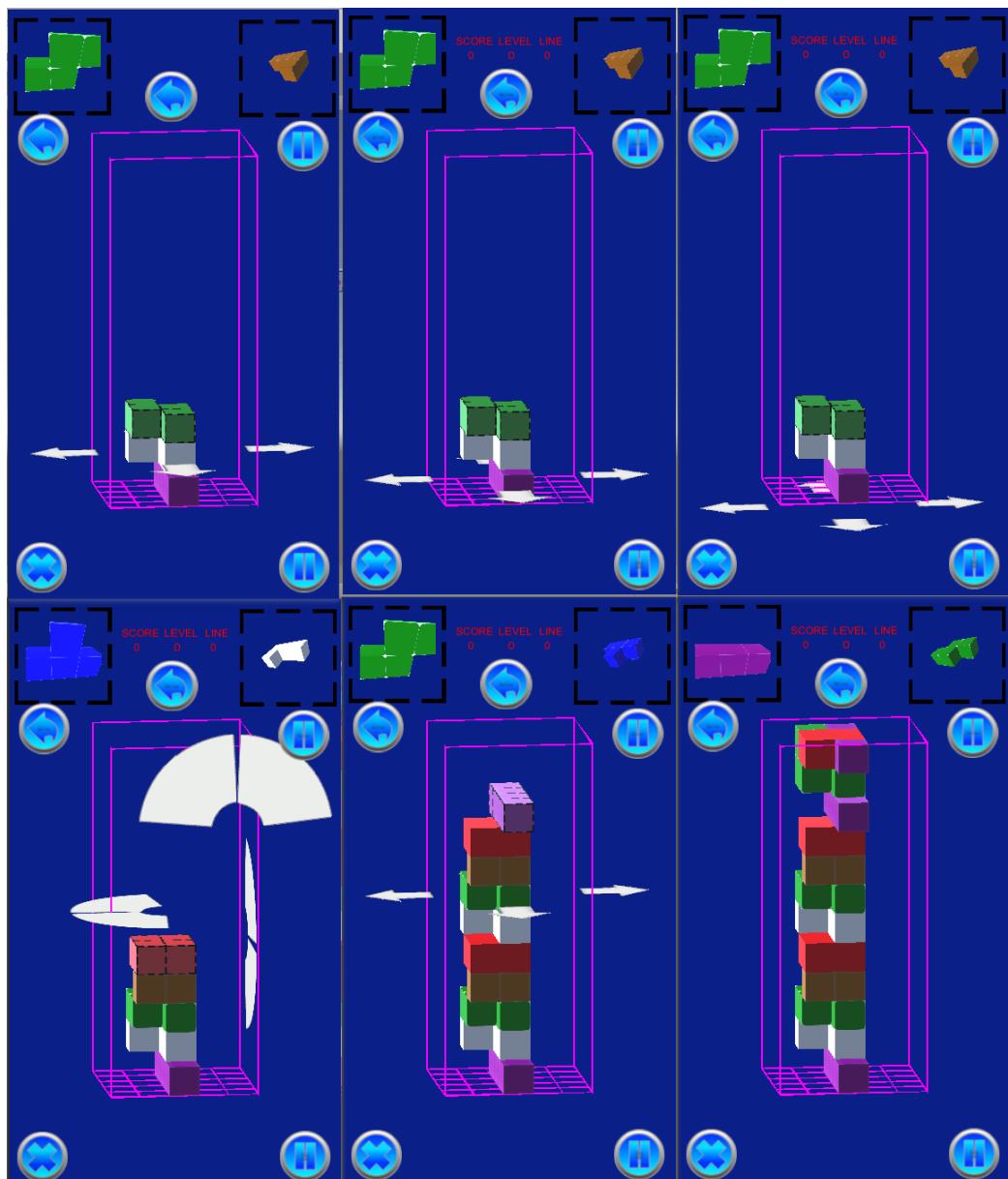
- 现系统里整合了 C# unity COROUTINE, 虽然还没能如预期执行, 但是游戏的整体难度相对提升.
- - 重要的逻辑基本都已经解决, 剩下的就是把游戏的逻辑连通 (今天中午吃多了, 下午傍晚脑袋都不是很好用)
- 接下来三天给自己放假远足到 WSU 的校园去看足球赛, 不更新, 周六会接着更新
- 继续现在改好的逻辑, 把游戏逻辑连起来. 基本现在能够想到的比较难一点儿的全都连通了, 狠开心, 爱表哥, 爱生活!!!
- 昨天平移和旋转画布上的两组按钮的点击事件没有问题; 可是今天 UI 面板上有些按钮点不通, 这个点击事件的传递系统, 也还需要花点时间. 热更新程序域里的绝大部分按钮都是点得通可以回调的, 平移组四个按钮, 旋转组应该也没有问题, 主游戏界面暂停按钮也可以好好工作, 只是有些按钮还连不能, 感觉是其它问题. (同类按钮, 同一面板上的按钮, 能够有一个可以运行, 那么这个大的框架逻辑是通的, 就不用担心了, 剩余只将是细节上的小修改)
- 现在最主要的逻辑有以上两个问题. 但都能够解决.(两个基本都解决了!!! 强大的 debugging strategy!!! 爱表哥, 爱生活!!!)
 - <Tetromino> <GhostTetromino> 继承自 MonoBehaviour 的脚本在运行时添加适配过程中出意外: instance 总是空, 也可以说是我的 AddComponent<T> 方法没有适配? 这个类型的适配有点儿没有做好 (这个应该是目前最重要的问题, 但不是不能解决的问题) 把官方 DEMO 中的例子好好运行好研究透彻再来试图解决自己目前遇到的问题 (两个项目可以参考)
 - 我不知道现存的方案里其它人的项目是如何实现的. 回到问题的本质, 那就变成为最简单的办法, 便是自己实现一个计时系统, 或是模拟一个每隔 (比如说 1 秒钟, 方块砖就下降一格就可以了, that's it!) 这次重构想要达到的目标便是基本绝大部分的逻辑都可以热更新重构. 那么只要我能够模拟每隔一秒更新一次就解决问题了, 这个项目对于我来说 80% 逻辑理顺, 剩下的就是热更新的服务器了
 - 自己实现计时器的方法大致思路: 那就分 section, 每个小节玩 5 分钟, 挑战模式可以加到 10 分钟; 每个方块砖每隔两秒下降一格; 需要考虑应用的离线时间, 就是游戏中去玩别人的应用了, 再回来时间连续计算一个小节 5 分钟
 - 今天下午理过思路包括:
 - * 刚才没有把问题想明白: 因为经过了适配, 本身的 UnityEngine.AddComponent<T>() UnityEngine.GetComponent<T>() 在热更新工程中的正常运行是没有问题的
 - 出问题的特殊之处是在: Tetromini.cs GhostTetromino.cs 是在热更新工程中定义的, 当游戏运行, unity 工程无法得知热更新工程中 Tetromino.cs GhostTetromino.cs 为何物
 - 上面说得不对, 因为加 component 本身是在热更新工程中, 它是知道自己工程中所定义的部件的
 - * 所以得想办法把这两个类移到 Unity 工程中来 (这个反而可能会比较繁琐, 也可能逻辑不通)
 - * 按照官方建议, 我们是可以重置这两个方法的, 让它有办法认得热更新工程中所定义的脚本 (顺着这条途径把问题理顺, 那么就发现别人的控件逻辑是在 Unity 主工程的, 也就是有主工程中的 MonoBehaviour 系来驱动各生命周期事件, 但是我的热更新控制逻辑是在热更新工程中, 并没有一个默认的游戏引擎来驱动事件的自行发生)
 - * 所以, 没有设置好的原因, 另一个是在热更新工程中, 我没有哪个地方来调用 UNITY 工程的系统的自动运行:
 - 前面的各种适配是适配给 unity, 让它认识热更新工程中的诸多类型函数等

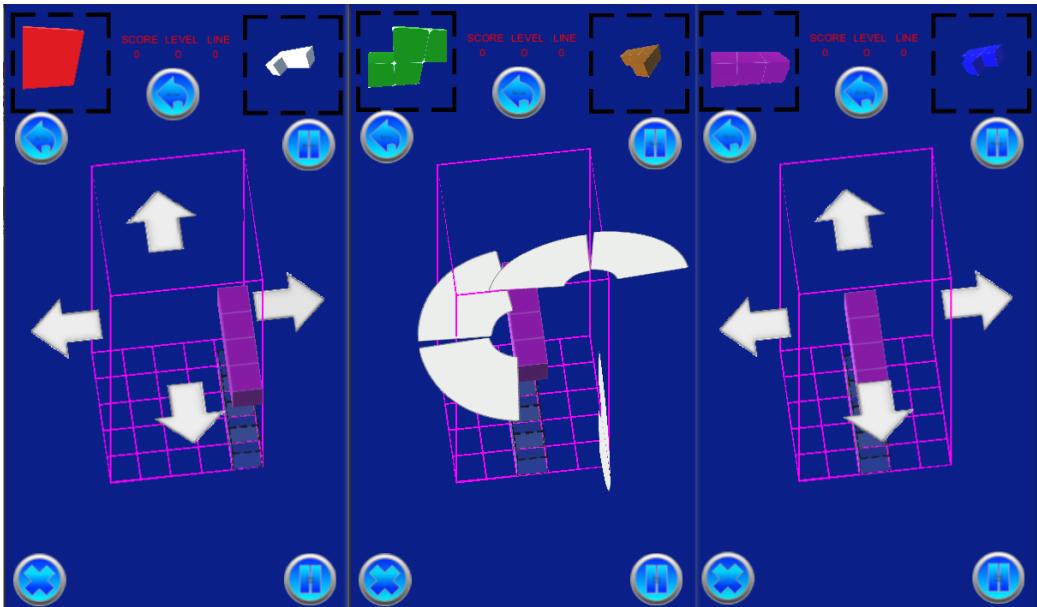
- 可是按照自己游戏逻辑, 感觉更像是热更新工程中需要适配 unity MonoBehaviour 的生命周期事件?
- 那么再回到上面, 刚想过的
- * 所以得想办法把这两个类移到 Unity 工程中来 (这个反而可能会比较繁琐, 也可能逻辑不通)
 - 那么这么试一下, 倒还是有可能的, unity MonoBehaviour 系能够自动驱动生命周期事件, 引导必要时候游戏的进行??? 测试一下
- 示例工程中这些劫持是, 代码适配用于提供给 Unity 工程来加载或是获取 (AddComponent<>(), GetComponent<>()) 热更新工程中 unity 所不认识的定义的类等, 与自己游戏逻辑不同, 不用
 - AudioManager,EventManager 可能需要适配, 就需要自己把原理都弄明白了
 - 先前的 PoolManager 的解决是采用 ViewManager 里静态管理的方法, 可以如期运行, 有待优化
 - 那么上面两个如果一时半会儿找不到更好的办法, 就可以参照上面的方法解决



- 已经解决了的先前的
 - 加载保存系统也已经完全没有问题了 (将 BinaryFormatter 保存加载系统放在主工程中的局限是这个保存加载模块的源码将来将无法热更新, 但暂时不考虑这个需求)







- 几个主要管理器的模块化逻辑基本搭建完成, 那么就可以把 audio, events, canvas tetromino, GhostTetromino 等的逻辑相对于游戏主逻辑分开, 使接下来的逻辑 debugging 不至于有太多混乱代码
- AudioManager 这个模块的实现暂时还没有遇到什么不适配的问题 (BUG: 游戏音乐暂停后, 当游戏恢复, 背景音乐还没能恢复),
- PoolManager 有不适配的问题, 暂放一下 (这个模块继续放在 ViewManager 里).
- EventManager: 构建完整, 可以工作, 热更新程序域里确实有了比较模块化的架构, 可以让游戏逻辑得以前行
 - 接下来先把游戏里另一个主要的传导系统 Evenet delegate 的逻辑在热更新域里理通理顺, 方便热更新程序域里有个比较好的架构
 - **TO BE FIXED:** 试了两种不同的体系: 将所以点击事件与代理放热更新域与, 把点击事件的触发与回调类型放主工程, 热更新中只作回调, 都可以做到无运行时错误, 但点击回调体系还没有连通. 我觉得理论知识上这块儿还有点儿欠缺, 需要一两个早上把这块的理论再理解得透彻一点. 会试着使至少这两个体系中的某一个运行, 作为热更新里主要按钮点击回调体系的构建
 - 我觉我的整个事件传递系统可以完全放在热更新里面来做. 放在两个不同的域 (把事件的定义与管理器放在主工程的坏处是: 它好像建了两个不同的管理器, 这会造成很多不便, 希望只有一个管理器来管理所有的事件, 所以可以很快放弃这个不成熟的想法)

```

LoveButtonListener: Start()
UnityEngine.Debug.LogHandler:Internal_Log(LogType, String, Object)
UnityEngine.Debug.LogHandler:LogFormat(LogType, Object, String, Object[])
UnityEngine.Logger:Log(LogType, Object)
UnityEngine.Debug:Log(Object)
(deepwateroo.tetris3d.Events.MoveButtonListener:OnEnable() (at Assets\Scripts\deepwateroo\tetris3d\Events\MoveButtonListener.cs:17)

(Filename: Assets/Scripts/deepwateroo/tetris3d/Events/MoveButtonListener.cs Line: 17)

EventManager Awake()
UnityEngine.Debug.LogHandler:Internal_Log(LogType, String, Object)
UnityEngine.Debug.LogHandler:LogFormat(LogType, Object, String, Object[])
UnityEngine.Logger:Log(LogType, Object)
UnityEngine.Debug:Log(Object)
(deepwateroo.tetris3d.Events.EventManager:Awake() (at Assets\Scripts\deepwateroo\tetris3d\Events\EventManager.cs:22)
UnityEngine.GameObject:AddComponentWith(Type)
UnityEngine.GameObject:AddComponent(Type) (at C:/buildslave/unity/build/artifacts/generated/common/runtime/GameObjectBindings.gen.cs:382)
UnityEngine.GameObject:AddComponent() (at C:/buildslave/unity/build/artifacts/generated/common/runtime/GameObjectBindings.gen.cs:387)
Framework.Utils.SingletonMono`1:get_Instance() (at Assets/Scripts/Framework/UtilSingleton.cs:26)
(deepwateroo.tetris3d.Events.MoveButtonListener:OnEnable() (at Assets\Scripts\deepwateroo\tetris3d\Events\MoveButtonListener.cs:19)

(Filename: Assets/Scripts/deepwateroo/tetris3d/Events\EventManager.cs Line: 22)

EventManager: RegisterListener()
UnityEngine.Debug.LogHandler:Internal_Log(LogType, String, Object)
UnityEngine.Debug.LogHandler:LogFormat(LogType, Object, String, Object[])
UnityEngine.Logger:Log(LogType, Object)
UnityEngine.Debug:Log(Object)
(deepwateroo.tetris3d.Events.EventManager:RegisterListener(EventListener`1) (at Assets\Scripts\deepwateroo\tetris3d\Events\EventManager.cs:46)
(deepwateroo.tetris3d.Events.MoveButtonListener:OnEnable() (at Assets\Scripts\deepwateroo\tetris3d\Events\MoveButtonListener.cs:19)

(Filename: Assets/Scripts/deepwateroo/tetris3d/Events\EventManager.cs Line: 46)

EventManager delegatesMap.Count after: 1
UnityEngine.Debug.LogHandler:Internal_Log(LogType, String, Object)
UnityEngine.Debug.LogHandler:LogFormat(LogType, Object, String, Object[])
UnityEngine.Logger:Log(LogType, Object)
UnityEngine.Debug:Log(Object)
(deepwateroo.tetris3d.Events.EventManager:RegisterListener(EventListener`1) (at Assets\Scripts\deepwateroo\tetris3d\Events\EventManager.cs:62)
(deepwateroo.tetris3d.Events.MoveButtonListener:OnEnable() (at Assets\Scripts\deepwateroo\tetris3d\Events\MoveButtonListener.cs:19)

(Filename: Assets/Scripts/deepwateroo/tetris3d/Events\EventManager.cs Line: 62)

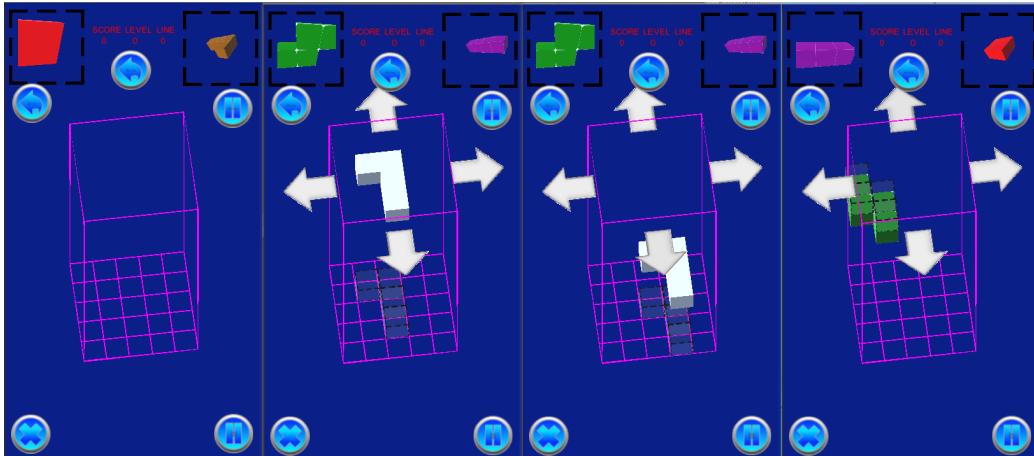
EventManager delegateLookupMap.Count after: 1
UnityEngine.Debug.LogHandler:Internal_Log(LogType, String, Object)
UnityEngine.Debug.LogHandler:LogFormat(LogType, Object, String, Object[])
UnityEngine.Logger:Log(LogType, Object)
UnityEngine.Debug:Log(Object)
(deepwateroo.tetris3d.Events.EventManager:RegisterListener(EventListener`1) (at Assets\Scripts\deepwateroo\tetris3d\Events\EventManager.cs:63)
(deepwateroo.tetris3d.Events.MoveButtonListener:OnEnable() (at Assets\Scripts\deepwateroo\tetris3d\Events\MoveButtonListener.cs:19)

(Filename: Assets/Scripts/deepwateroo/tetris3d/Events\EventManager.cs Line: 63)

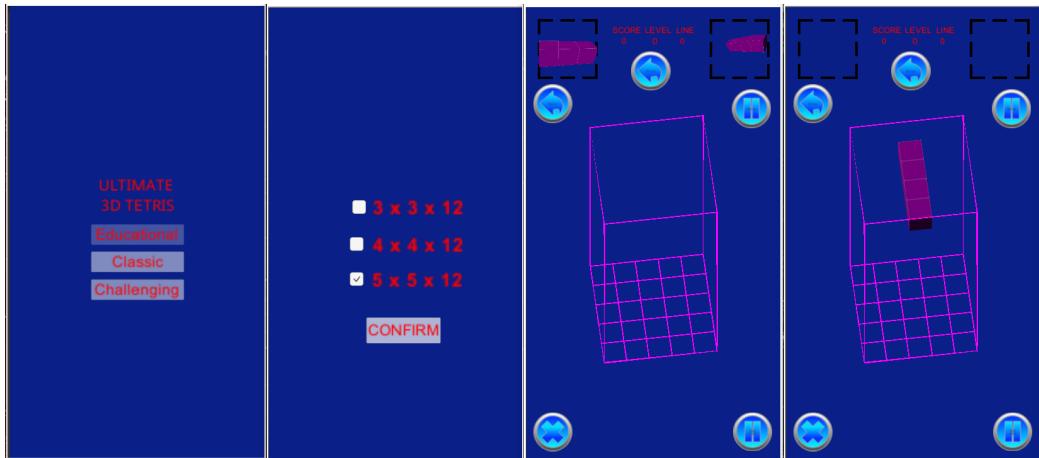
Mono: successfully reloaded assembly

```

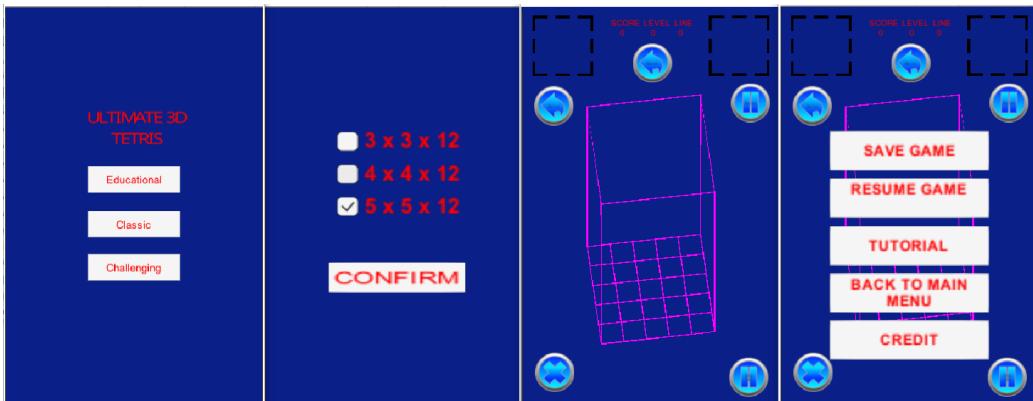
- 那么受限于热更新程序域里的静态调用，不能用最优的方法，但是把相应的按钮放在相应的 Canvas 里来处理



- 预设都做好了，现在要将预设打资源包，并从资源包读出来供视图实例化等
- finding the easiest way to refactor yet still be able to hotfix after app installed already.
- 现在游戏显示都没有问题了，开始 debug 游戏逻辑以及功能模块等（现在只是运行了可模拟测试版的，需要在热更新程序域里将这些逻辑重构到运行出这种效果来，明天写，明天下午写？还是什么时候来写这点儿呢？）
 - trying to link all necessary game logics and make game to run again in ILRuntime HotFix 程序域里。



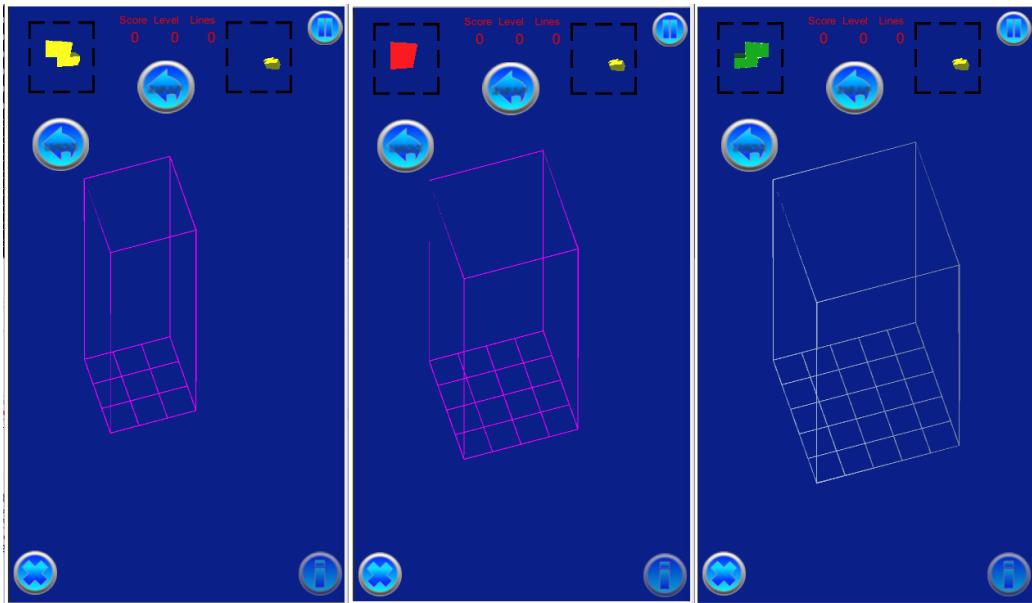
- moveCanvas rotateCanvas 上点击事件, 事件系统的传递. 如果上面的问题一时半会儿解决不了, 可以先试图解决这个并测试一下, 给上面最难的 BUG 一点儿网络搜索和解决问题的时间(很好解决) 这里只是用了最基础的方法来实现, 以前自己都曾实现过事件系统, 现在只是测试和解决主要关键点, 知道都可行可实现, 会再进一步的使用适当的设计模式来优化源码
- 两个预览方块砖的生成并画到视图上去: 现在解决这个问题
 - 原理很简: 将两个预览放在不会出现在主相机的两个固定的位置上; 再用两个不同的相机分别照在两个预览上, 并分别投射到一块渲染媒介, 显示在屏幕的固定投影位置上就可以了
 - 大致原理如此, 但运行时存在: 场景里各不同视图会被某些不确定的因素旋转某些角度, 以及放大缩小位数的问题.
 - 运行时可能涉及这块投影渲染媒介的实例化 (不知道目前不能很好地渲染是否是因为我打包时没有打包它? 还是说因为他们出现在两个不同视图的原因呢?)
 - 就是因为如上的目前我还不太理解的不确定性, 给这个游戏的 unity 视图显示造成一定的困难, 但也不是都解决不了的, 需要花时间来慢慢解决这些小问题
- at least temporatorily passed initial running
 - 现两个主要的小问题: 多维数组在 ILRuntime 热更新程序域里的适配,
 - 多维数组, 稍微改动了一下就可以了, 但里面还是有点儿小机关的
 - * AOT 不能使用二维数组 (多维数组) 例如 `bool[,]`
 - * 使用时报 `System.Boolean[,]:Get` 没有生成 AOT 代码
 - * 改用 `bool[][]` 是 OK 的
 - * ILRuntime Version
 - * 1.6.7
 - * 答案是: 需要正确生成 clr 绑定
- 热更新里重新实现在的游戏主场景如下:



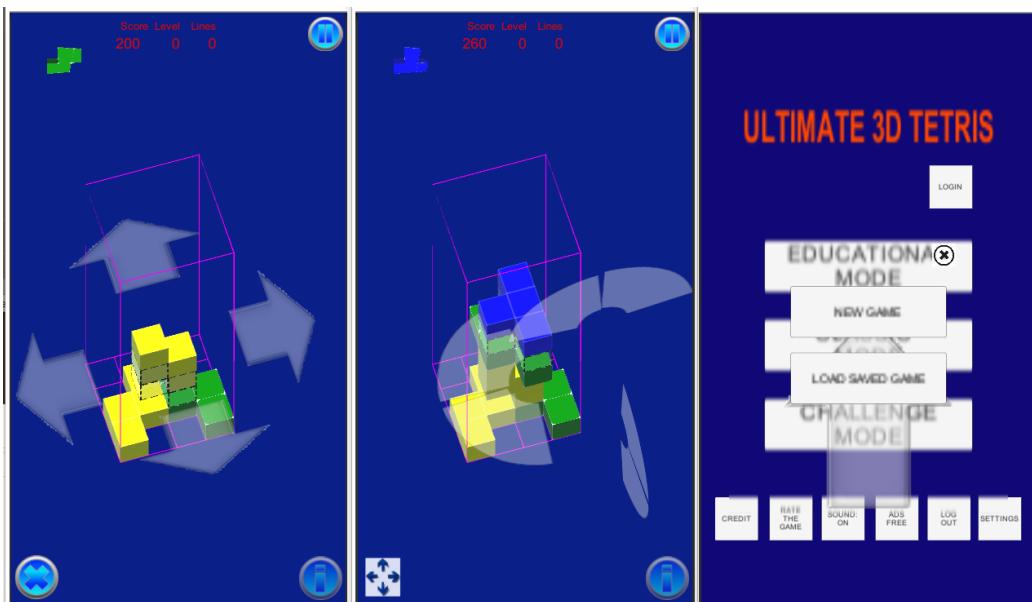
- 主游戏菜单与游戏中选择菜单: 最右为 Educational has 3 choices:



- 启蒙模式原本是想给小盆友玩儿的，有无限撤销方块功能，和粒子消除行与列。但是这具模式有可能最终被我砍掉，相关功能改加到其它模块
- 启蒙模式下的由易到难三种选择：Educational mode 的三种不同界面



- 传统游戏界面视图:(挑战模式下的界面丢了, 到时候再补吧, 或者可能只做 7 级, 剩余热更新)
- 两组共 10 个对各小方块砖方块砖平移与旋转的操纵: 平移与旋转按钮都太丑, 的摆放与位置需要优化
- load new game or saved games: 保存游戏数据的地址需要再改变一下, 改变到应用的内部, 而不是要存到什么其它的盘



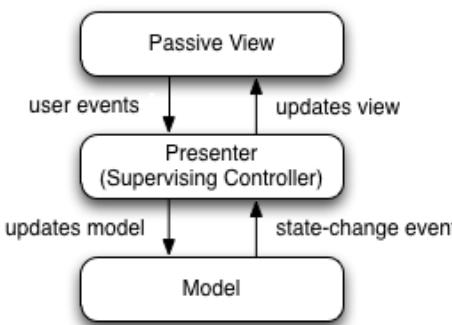
- 现在是热更新的框架到上个周末就搭好了, 这一两天忙点儿, 必要的游戏场景视图基本搭配到位: 场景的搭建没有任何复杂的地方, 只是相机的使用相对不够熟练, 所有的都只是场景搭建基本功

m

4 大致进展计划

- 不管是什么方法, 适配原源码也好, 基本也解决了现热更新程序域里的所有编译错误, 现在就是解决运行游戏过程中可能会遇到的所有问题, 让游戏在热更新框架下能够顺利运行起来
 - 处理立方体与方块砖资源包的打包与读取到视图中作必要的准备, 供运行时实时实例化, ViewManager.cs 整合资源池
 - 必要的预设都做好了, 要再理解一下从文本读取脚本资源, 运行中与预设是如何结合起来生成实例的, 把这部分的逻辑连通
- 重构把代码搬过来的编译错误也比较多, 就严格按照游戏的逻辑来, 一步一步地添加使之运行, 解决重构过程中可能会遇到的所有问题. 比如现在, 就先让教育模式下的两个供选择方块砖在游戏主视图加载的时候能够显示出来
- 暂时不处理摄像机与场景相关, 摄像机视角的热更新等游戏的主要逻辑完成后作为高级附加功能再添加整合模块; 因为方块砖游戏中只涉及到一个场景, 所以暂时不处理场景的热更新打包与加载等, 使用框架但细节略过, 因为场景中基本没有多的逻辑需要处理.
- 框架搭好测试运行好了, 必要的游戏场景资源建好了; 接下来 会侧重游戏逻辑 MVVM 设计模式, 视图与视图数据的分离与监听通知等
- 要上手就来一个怎样很好的设计, 对于目前来说还是相对庞大的游戏来说, 可以也并不是一样容易的事.
- 游戏几年前的实现逻辑大部分还能够回想得起来, 比较可行的办法是按照游戏的执行逻辑, 在热更新程序包里先一步一步链接好, 能够使游戏先运行起来, 在功能模块的不断的添加过程中, 一再优化这里面的数据或是热更新程序包里的游戏逻辑架构设计
- 现手上的资源项目没有使用 View 与 ViewModel 的数据双向传递 (或者是说 ViewModel 部分的逻辑根本就没有或是没有实现), 会再检查一遍. 这里就需要仔细地去想, 怎么模块化管理自己游戏中的数据 (MVVM, 为什么网络上他们会用 MVC 或是 MCP 呢)
- View 和 ViewModel, 在创建视图的时候就自然绑定视图模型了. 那么相应的视图模型就以观察某些数据 (是视图观察视图模型中的数据变化--自下向上传递; 视图中的按钮点击又下发更改相关数据等的逻辑, 自上向下传递)
- 搭桥: 怎么把单个视图层数据转变成为全局可访问数据, 接触到过的方法有写入 Settings.Global.ContentProvider, 用 SharedPreference 写入配置文件等. 这里考虑在热更新程序域里的特殊性
- 旋转按钮的画布做得非常差 (功能上相对完整, 只是看起来很差), 需要很有效地优化
- 更高层级的要求是使用 UniRx, 但是现在还是先实现出一套可运行的逻辑才再使用 UniRx 的响应式编程吧.....
- 资源池的部分:
- 把框架里面的 root view 的概念理解清楚: 建立起这个概念对于应用中主要游戏场景的隐藏与显示会比较方便调控
- 立方体与方块砖打在什么资源包里比较好, 怎么打包, 把他们单独打成一个包. 把它们单独打一个大包, 就相应的逻辑来读取这个立方体方块砖资源包 =====
- Mino Tetromino 阴影等的预设都很好做 (会把平移与旋转视图今天上午做好, 帮助推进游戏逻辑); 难的是高强偶合的游戏逻辑的模块化元件化解偶合, 游戏逻辑的拆解与链接

- Unity 中使用 Json 进行序列化与反序列化: 理解, 以及在方块砖项目中的使用, 包括了资源打包相关的序列化与反序列化, 以及游戏进展进度数据的保存与加载序列化反序列化. 这里涉及到一点点儿 OOD 设计, 从 TRANSFORM 到 mino 序列化, 到方块砖序列化, 到游戏进展进度数据的序列化等层层嵌套.....
 - 热更新重构前自己的游戏里的存储系统是使用的 binaryformatter, 但是现在可能把这个存储系统重构成为使用 Json 序列化与反序列化
 - * 前几年的理解力有限, 以前力所能及地想要提高效能的办法是, 比如消掉一行的时候, 某个元件 L 只消掉了右边的短横, 那么我只回收右边的短横; 并且我的资源池里也缓存到了每个小立方体的级别
 - * 现在重构一时半会儿还没有弄懂游戏场景的打资源包与从资源包加载初始化 (因为我的游戏可以只有一个场景, 其它全都只是视图的切换), 没有弄透游戏里的这个元件的序列化与反序化, 与自己先前的实现相比, 优恶各在什么地方? 如何在热更新里更为优雅地实现序列化反序列化同时还保证性能, 这些问题我一边试图透过更多的视角来理解现在项目体系中的某些设计与实现, 也会想要再网络搜索一下, 希望尽快能够思路清晰起来
- 为什么一部分的数据放在数据包 (主要负责序列化 [与反序列化]), 一部分逻辑相关的放在控制包 (Model, MVC vs MVP)? 序列化与反序列化的放数据包, 逻辑调控相关的放在控制包里?
- 需要同步弄懂的是: 方块砖资源池在热更新里的使用, 案例学习与自己游戏逻辑的实现
- 游戏暂时不考虑相机的动态调整与保存, 只当它只有一种固定不变的设置
- 把 Unity 程序域里定义的框架 ILRuntime MVVM 等主要模块都还理解得比较透彻了; 会去深入理解热更新程序域里的数据驱动与传递, 作要的 research, 把热更新程序域里的数据传递模块理解和设计好
- 前段时间一直想当然天真地以为这个框架是 ILRuntime + MVVM 设计模式, 实际上因为框架中使用了 UniRx, 这个框架应该更多的是 MVP? 需要再好好读一下理解一下框架中的双向数据传递以及数据驱动等, 把这些都弄懂理顺



- MVP 设计模式 Model-View-(Reactive)Presenter Pattern
- 用 UniRx 可以实现 MVP(MVRP) 设计模式。
- 为什么应该用 MVP 模式而不是 MVVM 模式? Unity 没有提供 UI 绑定机制, 创建一个绑定层过于复杂并且会对性能造成影响。尽管如此, 视图还是需要更新。Presenters 层知道 view 的组件并且能更新它们。虽然没有真的绑定, 但 Observables 可以通知订阅者, 功能上也差不多。这种模式叫做 Reactive Presenter:

```

// Presenter for scene(canvas) root.
public class ReactivePresenter : MonoBehaviour {

    // Presenter is aware of its View (binded in the inspector)
    public Button MyButton;
    public Toggle MyToggle;

    // State-Change-Events from Model by ReactiveProperty
    Enemy enemy = new Enemy(1000);

    void Start() {
        // Rx supplies user events from Views and Models in a reactive manner
        MyButton.OnClickAsObservable().Subscribe(_ => enemy.CurrentHp.Value -= 99);
        MyToggle.OnValueChangedAsObservable().SubscribeToInteractable(MyButton);

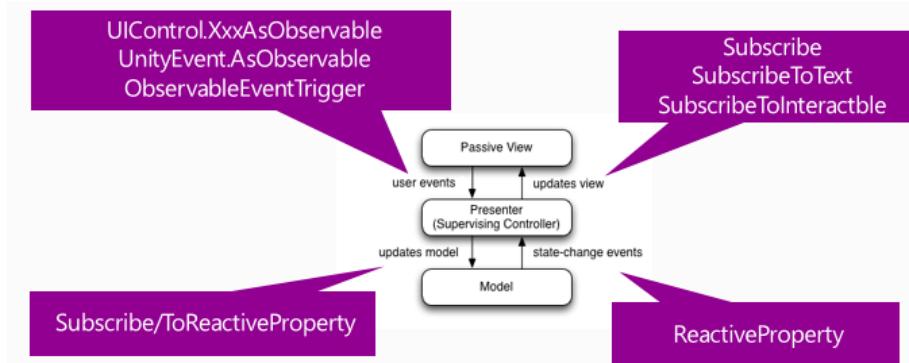
        // Models notify Presenters via Rx, and Presenters update their views
        enemy.CurrentHp.SubscribeToText(MyText);
        enemy.IsDead.Where(isDead => isDead == true)
            .Subscribe(_ => {
                MyToggle.interactable = MyButton.interactable = false;
            });
    }
}

// The Model. All property notify when their values change
public class Enemy {
    public ReactiveProperty<long> CurrentHp { get; private set; }
    public ReactiveProperty<bool> IsDead { get; private set; }

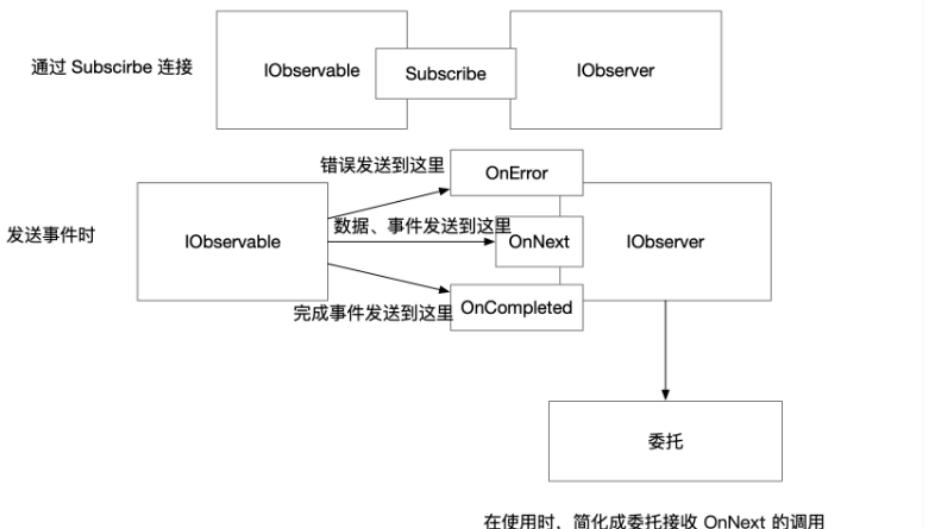
    public Enemy(int initialHp) {
        // Declarative Property
        CurrentHp = new ReactiveProperty<long>(initialHp);
        IsDead = CurrentHp.Select(x => x <= 0).ToReactiveProperty();
    }
}

```

- 视图层是一个场景 scene，是 Unity 的 hierarchy 定义的。展示层在 Unity 初始化时将视图层绑定。XxxAsObservable 方法可以很容易的创建事件信号 signals，没有任何开销。SubscribeToText 和 SubscribeToInteractable 都是简洁的类似绑定的辅助函数。虽然这些工具很简单，但是非常有用。在 Unity 中使用很平滑，性能很好，而且让你的代码更简洁。



- V -> RP -> M -> RP -> V 完全用响应式的方式连接。UniRx 提供了所有的适配方法和类，不过其他的 MVVM(or MV*) 框架也可以使用。UniRx/ReactiveProperty 只是一个简单的工具包。
- 下面有个 Rx 讲给小白说的话：



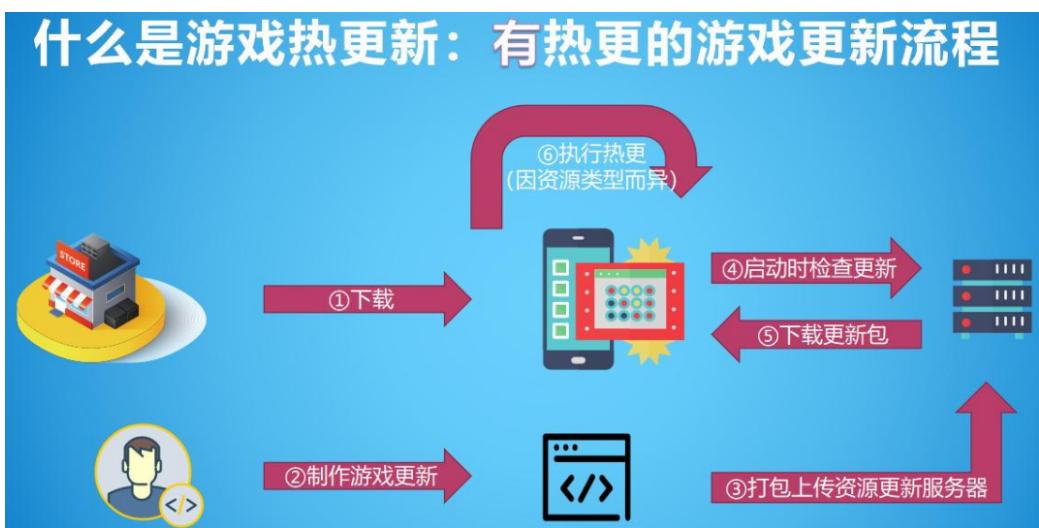
- 今天晚上和明天就力所能及地看可以 由现有的基本框架到明天傍晚能够实现多少基本流程
- 现在, 进行热更新重构后, 感觉 第一要务是尽快地把现有功能都整理实现做出来, 做出来是第一要务; 丑就丑, 美术和优化绝大部分实现完后才再考虑
- 过程中纪录自己感觉需要重构或实现的点滴, 需要补的知识点等; 在无聊近乎麻木的重构过程中也希望能尽快地捡起需要补的知识点; 希望最终整个游戏的实现流程由框架搭建测试通过, 到流程由简到难都是顺畅的
- 游戏场景里相机还需要一点儿处理 (需要加一个跟踪方块砖的脚本)
- 所有可能我还是需要把场景的热更新再理透一点儿, 分场景加载应该是更有利于内存的 (就是还没有使用的资源的有效的释放, 但也还是看情况)
- 以后有想法会再补这里

5 进展过程与基本问题

- 框架基本算是已经搭建起来了 (除了 还没有热更新的服务器以及 还不是很理解如何打资源包, 程序代码包相对简单很多);
- 游戏服务器打算暂时不着手处理, 因为主要是 想要深入理解 ILRuntime+MVVM 这个热更新框架
- 框架基本上算是搭起来了, 但是并不是说它就能够如愿运行得很好, 现在的主要问题是热更新的程序集里还有 60 个左右的主要是两个不同的程序域里类型转换相关的错误需要自己一一改正。
 - 同昨天晚上的那个错误一样, 会回去检查 Framework ILRuntime 里的所有的错误
 - 这里也需要自己对 ILRuntime 的深入理解
- 现在可以用相对较古老的版本凑合着运行起第一个视图, 项目可以用相对古老的版本继续往下建下去
- 但是我仍然希望能够自己试着去解决现存的热更新程序集里的约 60 个错误. 这个可能会花一些时间来一一消除它们, 但是值得尝试.

6 把原理弄懂

- 热更新的服务器是自己目前的难点，但可以放置再决定最终是想要如何解决（用还是不用）；
- 使用 unity 2017 .NET framework v3.5 的热更新流程（除了场景的加载还没有去试图理解，没有太花时间在上面，因为目前的项目还不会用到）到今天下午可以完全自己实现完整了，没有任何的问题
- Unity 程序域的各种代码 + 热更新模块程序域逻辑的实现 + UI 视图的各种资源打包 + Unity 里热更新代码领域的资源包打包：三四个模块的基本原理弄懂弄透，基本可以达到手撕的程度了.....
- 框架搭建基本算是圆满完成结束；从今天晚上开始，读自己原来的游戏程序代码，梳理一下接下来自己游戏玩法逻辑模块设计等，列个小计划，也需要理解触碰到现有逻辑里需要重新设计或是弥补的版块对于自己目前不够了解或是还相对陌生的地方需要补起来
- 热更新模块的实现：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（主要是理解与分析问题的能力，以及能够钻研进入解决问题的深度上比以前强太多了），需要把热更新模块补充出来；
- ILRuntime + MVVM 框架设计：两者结合，前几年的时候没能把 MVVM 理解透彻；ILRuntime 也没有看很懂，现在基本能够看懂，大致本地的热更新流程也能建得通运行得通
- 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；**服务器方面的知识点相对欠缺**
- 服务器是自己现在相对的难点，但是仍然是可以暂时复制粘贴来完成热更新资源的更新的，所以还是要能够快速开发出热更新模块的游戏视图与逻辑
- 以前被自己弄不的 JAVA 模式，因为现在要写 CSHARP，需要把 JAVA - 模式给修理好，让 csharp-mode 代码有相对干净清洁的 snippets 运行环境
- 下面有个很好玩的图：它描述了应用从店里下载安装后，热更新资源上载到服务器以及客户端检查更新，下载实现更新的大致过程。

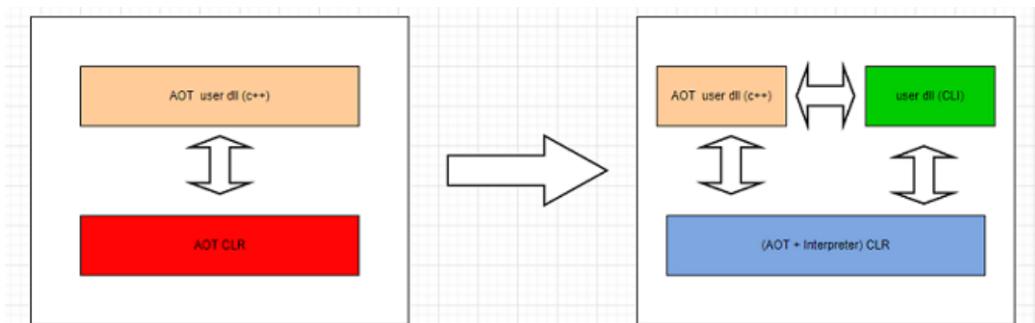


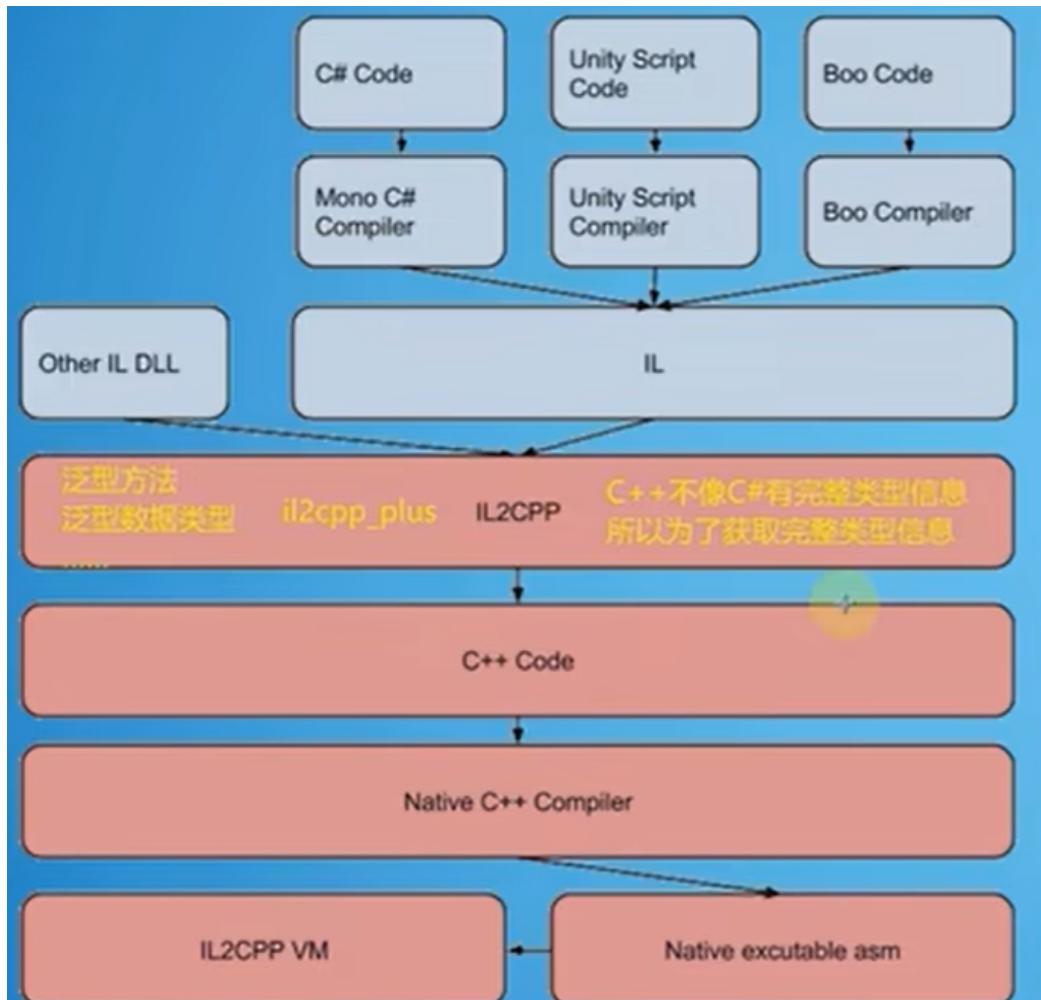
要是两个小项目：

- 资源包的准备：热更新分程序热更新和资源的热更新；那么现在的项目就是资源的热更新是分成了两个小项目来实现资源热更新资源包的自动打包(分场景打包和其它资源打包)；程序热更新因为主要是更新视图，游戏的所有基本逻辑主程序都运行在热更新程序包下，所以三个小项目便可以实现所有资源(是指包括资源和程序)的自动打包为可上载热更新服务器的程序包。(三个小项目看起来是最简单的，但是全部实现出来可能还是工作量最大的)
- 服务器层的相对理解：应该是需要一个好用的第三方程序，或是合适好有物服务器来提供必要的资源包上载到服务器；服务器层可能还需要根据不同的应用平台(IOS 安卓等)来进行一定的配置，以及必要的压力测试保证相对大量用户的情况下可以正常上载下载运行(后一步暂不考虑)
- 客户端：对于不同的客户端应用平台，游戏运行时的资源包 MD5 比对的原理要再熟悉一下
- 我觉得我该考虑尽快至少建个本地服务器了
- 性能优化：另外是对其实高级开发的越来越熟悉，希望应用的性能表现，尤其是渲染性能与速度等、这些更为高级和深入的特性成为这次二次开发的重点。
- 现在是把自己几年前的写的游戏全忘记了，需要回去把自己的源码找出来，再读一读熟悉一下自己的源码，了解当时设计的估缺点，由此改进更将

7 几种不同热更新模式的探讨

7.1 HybridCLR——划时代的 Unity 原生 C# 热更新技术：IL2CPP 与热更新





很不幸，不像 Mono 有 Hybrid mode execUtion，可支持动态加载 DLL。IL2CPP 是一个纯静态的 AOT 运行时，不支持运行时加载 DLL，因此不支持热更新。目前 unity 平台的主流热更新方案 xLUa、ILRuntime 之类都是引入一个第三方 VM (Virtual Machine)，在 VM 中解释执行代码，来实现热更新。这里我们只分析使用 C# 为开发语言的热更新方案。这些热更新方案的 VM 与 IL2CPP 是独立的，意味着它们的元数据系统是不相通的，在热更新里新增一个类型是无法被 IL2CPP 所识别的（例如，通过 System.Activator.CreateInstance 是不可能创建出这个热更新类型的实例），这种看起来像，但实际上又不是的伪 CLR 虚拟机，在与 IL2CPP 这种复杂的 CLR 运行时交互时，会产生极大量的兼容性问题，另外还有严重的性能问题。一个大胆的想法是，是否有可能对 IL2CPP 运行时进行扩充，添加 Interpreter 模块，进而实现 Mono hybrid mode execUtion 这样机制？这样一来就能彻底支持热更新，并且兼容性极佳。对开发者来说，除了解释模式运行的部分执行得比较慢，其他方面跟标准的运行时没有区别。对 IL2CPP 加以了解并且深思熟虑后的答案是——确实是可行的！具体分析参见第二节《关于 HybridCLR 可行性的思维实验》。这个想法诞生了 HybridCLR，unity 平台第一个支持 iOS 的跨平台原生 C# 热更新方案！

- 现在也简单地理解一下这个方案最简单原始案例实现的基本原理，若有兴趣，就可以再深入地探讨一下

8 环境弄得比较好的包括：

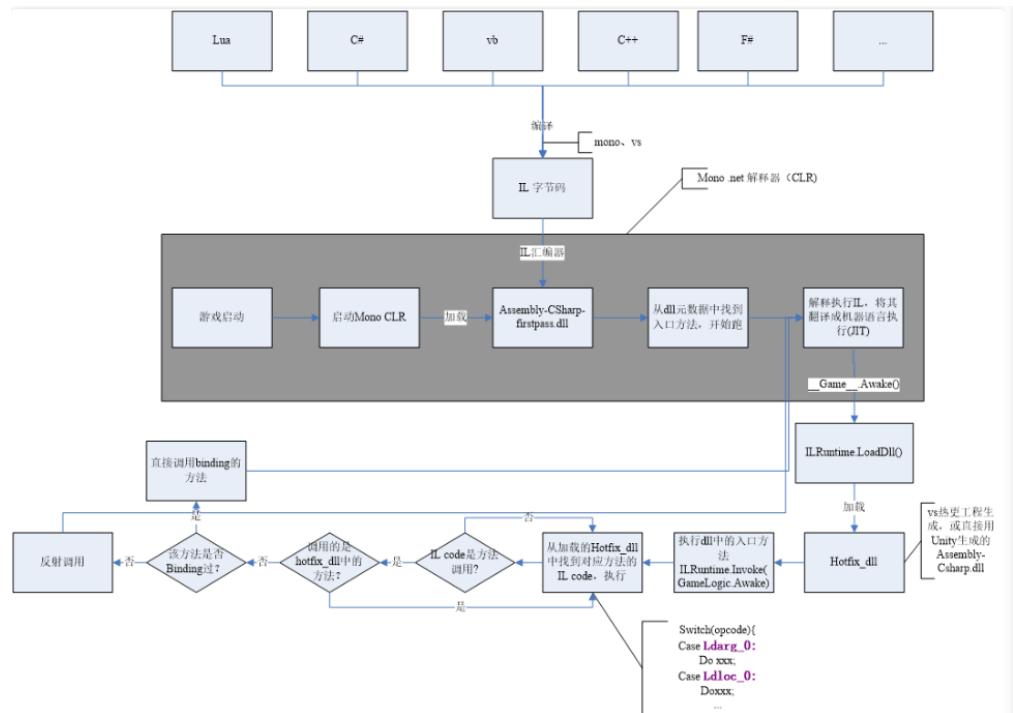
- 电脑的配置有限，文件稍微大一点儿的时候已经不太好处理了；所以不得不分割成多个小文件

- 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!!!

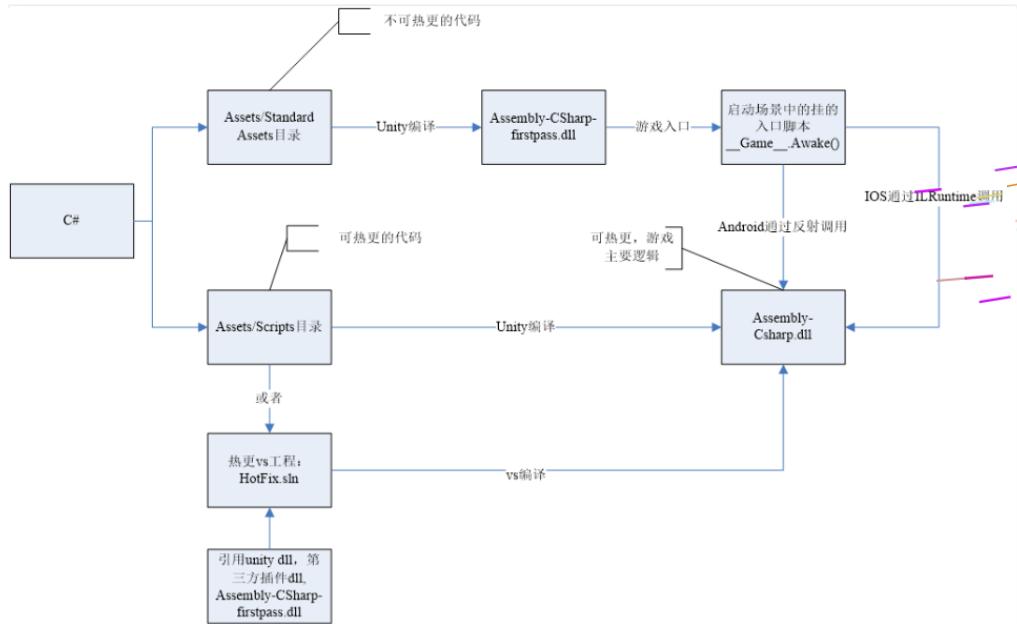
9 ILRuntime 库的系统再深入理解

9.1 ILRuntime 基本原理

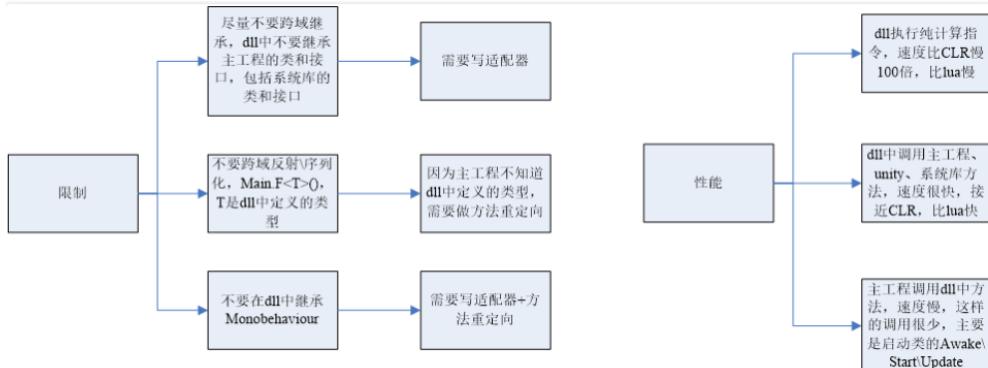
- ILRuntime 借助 Mono.Cecil 库来读取 DLL 的 PE 信息，以及当中类型的所有信息，最终得到方法的 IL 汇编码，然后通过内置的 IL 解译执行虚拟机来执行 DLL 中的代码。IL 解释器代码在 ILInterpreter.cs，通过 Opcode 来逐语句执行机器码，解释器的代码有四千多行。



9.2 ILRuntime 热更流程



9.3 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 RegisterDelegateConvertor**：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

```

app.DelegateManager.RegisterDelegateConvertor<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 解译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 ILInterpreter 的解释器里面实例化
- 不带返回值的重定向：

```

public static StackObject* CreateInstance(ILInterpreter 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 ILInterpreter.PushObject(esp, mStack, ((ILType)t).Instantiate());
        } else // 通过系统反射接口创建实例
            return ILInterpreter.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 redirect
        appdomain.RegisterCLRMethodRedirection(i, CreateInstance);
    }
}

```

- 带返回值方法的重定向

```

public unsafe static StackObject* DLog(ILInterpreter __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 = ILInterpreter.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 = ILInterpreter.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

9.4 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

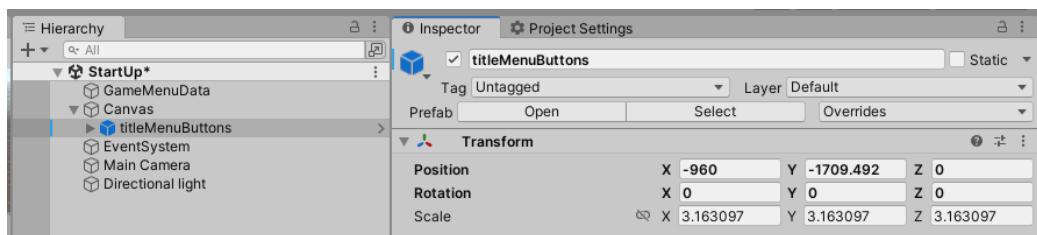
```
appdomain.DebugService.StartDebugService(56000)
```

- 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

9.5 线上项目和资料

- 掌趣很多项目都是使用 ILRuntime 开发，并上线运营，比如：真红之刃，境·界灵压对决，全民奇迹 2，龙族世界，热血足球
- 初音未来：梦幻歌姬使用补丁方式：<https://github.com/wuxiongbin/XIL>
- 本文流程图摘自：ILRuntime 的 QQ 群的《ILRuntime 热更框架.docx》(by a 704757217)
- Unity 实现 c# 热更新方案探究 (三)：<https://zhuanlan.zhihu.com/p/37375372>

10 remember necessary positoins



11 ILRuntime 的研究

- 借助网络上别人源码分析的步骤，自己（大项目中，以前的小项目源码内容大多已经很熟悉的小项目里找找源码的不算）找一找学习一下追溯源码的过程，去理解整个过程的关键步骤与原理、
- <https://www.igiven.com/unity-2019-09-02-ilruntime/>

11.1 工程运行的入口

11.1.1 HotFixILRunTime

```
public class HotFixILRunTime : SingletonMono<HotFixILRunTime>, IHotFixMain {
    public static ILRuntime.Runtime.Enviorment.AppDomain appDomain;
    void Start() {
        appDomain = new ILRuntime.Runtime.Enviorment.AppDomain(); // <<<<<<<
#if UNITY_EDITOR
        appDomain.UnityMainThreadID = System.Threading.Thread.CurrentThread.ManagedThreadId;
#endif
        TextAsset dllAsset = ResourceConstant.Loader.LoadAsset<TextAsset>("HotFix.dll", "HotFix.dll");
        var msDll = new System.IO.MemoryStream(dllAsset.bytes);
        if (GameApplication.Instance.usePDB) {
            ResourceConstant.Loader.LoadAssetAsyn<TextAsset>("HotFix.pdb", "HotFix.pdb", (pdbAsset) => {
                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();
}
}
}
}
}

```

- unity 工程在执行的时候，会构建一个默认的 appDomain，Assembly.Load，其实就是在那个程序域上加载 Dll，所以相关的实质和前面一个部分相差不大，这就是 c# 热更新在 unity 中的应用 (IOS 不包括)。

11.1.2 LoadAssembly(System.IO.Stream stream, System.IO.Stream symbol, ISymbolReaderProvider symbolReader)

- 基于 WWW 的方式加载 AssetBundle 或者 DLL/PDB 后，接下来是将其封入到 MemoryStream 中，将 dll 和 pdb 的 bytes 都存入到内存流中后，执行其内部实现的 LoadAssembly 方法。

```

// 从流加载 Assembly, 以及 symbol 符号文件 (pdb)
// <param name="stream">Assembly Stream</param>
// <param name="symbol">Symbol Stream</param>
// <param name="symbolReader">symbol 读取器 </param>
public void LoadAssembly(System.IO.Stream stream, System.IO.Stream symbol, ISymbolReaderProvider symbolReader) {

    var module = ModuleDefinition.ReadModule(stream); // 从 MONO 中加载模块 // <<<<<<<<<<<<<<

    if (symbolReader != null && symbol != null)
        module.ReadSymbols(symbolReader.GetSymbolReader(module, symbol)); // 加载符号表

    if (module.HasAssemblyReferences) { // 如果此模块引用了其他模块
        foreach (var ar in module.AssemblyReferences) {
            /*if (moduleref.Contains(ar.Name) == false)
                moduleref.Add(ar.Name);
            if (moduleref.Contains(ar.FullName) == false)
                moduleref.Add(ar.FullName);*/
        }
    }
    if (module.HasTypes) {
        List<ILType> types = new List<ILType>();
        foreach (var t in module.GetTypes()) { // 获取所有此模块定义的类型
            ILType type = new ILType(t, this);
            mapType[t.FullName] = type;
            types.Add(type);
        }
    }
    if (voidType == null) {
        voidType = GetType("System.Void");
        intType = GetType("System.Int32");
        longType = GetType("System.Int64");
        boolType = GetType("System.Boolean");
        floatType = GetType("System.Single");
        doubleType = GetType("System.Double");
        objectType = GetType("System.Object");
    }
    module.AssemblyResolver.ResolveFailure += AssemblyResolver_ResolveFailure;
#if DEBUG
    debugService.NotifyModuleLoaded(module.Name);
#endif
}

```

11.1.3 ReadModule(stream)

```

public static ModuleDefinition ReadModule(Stream stream, ReaderParameters parameters) {
    CheckStream(stream);
    if (!stream.CanRead || !stream.CanSeek)
        throw new ArgumentException();
}

```

```

    Mixin.CheckParameters(parameters);
    return ModuleReader.CreateModuleFrom(
        ImageReader.ReadImageFrom(stream), // <<<<<<<<<<<<
        parameters);
}

```

1. ReadImageFrom()

```

public static Image ReadImageFrom(Stream stream) {
    try {
        var reader = new ImageReader(stream); // <<<<<<<<<<<
        reader.ReadImage(); // <<<<<<<<<<<
        return reader.image;
    } catch (EndOfStreamException e) {
        throw new BadImageFormatException(Mixin.GetFullyQualifiedName(stream), e);
    }
}

```

(a) ImageReader 最终来自 BinaryReader:

```

namespace Mono.Cecil.PE {
    sealed class ImageReader : BinaryStreamReader {
        readonly Image image;
        DataDirectory cli;
        DataDirectory metadata;

        public ImageReader(Stream stream) : base(stream) { // <<<<<<<<<<
            image = new Image();
            image.FileName = Mixin.GetFullyQualifiedName(stream);
        }
    }

    class BinaryStreamReader : BinaryReader {
        public BinaryStreamReader (Stream stream) : base (stream) { }
        protected void Advance (int bytes) {
            BaseStream.Seek (bytes, SeekOrigin.Current);
        }
        protected DataDirectory ReadDataDirectory () {
            return new DataDirectory (ReadUInt32 (), ReadUInt32 ());
        }
    }

    // Summary:
    //     Reads primitive data types as binary values in a specific encoding.
    [ComVisible(true)]
    public class BinaryReader : IDisposable {
        public BinaryReader(Stream input);
        public BinaryReader(Stream input, Encoding encoding);
        public virtual Stream BaseStream { get; }
        public virtual void Close();
        public virtual int PeekChar();
        public virtual int Read();
        public virtual int Read(char[] buffer, int index, int count);
        public virtual int Read(byte[] buffer, int index, int count);
        public virtual bool ReadBoolean();
        public virtual byte ReadByte();
        public virtual byte[] ReadBytes(int count);
        public virtual char ReadChar();
        public virtual char[] ReadChars(int count);
        public virtual decimal ReadDecimal();
        public virtual double ReadDouble();
        public virtual short ReadInt16();
        public virtual int ReadInt32();
        public virtual long ReadInt64();
        [CLSCompliant(false)]
        public virtual sbyte ReadSByte();
        public virtual float ReadSingle();
        public virtual string ReadString();
        [CLSCompliant(false)]
        public virtual ushort ReadUInt16();
        [CLSCompliant(false)]
        public virtual uint ReadUInt32();
        [CLSCompliant(false)]
        public virtual ulong ReadUInt64();
    }
}

```

```

protected virtual void Dispose(bool disposing);
protected virtual void FillBuffer(int numBytes);
protected internal int Read7BitEncodedInt();
}

```

(b) 接下来的 ReadImage 操作:

```

void ReadImage() {
    if (BaseStream.Length < 128)
        throw new BadImageFormatException();

    // - DOSHeader
    // PE                      2
    // Start                  58
    // Lfanew                 4
    // End                     64

    if (ReadUInt16() != 0x5a4d)
        throw new BadImageFormatException();
    Advance(58);
    MoveTo(ReadUInt32());

    if (ReadUInt32() != 0x00004550)
        throw new BadImageFormatException();

    // - PEFileHeader

    // Machine                  2
    image.Architecture = ReadArchitecture();

    // NumberOfSections          2
    ushort sections = ReadUInt16();

    // TimeDateStamp            4
    // PointerToSymbolTable     4
    // NumberOfSymbols          4
    // OptionalHeaderSize       2
    Advance(14);

    // Characteristics           2
    ushort characteristics = ReadUInt16();

    // 这四个操作，是最核心的操作，分别读取 DLL 的 PE 的各个信息，这样我们就进入下一个步骤。
    ushort subsystem, dll_characteristics;
    ReadOptionalHeaders(out subsystem, out dll_characteristics);
    ReadSections(sections);
    ReadCLHeader();
    ReadMetadata();

    image.Kind = GetModuleKind(characteristics, subsystem);
    image.Characteristics = (ModuleCharacteristics)dll_characteristics;
}

```

(c) 最终得到方法的 IL 汇编码

- 让我们分拆来看看这几个读取函数的实现

i. 1) ReadOptionalHeaders(out subsystem, out dll_characteristics)

- 主要读取 PE 的相关信息，不做过多解释，可以参看源码阅读理解；

```

void ReadOptionalHeaders(out ushort subsystem, out ushort dll_characteristics) {
    // - PEOptionalHeader
    //   - StandardFieldsHeader

    // Magic                  2
    bool pe64 = ReadUInt16() == 0x20b;

    //                         1
    // LMajor                 1
    // LMinor                 1
    // CodeSize                4
    // InitializedDataSize     4
    // UninitializedDataSize4
    // EntryPointRVA           4
    // BaseOfCode               4

```

```

// BaseOfData           4 || 0

// - NTSpecificFieldsHeader

// ImageBase            4 || 8
// SectionAlignment     4
// FileAlignment        4
// OSMajor              2
// OSMinor              2
// UserMajor             2
// UserMinor             2
// SubSysMajor           2
// SubSysMinor           2
// Reserved             4
// ImageSize             4
// HeaderSize            4
// FileChecksum           4
Advance(66);

// SubSystem             2
subsystem = ReadUInt16();

// DLLFlags              2
dll_characteristics = ReadUInt16();
// StackReserveSize       4 || 8
// StackCommitSize         4 || 8
// HeapReserveSize         4 || 8
// HeapCommitSize          4 || 8
// LoaderFlags             4
// NumberOfDataDir         4

// - DataDirectoriesHeader

// ExportTable             8
// ImportTable              8
// ResourceTable             8
// ExceptionTable             8
// CertificateTable            8
// BaseRelocationTable        8

Advance(pe64 ? 88 : 72);

// Debug                  8
image.Debug = ReadDataDirectory();

// Copyright                8
// GlobalPtr                 8
// TLSTable                   8
// LoadConfigTable             8
// BoundImport                  8
// IAT                         8
// DelayImportDescriptor8
Advance(56);

// CLIHeader                  8
cli = ReadDataDirectory();

if (cli.IsNullOrZero())
    throw new BadImageFormatException();

// Reserved                  8
Advance(8);
}

```

ii. 2) ReadSections(sections)

- 读取分块数据

```

void ReadSections(ushort count) {
    var sections = new Section[count];

    for (int i = 0; i < count; i++) {
        var section = new Section();

        // 封装一个 Section, 然后去执行读取, 然后赋值给 section 的 Data, 注意回退了 Index
        // Name
    }
}

```

```

        section.Name = ReadZeroTerminatedString(8);

        // VirtualSize      4
        Advance(4);

        // VirtualAddress   4
        section.VirtualAddress = ReadUInt32();
        // SizeOfRawData    4
        section.SizeOfRawData = ReadUInt32();
        // PointerToRawData 4
        section.PointerToRawData = ReadUInt32();

        // PointerToRelocations 4
        // PointerToLineNumbers 4
        // NumberOfRelocations 2
        // NumberOfLineNumbers 2
        // Characteristics     4
        Advance(16);

        sections[i] = section;

        ReadSectionData(section); // <<<<<<<<<<<<
    }

    image.Sections = sections;
}
void ReadSectionData(Section section) {
    var position = BaseStream.Position;

    MoveTo(section.PointerToRawData);

    var length = (int)section.SizeOfRawData;
    var data = new byte[length];
    int offset = 0, read;

    // <<<<<<<<<<<<<
    while ((read = Read(data, offset, length - offset)) > 0) // Read: BinaryReader 里 Read 方法的实现
        offset += read;
    section.Data = data;

    BaseStream.Position = position;
}

```

iii. 3) ReadCLIDHeader(): 基本简单就读完了

```

void ReadCLIDHeader() {
    MoveTo(cli);

    // - CLIDHeader

    // Cb                  4
    // MajorRuntimeVersion 2
    // MinorRuntimeVersion 2
    Advance(8);

    // Metadata            8
    metadata = ReadDataDirectory();
    // Flags               4
    image.Attributes = (ModuleAttributes)ReadUInt32();
    // EntryPointToken     4
    image.EntryPointToken = ReadUInt32();
    // Resources           8
    image.Resources = ReadDataDirectory();
    // StrongNameSignature 8
    image.StrongName = ReadDataDirectory();
    // CodeManagerTable     8
    // VTableFixups         8
    // ExportAddressTableJumps 8
    // ManagedNativeHeader 8
}

```

iv. 4) ReadMetadata()

```

void ReadMetadata() {
    MoveTo(metadata);

    if (ReadUInt32() != 0x424a5342)

```

```

        throw new BadImageFormatException();

    // MajorVersion      2
    // MinorVersion      2
    // Reserved          4
    Advance(8);

    var version = ReadZeroTerminatedString(ReadInt32());
    image.Runtime = Mixin.ParseRuntime(version);

    // Flags            2
    Advance(2);

    var streams = ReadUInt16();

    var section = image.GetSectionAtVirtualAddress(metadata.VirtualAddress);
    if (section == null)
        throw new BadImageFormatException();

    image.MetadataSection = section;

    for (int i = 0; i < streams; i++) // <<<<<<<<<<<<
        ReadMetadataStream(section);

    if (image.TableHeap != null)
        ReadTableHeap(); // <<<<<<<<<<<<
}
void ReadMetadataStream(Section section) {
    // Offset           4
    uint start = metadata.VirtualAddress - section.VirtualAddress + ReadUInt32(); // relative to the secti

    // Size             4
    uint size = ReadUInt32();

    var name = ReadAlignedString(16);
    switch (name) { // <<<<<<<<<<<< 下面的是重点
        case "#~":
        case "#-":
            image.TableHeap = new TableHeap(section, start, size);
            break;
        case "#Strings":
            image.StringHeap = new StringHeap(section, start, size);
            break;
        case "#Blob":
            image.BlobHeap = new BlobHeap(section, start, size);
            break;
        case "#GUID":
            image.GuidHeap = new GuidHeap(section, start, size);
            break;
        case "#US":
            image.UserStringHeap = new UserStringHeap(section, start, size);
            break;
    }
}

```

- 核心是两个操作，一个是 ReadMetadataStream，就是根据不同的标识符来新建不同的存储结构；一个是 ReadTableHeap：

A. ReadTableHeap()

```

void ReadTableHeap() {
    var heap = image.TableHeap;

    uint start = heap.Section.PointerToRawData;

    MoveTo(heap.Offset + start);

    // Reserved          4
    // MajorVersion       1
    // MinorVersion       1
    Advance(6);

    // HeapSizes          1
    var sizes = ReadByte();

    // Reserved2         1
}

```

```

        Advance(1);

    // Valid           8
    heap.Valid = ReadInt64();

    // Sorted          8
    heap.Sorted = ReadInt64();

    for (int i = 0; i < TableHeap.TableCount; i++) {
        if (!heap.ToTable((Table)i))
            continue;
        heap.Tables[i].Length = ReadUInt32(); // <<<<<<<<<<<<<
    }
    SetIndexSize(image.StringHeap, sizes, 0x1);
    SetIndexSize(image.GuidHeap, sizes, 0x2);
    SetIndexSize(image.BlobHeap, sizes, 0x4);

    ComputeTableInformations();
}

```

- 初始化 heap 中的 Table 后，进行一次 Compute，获取 size:

B. ComputeTableInformations()

```

void ComputeTableInformations() {
    uint offset = (uint)BaseStream.Position - image.MetadataSection.PointerToRawData; // header

    int stridx_size = image.StringHeap.IndexSize;
    int blobidx_size = image.BlobHeap != null ? image.BlobHeap.IndexSize : 2;

    var heap = image.TableHeap;
    var tables = heap.Tables;

    for (int i = 0; i < TableHeap.TableCount; i++) {
        var table = (Table)i;
        if (!heap.ToTable(table))
            continue;

        int size;
        switch (table) {
        case Table.Module:
            size = 2 // Generation
            + stridx_size // Name
            + (image.GuidHeap.IndexSize * 3); // Mvid, EncId, EncBaseId
            break;
        case Table.TypeRef:
            size = GetCodedIndexSize(CodedIndex.ResolutionScope) // ResolutionScope
            + (stridx_size * 2); // Name, Namespace
            break;
        // 中间省略无数步
        default:
            throw new NotSupportedException();
        }

        tables[i].RowSize = (uint)size; // <<<<<<<<<<<< 然后填充 size:
        tables[i].Offset = offset;

        offset += (uint)size * tables[i].Length;
    }
}

```

- 基于这四步操作，我们可以将 IL 的汇编码存储到 Image 中，然后进一步执行后续的 CreateModule 操作:

2. CreateModule 操作:

```

public static ModuleDefinition ReadModule(Stream stream, ReaderParameters parameters) {
    CheckStream(stream);
    if (!stream.CanRead || !stream.CanSeek)
        throw new ArgumentException();
    Mixin.CheckParameters(parameters);
    return ModuleReader.CreateModuleFrom( // <<<<<<<<<<<
        ImageReader.ReadImageFrom(stream),
        parameters);
}

```

(a) CreateModuleFrom(Image image, ReaderParameters parameters)

```
public static ModuleDefinition CreateModuleFrom(Image image, ReaderParameters parameters) {  
    var module = ReadModule(image, parameters); // <<<<<<<<<<  
  
    ReadSymbols(module, parameters);  
    if (parameters.AssemblyResolver != null)  
        module.Assembly_resolver = parameters.AssemblyResolver;  
    if (parameters.MetadataResolver != null)  
        module.Metadata_resolver = parameters.MetadataResolver;  
    return module;  
}
```

- 具体过程步骤如下：

```
public static ModuleDefinition CreateModuleFrom(Image image, ReaderParameters parameters) {  
  
    var module = ReadModule(image, parameters); // <<<<<<<<<<  
  
    ReadSymbols(module, parameters);  
    if (parameters.AssemblyResolver != null)  
        module.Assembly_resolver = parameters.AssemblyResolver;  
    if (parameters.MetadataResolver != null)  
        module.Metadata_resolver = parameters.MetadataResolver;  
    return module;  
}  
static ModuleDefinition ReadModule(Image image, ReaderParameters parameters) {  
    var reader = CreateModuleReader(image, parameters.ReadingMode);  
  
    reader.ReadModule(); // <<<<<<<<<<  
  
    return reader.module;  
}  
protected override void ReadModule() {  
    this.module.Read(this.module, (module, reader) => {  
        ReadModuleManifest(reader);  
        ReadModule(module);  
        return module;  
    });  
}
```

p 基于 LoadedTypes 来实现反射方法的调用

- 这些，方法学会了就自己去追一追源码，把它们看懂

12 热更新资源加载的过程

12.1 AssetBundleList.txt

- 就是列举了所有资源包（包括热更新程序资源包和真正的材质等的资源包）的列表
- 每一行列举了一个资源包的名称以及细节等等

```
hotfix.dll.ab,a0db62110d9bd581941b02f5f29d9859,24302  
hotfix.pdb.ab,cf5b2a1abd05b962cedf3a5081e0e1dc,11603  
scene/config/typeone.ab,ed121261eb85d9da9bc4f55e1a4f1180,1907  
// ....
```