**工作笔记**

1：看了一篇介绍unity制作<愤怒的小鸟>对的文章，里面用到了Physics Material 2D，本以为是个脚本去Inspector界面找脚本，后来在unity官网上看到了用法，原来是一个内置的asset，通过Assets/目录下创建。不管何时，遇到unity里的生僻知识点，第一时间先去官网上找介绍，这个才是最靠谱的。

2：在做扔雪球界面时，界面的初始化阶段需要读取当前设备的分辨率，我之前在电脑上都是用x对应Screen.width，y对应Screen.height，到手机上后发现启动app时的竖屏横屏会影响到读取到的两个值得对应关系，也就是说在app启动时手机的方向决定了width，height是什么？最后把这两个值里较大的一个对应x，较小的一个对应y，最后解决了问题

3：在做扔雪球界面时，要把触摸点的屏幕坐标转换到ngui坐标下和Transform做逻辑，开始用Gesture插件，最后发现这东西不支持ngui，最后要把Unity.Touch这个结构wrap出来给Lua用，可按照之前的方法却没有生成wrap文件，甚至导出类列表里都有一个Unity.Rect同样的结构，最后发现这个导出设置并没有真正的生成Unity.Rect的wrap文件，tolua工具里有一个UnityEngine的目录，里面重写了一些Unity常用的类，其中就有一个Rect和一个Touch，找了半天竟然这里，折腾这么久不应该

4：在获取触摸点屏幕坐标时，祥志分享了一个方法，UICamera.lastWorldPosition，还有一个UICamera.lastEventPosition，这两个方法是NGUI的UICamera提供的静态方法。前一个是记录上次点击的世界坐标，再配合Transform的成员方法InverseTransformPoint可以把一个世界坐标转换到当前transform的局部坐标下，用于判断是否触摸到某个控件区域非常好用

5：扔雪球的三个特效拿到后导入的界面上，RenderQueue也设置正确了，在Scene视图下可以看到特效，但在Game视图下就是看不到。层级，缩放也都检查了，就是不行，最后才意识到是特效的层不对，把特效的层设置到UI层上问题解决了。

6：还是扔雪球的某一个特效，导入到界面看不到，最后发现是缩放的问题，特效太小了以至于放大100倍才看到。

56Ex：关于显示的问题，你自认为显示器上能显示出来的东西却看不到，这时候就涉及到一个物体渲染到屏幕上的整个流程。物体有一个所属Layer属性，每个相机也有一个MaskLayer属性，得对应起来才行。相机和相机之间也有一个遮挡关系，通过Camera组件的ordre字段控制，相机还有一个看见深度，编辑器下调整到3d模式查看相机的深度是否覆盖到了物体，如果在Camera Preview里看到了物体却在显示屏幕上看不到，说明是相机的order有问题，可以调整order或者调整MaskLayer。还有可能是物体本身的缩放问题，双击一下看看物体到底在哪，位置是不是在相机视野内

7：由此可以总结，在2d界面上放置3d特效，在和3d特效制作人沟通以及导入到界面时的一些常见注意项。特效最好以unitypackage的形式给，直接导入到界面一个prefab，特效prefab的位置什么的要保证合适，大小也要合适，还有prefab的层级设置。导入到界面后在Scene视图下可以看到特效但在Game视图下看不到则说明是prefab层的问题，如果在Scene视图下都看不到尝试双击prefab看一下是不是特效本身很小。

8：在一个新项目里，代码完全不熟悉的情况下，学会使用全局精准搜索和模糊搜索。我想在代码里找Update方法，却在全局里搜不到，因为真正的方法是UpdateBeat，而我的搜索筛选条件又是区分大小写|全字匹配，所以搜索不到。因此：搜索代码里的变量在哪里用过，用全匹配，尝试性的搜索一些功能，用模糊搜

9：在任何时候遇到一个新的脚本，新的类，新的方法，去unity官网查看是最靠谱的，如果搜索不到则可能是搜索出现了偏差，这时去谷歌下这个东西到底是啥，干嘛的。看了一篇介绍unity制作<愤怒的小鸟>对的文章，里面用到了Physics Material 2D，本以为是个脚本去Inspector界面找脚本，后来在unity官网上看到了用法，原来是一个内置的asset，通过Assets/目录下创建。不管何时，去找官网上的介绍，这个才是最靠谱的

10：扔雪球界面有一个tween动画，用同一个脚本执行一次放大和一次缩小，两次过程分别对应一个回调，这种情况下会出现递归调用类似的现象，只是在Lua上测试结果如此，没有在c#上测，最后把两个动画分别交给两个脚本，这才解决

11：纠结于为什么是这个图片的尺寸大小，其实是UIRoot脚本的宽高，再加上UICamera的size，这刚好是背景图的尺寸大小，也就是说在这个UIRoot设置下，任何设备上显示该界面，雪球的坐标范围都是背景图尺寸大小，至于如何在大分辨率和小分辨率设备下同等效果，和UIRoot的Scale值有关

12：打开LuaStudio写一句Lua代码，执行有运行结果，可我并没有装Lua开发环境（Lua解析器）LuaStudio是怎么做到运行Lua代码的；同理，我们写的lua代码也能在电脑上运行，在安卓苹果设备上运行，这是怎么做到的，是不同的设备都本身支持lua代码的运行？还是C#代码提供了运行lua的支持？

13：一个ab文件被动态加载并在代码里使用的整个流程。》代码里要索引正确的资源名字，》资源策划要将资源正确的上传到资源目录，》收集资源拷贝到UpdateYY/Res目录下（这个过程可能存在不同目录的同名文件覆盖拷贝问题），》上传到本地资源服务器（要确保上传上去了），》在Assets/AssetStreaming目录下的Remote.zl里指定的是本地的资源服务器路径

14：unity编辑器上的Pivot/Center，Local/Global两个设置是有用的，也是容易被忽视的。今天在调整聊天界面时，一个GameObject上挂了一个UITable的脚本，该obj的位置要跟scrollview的panel对齐，因为该obj上没有渲染物件（uiwidget），不好看区域，调成显示位置后发现Transform上的位置数值在改变，但这个Gizmos显示的地方都一样，问题出在了Pivot/Center设置上，当时选择的是Center，而选中的UITable下方有固定数量和固定尺寸的子物体，所以不管怎么改UITable的位置数值，Gizmos显示都不变，因为Center是计算当前物体和所有子物体的中心点，而Pivot则是计算自身的。Local/Global的设置和旋转有关，一般编辑器设置选择为Pivot,Local两个选项

15：调整聊天界面时，聊天框内显示一句话，看起来分明没有填充慢panel，但却可以滑动（因为scrollview设置了不满一panel不允许滑动），而且每次向下拖拽的的时候并不会回弹置顶，反而是向下拖拽松手后回弹到文本和panel下边界对齐，向上拖拽松手后回弹到5文本和panel上边界对齐。这里面引出了几个问题：

1：看起来文本并没有填充满一panel，却能拖拽，其实是文本的上边缘已经超出了panel的上边缘，所以是可以拖拽的，也就是说panel内的内容但凡是超出了panel的渲染区域，不管是上超出，下超出，还是上下都超出了，scrollview都会允许拖拽。

2：这时候对于scrollview来说，并不能拖拽，这时即使拖拽了也不能达到想要的回弹置顶效果，scrollview只会回弹到文本内容全部包含在panel区域内为止

3：最后解决这个问题，是把UITable的位置往下挪了一像素

16：Shader是被当做一个asset资源处理，而并不是脚本

17：dangong1和dangong2的spine动画资源苦于不知道怎么打包和加载，后来新鹏说把这两个动画单独做成两个prefab，用obj导出格式，这样动画关联的资源都会存在obj的包里，通过加载prefab通用接口加载设置父节点。

18：shader本身并没有平台区分，它只是一个字符串片段。但shader跟随prefab打包成pc或者ios格式，这时shader也就有了格式区分，pc上显示ios版本的shader，就会是粉色的一片

19：接上面的问题，dangong1和2加载到游戏后发现是粉红色的一片（根源是shader被打包成ios格式，但加载到了pc上），于是乎就重新选择shader，这时虽然粉色一片的问题解决了，但之前调整好的层级却出了问题，怀疑是RenderQueueEx脚本出了问题，后来发现，其实是重新设置shader后，两个动画obj被重新设置了RenderQueue值。

20：测试那边发现，聊天界面上的好友和帮派两个按钮图标在真机上有切边现象，最初怀疑是图集打得问题，尝试修改padding，发现还是解决不了，对于sprite来说没有任何变化，最后去看这张图的inspector视图，发现是这两张图本身有问题，美术给的原图本身就有切边，所以才出现了测试发现的问题。这是关于UI图片图集编辑器设置上的问题里，第一个独立分析追踪并找到问题根源的，值得纪念

21：之前遇到一个问题，ngui打出来的图集尺寸过大，查看碎图资源里是否有大尺寸图片，尺寸超过512的一般要做成纹理加载，不应该再做成图集

22：一张图片的边界区域是空白，那么在inspector视图里的下面看到的尺寸和uisprite组件里看到的尺寸是不一样的，因为在inspector视图里的图片边缘空白区域被裁减过

23：同一个UI前后不同的人负责，新改版的资源名字和之前的对不上，需要程序去添加新的，删除旧的，不能做到同名替换

24：为什么现在一个界面上的点击要放在一个OnClick方法里捕捉，一个界面上的点击需要被事先分配好id，这个工作做起来难以维护，当初采用这种方式是因为使用了lua，才需要把点击事件这么处理么？

25：扔雪球界面加入spine动画后，背景，船，树的动画添加进来，并且用重新设置shader的方法解决了一片粉红色的问题，可到条幅的时候却出了问题，重新设置了shader却没有起作用，最后发现是重设shader的代码里用的是materials，之前的没问题是没有复制新prefab，而条幅是通过一个模板生成很多新的，这时候就需要用到shaderMaterials，总而言之用它就没错

26：策划说遇到了问题，有些任务没有。听描述感觉是数据的问题，但他确认说是最新的数据和资源后，我又看了下代码是否有错误，或者是账号的问题，最后发现都没问题，而且svn目录都更新到了最新，但上传的时候却出了问题，server.exe的端口被占用了，改了之后重新上传资源ok了

27：今天特效说真机上的某一个特效表现和他们制作的编辑器里表现不一致，特效时一个界面特效，放在uiroot目录下的，但特效并没有在uiroot目录下制作，因为涉及到uiroot的大比例缩放，看起来是特效方向的问题，但其本质是大比例缩放引起的视觉效果差异，因此可以得出，以后的特效制作，界面上的特效必须在uiroot下做，并且需要参照着界面背景找准合适的大小

29：今天扔雪球界面的代码和资源全提后，在策划那看效果，由于策划长时间没有更新工程，什么都准备好了后启动游戏发现报错，看不出来是哪里的问题，最后把Client\_Demo目录下的MeData下的文件全部删除，再次启动游戏可以了，看来Client\_Demo/MeData目录下的文件是缓存文件，长时间不更新要删除后重新加载

30：lua在引用unity内置类的时候，有的需要加UnityEngine前缀，有的不需要，这是为什么添加了Spine动画后，也没有加名空间前缀还是可以访问到C#的类

答：tolua在导出C#类的时候会自动带上该类的名空间，所以unity内置的类导出后都会带上UnityEngine前缀，在lua层访问该类的时候用typeof方式，需要添加完整的名空间，如果是用字符串方式则只需要一个类名

31：png/jpg这类是硬盘压缩格式，而dxt/pvrtc/etc这类是gpu压缩格式。抛开unity，一张png的图要在ios上渲染出来，需要把png图从硬盘加载到内存，然而png是不被ios的gpu所识别的，这时cpu会把图片解码成rgba32形式交给gpu渲染，这种情况下（显示gpu不识别的纹理格式）费了cpu也费了内存和带宽。解决办法是，事前把png图片转码成ios的gpu识别的格式（pvrtc）这个转码工具是gpu的厂商提供的，转码后得到一个新的图片文件，这个文件加载到内存由cpu直接交给gpu处理，gpu拿到数据后直接走渲染流程，这是gpu在硬件层次上对该纹理格式（pvrtc）的支持，中间不需要解码过程，所以叫做gpu纹理压缩。回到unity上，我们导入到unity的图片可以是大多数类型，但一般选择的是png格式，从png格式到各种pgu压缩格式，这个过程unity帮我们做了，我们不需要关心。最后我们的纹理资源都是直接面向各种gpu平台的，少了解码过程，还占用了更少的内存与带宽。

32：spine动画的几个资源里涉及到shader，打出ios包后，条幅的动画在ios版本上是粉红色的一片，是因为在重设shader的代码里没有添加IsEditor判断，导致在ios平台上本是ios版本的shader被重设了一次，会出问题

33：lua中的唯一数据结构table，可以作为标准数组使用，下标从1开始并且递增。Table求长度和ipairs迭代器遇到值为nil的结束，所以如果一个table要作为数组使用，其值不能为nil。Pairs迭代器适用于字典形式，遍历所有的value不为nil的key/value对，另外table中的key不允许为nil

34：table的访问和添加字段的常用方法t[“str”]，也可以是t[“1/2”]，甚至还可以是一个变量（变量也可能是一个table）t[val]。

t[“str”]可以转换成t.str，但t[“1”]就不可以转换成“.”形式，t[val]中val本身是一个变量，其不能为nil，否则会报错，只能使用[]形式

35：lua里的module和require搭配使用，module(“name1”) ，require(“name2”)，require搜索名为name2的文件，找到了则加载找不到则报错。Module是把名为name1的文件里的代码“打包”“装进”一个名为name1的表里（此刻还要注意一点，该文件里的非local代码，也就是说经module打包的table，是访问不到其中的local字段的），如果name1和其文件同名，则require函数加载的结果是一个table，如果name1和其文件不同名，则require函数加载的结果是一个boolean值。

36：lua中的函数表示方法，lua中只有值得概念，每一个值有一个所属的类型，table的构造式是花括号{}，函数的构造式是function()…end，这是一个整体语法，返回一个函数“值”，需要用一个变量去接住这个值于是就有了func = function()…end（该写法的一个变种是，类似于语法糖function func()…end），局部的写法是local func = function()…end，至于局部的写法还有同样的变种是local function func()…end。

37：当执行了module(name)函数后，后面的代码里看不到全局环境，也无法访问全局环境的变量，是因为module函数里有一句setfenv(1, M)代码，该句代码的意思是将当前函数的全局环境设置成模块表M，换句话说接下来的看似全局的变量其实都会在M里，可以添加package.seeall可选参数解决这个问题，seeall本质是一个函数，为模块table设置了一个元表，元表的\_\_index字段是全局环境表\_G。所以，一个模块以module(name, package.seeall)形式定义时，require(name)获取到的table，getmetatable(table)就是\_G。所以，创建模块的最好方法还是最朴素的形式，定义local table，table里放置需要的内容，然后返回。

38：当以module(name[, package.seeall])形式创建模块后，又在代码的结尾return xxx，这xxx是前面代码中的一个table，则require(name)获取到的结果不是名为name的表，而是这个xxx。

39：windows自带的记事本软件，打开后写“联通”两个字，保存后，再次使用记事本软件打开，发现是乱码。记事本软件自身默认的编码格式是ANSI，这是windows平台上抽象的字符编码格式，跟随系统语言不同而不同，美国的windows系统上的ANSI对应的是ASCII字符编码，而在简体中文操作系统上的ANSI则是GB2312。第一次打开记事本写下“联通”二字后，点击另存为可以看到编码格式默认的是ANSI，也就是GB2312。好戏来了，GB2312采用双字节编码【1】，具体值，联：C1AA（1100 0001 1010 1010），通：CDA8（1100 1101 1010 1000）。写下“联通”二字后保存下来的就是这一串二进制串，当再次打开的时候，记事本软件会优先判断该字符串编码是否为utf-8编码格式，要命的是这个二进制串恰好符合utf-8的编码格式，utf-8的双字节格式：（110x xxxx 10xx xxxx），仔细看上面的两个串，是符合这个格式的，这时会按照utf-8来提取信息，把x的位拿出来去unicode编码表里找对应的字符，发现找不到，所以显示出来的是乱码。而且微软的utf-8是带有BOM的，这时候该文本被解释成了utf-8格式的乱码，再次转换回ANSI的话仍是乱码，但理论上如果把乱码的utf-8的BOM去掉二进制串还是C1AA CDAB，这时以ANSI格式去解析显示出来的仍然是“联通”二字（在接下来的计划里，写程序验证）

【1】：顺带说一句，GB2312编码格式理论上能容纳8000出头个汉字，实际上记录了6700多个汉字，GB2312是上世纪80年代完成的汉字编码方案。广义上的汉字实在是太多了，简体繁体的，日韩文里的，各种少数民族的等等，极限值目前是在10W量级，汉字有很多种字符编码格式，他们是在不同的年代里被发明出来的，能表示（编码）的汉字集合也是各不相同，所以导致了一个现象，一个汉字在不同的编码上占用的字节数不等，从2-6字节不等，理论上越早的编码方案占用字节数越少，越往后的编码方案要容纳的汉字数量多了起来，自然要更多的字节数来编码。

40：今天在lua官方的编辑器SciTE上写了一个字符串local str = “赵昊杨”，然后求长度打印，打出来的是6。而在VSCode编辑器下同样的操作打印出来的却是9。

分析原因：lua的字符串原生是什么编码的是unicode的还是ascii的？（这里插一句，在C#里求长度打印出来是3，是否可以理解为C#语言原生字符串编码格式是utf-8的，而写C#代码的VS编辑器又是以utf-8编码汉字的，所以会求得三个汉字长度为3，对于C#的求取字符串长度算法和lua是不同的，C#是按照一个utf-8字符来求取的，lua是按照字节求取的，那疑问是：为什么在有的语境下一个汉字求出的长度是2或者3，在utf-8下就是1呢？这除了跟编程语言的原生字符串默认编码格式有关，是否还和写代码的文本编辑器有关，比如SciTE默认的是ANSI，即为GB2312，双字节编码中文，在这种编辑器下写出的lua代码，lua存储该汉字字符串时自然选择双字节存储一个汉字，在VSCode这种编辑器下以utf-8编码汉字，所以lua就需要用3字节来存储一个汉字）

41：根据上面的描述与猜测（猜测要在后续的查阅和写代码测试中验证）思考下面两个问题

1：输入法是如何与各种文本编辑器合作的？

比如在SciTE文本编辑器下输入“赵昊杨”，在VSCode下输入他们有什么区别吗？能猜测到的SciTE是ANSI编码的，而VSCode是utf-8编码的，输入法是事先检测出要输入的文本接收区域所支持的默认编码格式，然后再把对应的字符（汉字或英文字母）的对应的编码交给文本编辑器吗？这样在SciTE就接收到了“赵昊杨”三个字的GB2312二进制编码，而VSCode就接收到了utf-8对应的二进制编码。（这个问题的猜测待验证）

2：复制+粘贴这个连续操作是怎么处理字符串的不同编码格式问题的？

比如我在SciTE里把“赵昊杨”字符串复制出来粘贴到了VSCode的某一个字符串里，对于这个粘贴来的字符串求取长度仍然是9，也就是说粘贴来的字符串已经有了VSCode默认的编码格式？（以上猜测待验证）

42：之前用SciTE练习lua时新建了几个文本命名为luatest1-8.lua，都是用SciTE编辑器打开的，有的写入了纯英文，有的是是掺杂了中文的。保存后，后面再用VSCode打开依次看之前保存的文本格式，下面有趣的事情发生了。

1：luatest.lua文本是Big5格式，文本开头有两个字“继承”，在SciTE编辑器下正常显示，在VSCode下两个中文字显示的是不认识的两个字，但感觉仍然是某一个编码下的正儿八经的字，在notepad++里看到的也是不正常文字，但是切换到简体中文字符编码就正常了。（为什么会出现Big5编码下，用简体中文字符集对应的，一般来说一个字符编码格式就对应一个字符集的啊）

2：luatest2-8.lua里纯英文的文本在VSCode里打开看是utf-8的，夹带着中文的文本在VSCode里打开看是GB2312的。推断出，SciTE编辑器是支持utf-8的，只是它不支持utf-8里的中文而已，所以会把带有中文的文本编码格式设置会GB2312，而且文本编辑器对于正在编辑的文本的编码格式确定也是一个逐步的过程，比如最初写下的都是英文，那SciTE可以准确的将其编码为utf-8格式，后来加入了中文，而SciTE所支持的utf-8又不支持对中文编码，但其支持GB2312编码格式，而GB2312又能全部容纳英文和新收入的汉语，所以就将其格式设定为了GB2312。那么疑问：假如往一个编辑器里输入了一个其不支持的编码格式的字符，编辑器会咋办，它会无能为力的显示为乱码吗？还有它会改变其既有的编码格式吗？

43：大二的时候参加学长学姐的cdio项目组，经常听他们说起，一个汉字占两个英文字符长度，这在中文字符串里求取长度是一个噩梦。出现这个问题的原因有

先拿Lua语言举例

1：SciTE编辑器下的lua代码对于一个汉字的长度是2，在VSCode编辑器下的lua代码对一个汉字的长度是3，lua解析器本身并没有对字符串有更多的解释与处理，它对于字符串的处理依赖于当前代码文本的编码格式。

2：当前语言里的求取字符串长度的函数是怎么定义长度的。Lua是按照字节求取长度的，自然，不同的编码格式下的汉字长度是不同的

再拿C#举例

1：C#在VS的编辑器里的代码文本文件是utf-8 with BOM（微软的习惯，微软嘴里的utf-8就是utf-8 with BOM，utf-8 without BOM会单独列出来），但C#语言本身的字符串表示采用utf-16格式

2：C#求取字符串长度并不是按照字节计算，而是根据字符求取的。

44：UCS-2是早期的Unicode实现，属于定长编码。而utf-16自始至终都是变长编码，事实上根据规范，所有的utf-x编码都是变长编码。C#，java，js等主流语言都是采用utf-16编码，当时语言开发之初，unicode字符集里还没有填满16bit，在当时的人们看来utf-16是定长编码，后来unicode字符集扩展后，utf-16就尴尬了，不是定长，表示一个ascii又得用16bit，但这些语言已经广泛的使用了，没办法迁移到utf-8。Unicode提出使用代理对（surrogate pair）的方式解决utf-16的变长方案。GBxxx系列编码是属于变长编码，一般情况下一个汉字需要两个字节表示，除非一些生僻字需要更多的字节，所以一般的中文网站都是采用GBxxx编码，比utf-8更省流量。

45：lua里的全局变量和局部变量之间，局部变量会覆盖全局变量的定义

46：尽可能的使用local变量，访问更快。在代码里频繁使用的全局函数比如math库的或者table库的，最好在文件开头local一份使用。因为lua代码在交给解析器执行前会被转换成中间代码，中间代码类似于指令，全局变量的访问会单独生成SETGLOBAL/GETGLOBAL指令，而local变量则直接访问。因为local变量存放在了寄存器里，这里的寄存器是lua函数里的寄存器。

47：大量创建的size较小的table时，预先填充好表的大小。Lua中的table由两部分实现，一部分是数组，下标从1到n之间的整数作为数组索引，一部分是哈希，哈希的本质也是一个数组。当往table里添加新的key/value时，如果数组或者哈希表满了，会触发数组或者哈希表的rehash，数组的话是重新申请一块内存，长度是：能使数组部分超过一半的空间被填满的2的最大的幂。然后把原来的数组元素拷贝过来，旧数组丢弃掉。哈希表部分的rehash，因为其本质也是一个数组，同样开辟一块长度为2的size次方长度数组，只不过哈希表并不是直接的复制元素，而是将元素重新通过哈希函数计算key，然后散落到新哈希表里，原来的哈希表的数组丢弃。所以尽量减少动态的往table里添加key/value，不管是以数组形式还是哈希表形式。预先填充好表，或者用临时数据占位都行。

48：t = { [1] = 1, [2] = 2 }，这种形式的table构造式，lua认为是哈希部分的初始式，数组部分的初始式是t = { 1, 2 }。动态的往table里添加元素时，table.insert是数组式，t[#t + 1]是添加到数组部分。t.xxx是添加到哈希部分。如上数组的构造式中有两个元素t = { 1, 2 }，这时t[3] = 3会触发数组部分的重新哈希，新元素有3个，需要长度为4的数组才能使得数组长度超过一半的空间被填满。但如果是t[4] = 4或者t[5] = 5，则不会触发数组部分的哈希，因为lua不认为这是数组部分的增长，而被看待为哈希部分的增长。同理如果一个空table = {}，上来就是t[2/3]，lua也不会看做是数组部分，而是哈希部分

49：table的数组部分填充与扩展是很严格的按照[下标不间断增加]，table的数组部分长度是可以任意值的，只不过是在重新哈希的时候长度为2的n次方，哈希部分的数组长度也是如此。

50：lua的字符串处理函数的返回值，很多都是返回索引，而不是要的子串，因为返回子串的话会额外创建一份子串的内存，库函数只是返回索引，避免了子串的创建。

51：lua的垃圾收集器在一个自动的状态下工作，采用渐进式（分步）收集。它会根据内存消耗的情况，自动的匹配回收工作。内存消耗的快，回收工作也执行的快。垃圾收集器还可以完全手动控制，在性能吃紧的地方停止回收工作，在程序性能不吃紧的地方执行回收工作，这样可以把垃圾回收的开销给均摊下去。还可以设置垃圾收集器的两个参数：1暂停时间，2步进系数

1：自上一次垃圾回收后，隔多长时间执行下一次

2：每步进回收多大容量的内存

总的来说，暂停时间越短，步进系数越大，则回收的越及时，内存线维持在一个比较稳定且低的线上，但这要执行更多次的回收工作，占用了cpu开销。

52：基于unity+tolua#的游戏，tolua#的作用是C#和lua无缝交互，并且尽可能的在交互过程中减少gc。tolua#是一个集成的东西，里面包括了lua源码的实现之一luajit，还有一个tolua.c文件，以及一些跨平台相关的东西。Tolua在项目中被使用是通过库的形式，生成对应平台的库文件，比如windows下tolua.dll，在安卓下是tolua.so，在ios下是tolua.a，在macos下不懂是啥。也就是说一个游戏在运行的过程中，下层有一个lua虚拟机（luajit）在跑，还有一个tolua.c文件在忙碌着C#和lua的交互。

Tolua.c对应一个C#版本的类，名字也叫tolua.cs，C#和lua交互的本质是利用一个模拟的栈进行的，而tolua.c/tolua.cs就是针对栈的各种操作封装与抽象。Tolua#还有一些在C#层对lua的语法抽象，如LuaFunction，LuaTable，最主要的还是对lua执行环境的抽象LuaState。这些C#的类在交互的过程中多对应着一个lua层的实体。

Tolua#是通过静态绑定的方式，给lua提供了访问C#的能力。绑定的代码是由tolua#的编辑器目录代码自动生成的，关键点在于C#层的各种类型是如何导出的，这个导出的过程因不同的C#类型而异。

local trs = m\_trsTree:Find("Scroll\_" .. data.id)

以代码为例，m\_trsTtee在lua虚拟机看来是一个userdata，其有一个元表，里面注册了该类型对应的C#类的方法和变量等等，这些已经注册了的元素就是静态绑定时候注册进去的。在TransformWrap类的Find方法里，检查栈里是否有两个参数，然后取出第一个参数作为Transform类型，取出第二个参数作为string类型，调用Transform的Find方法，得到一个返回值trs，把这个trs通过ToLua.Push的形式压入栈中，整个语句是一条Lua语法里的赋值语句，赋值操作会取出栈顶元素赋给lua的trs变量，这一次压栈一次出栈，做了如下工作，local trs，该trs被lua虚拟机当做一个userdata，为其设置了一个元表，并且元表里的方法都是Transform类型静态绑定时注册进去的那些，下次使用trs变量时，lua虚拟机就知道如果在trs找不到则去其元表里找，trs元表和C#层的Transform类型是一一对应的关系，这个关系是由tolua#框架维护起来的，最终trs的各种调用转接到了其导出文件TransformWrap类的各个类里，该类的方法里全是栈的操作，实现了数据交互。

这样一来，lua代码里和C#关联着的变量全是userdata类型的，其每一个变量都有一个元表，同一类型的userdata共享同一个元表，该元表和C#层的类是一个对应关系，通过对这些lua里的userdata各种操作，先是转接到Wrap类的方法里，最后真正调用了C#的类方法。

53：刀剑手游项目里的主界面上的粒子特效，没有处理与ngui元素的关系，而是取了个巧，把特效的渲染队列设置成了3000，这也是其默认的值，ngui元素的渲染队列也是从3000开始的，所有只要是把粒子特效放在所有ui的最下面，就不会有错。

54：常说的unity的mono内存只增不减，指的是unity向os申请的用于mono内存的部分只增不减，而我们的C#代码使用的mono内存会随着gc减少，但unity不会把内存归还给os。所以要在游戏内处理好mono内存的增长问题，将其稳定在某一个线上，不让其出现峰值，因为一旦峰值，则意味着内存再也回不到os了。

55：mono内存类似于.net的托管内存，在unity中，我们使用C#脚本new出来的引用类型对象存储在mono内存上，而unity工程里Hierarchy视图下，GameObject的整个层级关系的维护所使用的内存在引擎层面，可能是C++维护的内存，还有比如Transform有一个子Transform数组，该数组的内存也是引擎层维护的，因为这些继承自MonoBehaviour的类的创建方式都是遵循unity规则的，而并不是使用new，所以mono内存的大小和GameObject的数量，assets（资源）量没有关系，他只跟C#代码new了多少个引用类型对象有关系。

56：C#中，静态初始化的字符串使用了“常量折叠”原则，字符串内容相同的string变量引用同一份字符串存储地址，而动态生成的字符串则不具备这个原则，即：每次字符串拼接时都会生成新的内存地址，并不检查是否有内容相同的现成的字符串。

57：冷启动：把系统后台进程杀掉后，点击游戏图标重新启动游戏的整个过程。热启动：后台进程还在，从桌面上点击图标启动游戏。

58：加快冷启动，缩小等待时间的方案？

1：尽量少的使用Resources管理资源，如果可以，禁止使用Resources目录管理资源，有一个YAML格式的描述Resources目录下所有资源的文件，该文件会在冷启动时加载到内存

2：使用il2cpp。mono编译模式下，每个.CS文件被当做一个TestAsset，冷启动时全部加载到内存，il2cpp把.cs文件编译成C代码，然后执行C/C++代码的编译，链接，装载流程

3：使用loading场景，该场景尽可能少的关联资源，是冷启动时要加载的场景，并且以后在切换场景时使用。

59：unity作为一个游戏引擎，其绝大多数的引擎功能代码都是C/C++完成的，而unity提供给我们的引擎功能接口却是C#的，这是因为在整个unity技术栈里，C#仅相当于unity游戏引擎的脚本语言，为了跨平台并且为了追求开发效率，最后unity选定了mono和C#。unity的底层功能实现里几乎没有C#代码，而unity的各种功能类接口以C#代码呈现，这类代码都在UnityEngine.dll里，通过反编译看到该dll里的代码，全是extern的，也就是说Transform/GameObject这等核心类全是引擎底层C++具体实现的一个代理。

60：C#类里维护了一个IntPtr来标识C++里的一个对象，所以一个C#的对象在托管堆里本身占用很少的并且是固定量的内存。一张图片在unity里所占的内存属于C++维护的，也叫作资源内存。也就是说我们在做游戏内存优化时，一般是mono内存和资源内存，一个在C#层，一个在C++层。如果考虑tolua#，减少的是我们逻辑代码里C#对象的数量，GameObject和Transform和各种资源的数量并不会减少，所以tolua#减少了一部分mono内存，同时把游戏业务逻辑代码迁移到了lua层，并且lua层的内存属于C范畴，并不会出现mono内存那种只增不减的情况，所以内存非开辟不可的时候，宁可在lua上也别在mono上。Lua可以通过gc把内存线维持在一个较低且平稳的线上，虽然频繁的调用将会加重cpu负担，但内存总之回收回来还给os，但mono却抓住内存不放手。

61：lua和C#的关系，类似于C#和C++的关系，前者tolua#充当了虚拟机，后者mono充当了虚拟机。前者的具体执行在C上，后者具体的执行在C++上。

62：在UIRoot下添加一个UI，在编辑器下该UI能自适应，但打出的PC包里却没有自适应。NGUI的自适应模式有三个，Constrained模式，而PC包里却是Flexible模式，怀疑是在判断平台时对于UIRoot脚本设置的ScaleStyle做了一些规则处理，导致了同样的代码在编辑器下和PC包里出现了不一致的ScaleStyle表现。查看UIRoot脚本，在获取ScaleStyle时，对不同平台做出了不同的返回值处理，导致了在PC包下返回的却是Flexible。如果选择Constrained On Mobiles在编辑器下和在各种真机上是Constrained模式，如果实在PC包里却是Flexible，这个可以改下，把PC包也加进去，但需要修改NGUI的UIRoot下的代码。

63：在做UI资源加载与卸载测试时，发现UnityFont也就是游戏里的ttf字体，其在内存中有两份占用，打出的PC包里也是如此。最初是以为没有找到正确卸载的办法，最后发现刀剑的PC包里也是如此，他们游戏里的ttf字体的内存占用也是两份。再考虑到动态字体的创建于显示规则，两份内存占用可能是unity引擎的考虑吧。

64：先不删除，试试看ab资源以什么形式存在内存里，再尝试使用false/true删除，查看什么现象?

ab在内存里以原生资源形式存在，比如Material，并且只有一个引用。

使用false是正确的，并且Unload之后，ab为null。使用true则会把从ab里加载出来的资源和实例化出来的资源全部销毁掉，使用了true根没加载ab一样

65：当执行ab.Unload(false);ab = null;后。试试看m\_originObject变量是否还能继续访问，初步看，m\_originObjec也要执行删除操作才行？（m\_originObject是ab.mainAsset的引用）

ab.mainAsset相当于ab.LoadAsset，出来的m\_originObject和ab已经完全没有关系，其代表着打包前的原生资源

66：这句代码：GameObject goAtlas = m\_originObject as GameObject;

多次as操作后，内存中的图集纹理并没有多份。Unity内部（C++层面）对于对象和资源的引用关系做了处理，像这种纯粹的资源类型一般都是引用计数的方式处理。而m\_originObject对象本身占用内存并不多，也包括goAtlas对象本身的内存在C++层，在mono上只是一个整数化的地址。

在这里，goAtlas和m\_originObject是同一份内存，虽然对于Atlas和NGUIFont，我们需要的是其上的脚本，但GameObject本身也需要一直保存，等Dispose时再作处理

如果是Material/Font/Texture asset = m\_originObject as Material/Font/Texture;形式，这种原生资源，我们使用的就是m\_originObject，在Dispose时，Resources.UnloadAsset(asset)后，m\_originObject和asset都为null

67：最终得出结论，仅限UI资源相关，暂时的

使用AssetBundle.LoadFromFile加载，获取ab

使用ab析出对应资源，并as转换到具体类型，存储具体类型。

调用ab.Unload(false)，卸载ab

根据资源类型不同，对mainAsset作不同处理，66条即是。

存储的具体类型资源按照引用计数的方式卸载

68：在UI\_Story\_NPC\_Guide脚本里，看到一个Param字段，是直接使用的，并没有赋值的地方。这种情况下应该是在外部通过UI\_Story\_NPC\_Guide.Param的形式赋值的。

69：proto文件定义时，字段的序号不能重复（可以不连续，但不能重复）

70：血条无法隐藏问题，相机视野的负值太大，导致能看到背后的东西，利用相机的裁剪，不用自己判断

71：数据表格目录下的导表bat文件编码格式必须为GBK，每次创建新bat文件时务必用vscode

72：StatusJumpWordData在Init方法里，设置ReadData回调时，该ReadData局部函数，必须定义在Init方法之前，回调这种方法一般都设置成该模块的私有方法，即为lcoal类型，很多通过class构造的“类”里都有local方法

73：MapManager里有一句直接注册回调的代码，该代码没有在哪个方法里，而是在该Lua文件被require时，自动执行的

74：更新svn出现某些dll不可访问，一般是unity打开状态锁住了该dll，需要关闭unity。找到该dll， 删除它，然后再更新该dll

75：伤害跳字相关的修改代码测试完后，提交了所有东西。但下午更新到最新后，进游戏发现没有伤害跳字了，代码没有报错，最后发现是伤害跳字的资源缺失，在鹿鼎记这个项目里，资源缺失问题。因为资源缺失导致的问题，在追查问题的时候很容易被忽略掉，以后要把控制台的三项输出全部打开，一句警告都不放过，一句log也不放过，并不是红色的error才需要引起警惕。

76：在lua代码里，确保自己的代码有访问权限的概念，虽然lua里并没有提供这些语法。比如，要形成固定的命名规范区分私有字段和共有字段；该lua文件内的方法如果不是供外部调用，就设置成local，这样内部调用起来会更快。在lua中，以module方式构造的“模块”，文件内的local变量和方法，都必须定义在使用前，也就是说，local方法和变量都必须定义在文件的头部，并且被调用的方法放在调用者的前面。

77：Lua传给C#一个数组时，lua构造的表要是数组类型的（key从1开始自增），lua中数组的求长度符号#arr，以1开始为key查找，自增到第一个nil为止。所以中间出现数字断档，即使后面还有数字型key，也不会计入长度。至于是否只能用table.insert方式构造的才是lua的数组类型，这个并没有规定。使用table.insert形式构造的表key从1开始自增，也可以使用键值对形式对一个表从key等于1开始自增，赋值。

78：对于lua的table.sort方法，最开始也是获取数组长度，和求数组长度符号相同的操作。所以table.srot方法也是针对于从1开始并且自增的key。

79：对话界面之前是在prefab上设置了sortorder为-2，后来prefab为了拼界面方便给设置成了0，最后任务流程的时候出了问题，prefab上想要的一些效果，最好在代码里做初始化

80：Item.Proto里的RacialEnum字段，设置了default值，可读出来的还是nil，查原因，但是，立绘对话的cameraSize表里没填值，读出来的是0。弄清楚proto的lua版本处理default的机制

81：装备的职业不符判断，跟策划沟通后，发现策划对于种族和职业的规划并没有在相关表格中落实下来，要改动其中的一个表。而且服务器这边并没有同步下来角色数据的种族和职业信息，刚好种族职业表的特殊性，使用了两个id组合的形式作为key。现在服务器传来了正确的种族和职业，反而因为种族职业表的原始数据不够严谨导致以正确的种族和职业找不到对应的静态数据，进而报错进不去游戏。这个问题解决的其实已经很好了，可就差最后一点，没考虑当天晚上改了一半的代码（主要是处理服务器传来的种族和职业数据判断装备的可穿戴提示），而且也发现了获取不到种族职业数据导致出错进不去，只是在本地做了临时判断，没想到第二天别人更到我的最新代码后，也会出现找不到种族职业数据而报错进不去游戏，思来想去就是落下了这个，还是最明显的问题。因为策划当时已经说了，目前只要种族数据，不要职业数据，完全可以把种族职业管理的地方改一下，这样就能获取到正确的种族职业数据。也不用做临时的判断。千不该万不该，，，

82：修改了对话表的proto中的cameraSize字段-》modelSize。本是为了更符合语义，本是一件好事，可却导致了没能及时提交bytes，导致读bytes失败进不去游戏。回想一下，当初改了proto后，生成导表使用的pb文件，改了某一个表的字段，第一次导表时，没发现bytes有改变，确实没可提交的内容。就轻易的以为修改字段并不会对bytes有影响。这实在不该，也是对于导表过程不是非常清楚的原因，才会这么轻易下了错误的判断。

83：proto大家会修改了就提交，影响并不大。当更到自己需要的proto后，生成pb文件的过程中也生成了很多其他的pb，这时候要小心了，这些pb可能导致游戏进不去，比如和现有的代码对不上。最好的处理方式，更到自己要用的proto，生成pb，把连带生成的自己没用到的pb全部回退回去，因为这样是最保险的，pb还是等做功能的人提上来为好。

84：Dialog\_pb文件内网更到最新后，发现proto的某一个字段被删除了，可外网更到最新后却没有发现这个问题，而且发现本地和svn上的版本一致。还以为是svn出了莫名其妙的bug，也想着到底是啥操作才能引起这种Dialog\_pb不对应的bug。最后发现，是外网并没有更到最新，当时只是更了内网，转头来到外网后就误以为外网也更新了。当初就应该想到外网上的本地文件和svn上最新的不一致，本地文件的版本号和svn上的某一个版本那个，该记录为黑体。

85：lua版的Vector3，给的x,y,z参数如果为nil则会默认为0

86：lua是一个可扩展的语言，lua本身的很多功能都是用库的形式实现的，比如lua里的table结构，背后的源码就是一个库ltable.h/c，注册到lua里（在创建一个LuaState时，lua会调用一个openlibs方法，打开/注册所有的C语言提供给lua的库，这些库有的是lua的自身功能一部分，如ltable.c/lstring.c，有的是lua的标准库，如lbaselib.c/lstr.c，这种供lua语法使用的C库文件都是以lib.c结尾。）

87：Lua的所有CApi（提供给lua使用的C库，lbaselibl.ci的代码，C#的wrap.cs里的方法，都是int func(LuaState\*)形式。总的来说，C#的一个类或方法包装后给lua使用，这个wrap.cs类，就是一个来自C#语言的为了扩展lua语法的外部库）

88：当lua代码写下pcall(fun, …)时，解析器解析pcall对应的c代码方法，pcall是lbaselib.c库提供的lua方法，对应的C代码是luaB\_pcall，luaB\_pcall是一个int fun(LuaState\*)类型。当解析器去执行luaB\_pcall时，已经把pcall方法里的参数都压栈完成。在luaB\_pcall里只需要从栈上获取你这个方法需要的参数即可。

89：在tolua里，C#用LuaFunction类表示一个lua方法，而且调用的时候提供Call和PCall两种形式。就算是Call方法里调用的也是lua的lua\_pcall方法。所以在C#层面调用lua方法总是以pcall模式。Lua\_pcall尝试执行一个方法，如果成功则返回0。如果失败，则在栈展开以前，收集调用栈信息，并组织一个字符串压入栈顶。目前在一个UI功能，Onxxx系列方法都是注册到C#的lua方法，如果出错都是这种形式的堆栈信息。如果是纯lua的代码逻辑，并不牵扯到C#调用，或者因为不在unity主线程而导致出错后堆栈里并没有过多的C#代码信息。例如下面的字符串

LuaException: [string "Main"]:81: attempt to perform arithmetic on a string value

stack traceback:

[string "Main"]:81: in function 'foo2'

[string "Main"]:87: in function 'TestError'

[string "Main"]:38: in function 'Test1'

[string "GameStart.lua"]:18: in function <[string "GameStart.lua"]:12>

在LuaFunction.PCall里，除了收集lua栈上的信息，C#也收集了C#栈上的信息，利用这些字符串包装了一个LuaException并抛出。被抛出的C#异常，在unity里打印出了如下完整的调用堆栈

LuaException: [string "Main"]:81: attempt to perform arithmetic on a string value

stack traceback:

[string "Main"]:81: in function 'foo2'

[string "Main"]:87: in function 'TestError'

[string "Main"]:38: in function 'Test1'

[string "GameStart.lua"]:18: in function <[string "GameStart.lua"]:12>

LuaInterface.LuaState:PCall(Int32, Int32) (at Assets/LuaFramework/ToLua/Core/LuaState.cs:739)

LuaInterface.LuaFunction:PCall() (at Assets/LuaFramework/ToLua/Core/LuaFunction.cs:96)

LuaInterface.LuaFunction:Call() (at Assets/LuaFramework/ToLua/Core/LuaFunction.cs:120)

LuaFramework.LuaManager:StartMain() (at Assets/LuaFramework/Scripts/Manager/LuaManager.cs:78)

LuaFramework.LuaManager:InitStart() (at Assets/LuaFramework/Scripts/Manager/LuaManager.cs:31)

。。。前面还有很多，但都不重要了，C#启动lua的地方算作源头。

90：使用xpcall/pcall方法，以保护的方式调用指定方法。这样即使方法内部出错，也不至于影响接下来的其他代码。这在关键的逻辑里很重要，确保不会因为某一处的问题而导致整个程序的异常。

91：module(“name”)以这种方式创建的模块，其在package.loaded里是true，module方法本身会把该文件的全局变量收集起来放到一个table里，并且把这个table以name为key存储到\_G里。

以module(…)创建的模块，其在package.loaded里是一个table，一个正常的模块，module方法并不会把该模块存储到\_G里。造成这两种区别的应该和require有关，更准确的说是和lua文件加载器有关。

require(“文件路径”)加载模块的时候，指定的字符串为文件路径，lua会去找package.path以模式匹配的方式去找lua文件，找不到则去package.cpath以模式匹配的方式找dll。

总的来说，创建模块较好的方式是第二种，而第一种方式应该极力避免才对。

92：在Controller.FreeCamera.cs里，添加了ZoopSpeed等属性，也暴露出来，都提交了。别人更到最新，能看到镜头移动效果。可内网机更到最新后，却在lua里提示找不到ZoopSpeed，查了内网机上的.cs文件和wrap.cs文件，都没有问题。最后在.cs文件里打了log查看，unity编译了该文件，最后在lua里能访问到新添加的一系列属性了。代码虽然是新的，但并没有被unity编译，所以unity用的C#代码的dll还对应的是旧代码的。所以在lua里访问不到。后来回想起，在内网更新完后，直接关了unity，可能并没有让unity完全编译完最新的C#代码导致的。

93：ButtonTween.cs脚本是按钮的缩放动画，最初的bug是，点击按钮关闭界面，再次打开界面时，按钮会执行回弹效果，像是上次关闭前按钮没有执行完的缩放动画在再次打开时恢复播放了。后来在ButtonTween.cs的OnDisable方法里重置各种控制动画的字段为初始值。当初测试的时候发现改好了，就没再多看。直到最后在多选择对话框界面里发现还有这个问题，起初怀疑是多选择对话界面的多个ButtonTween脚本对应了同一个TargetTransform，最后多次试验发现还是没能解决问题。最后新建了一个界面，只有一个按钮，挂在了ButtonTween.cs脚本，发现是正常的。打日志发现出问题的多选择对话界面里，在关闭了界面后，仍然会执行ButtonTween.OnClick方法。而正常的界面里却是正常的顺序，先是ButtonTween.OnClick，再是ButtonTween.OnDisable。最后问祥志，得出结论：挂载ButtonTween.cs脚本的GameObject上还有一个UIEvent脚本（该脚本负责界面里的各种UI事件，OnClick等），这两个脚本在Inspector上的前后顺序导致了这个bug。该GameObject上挂载了BoxCollider，点击后，Unity通过SendMessage方式广播给该GameObject上的所有脚本，定义了OnClick方法的C#脚本都会受到该事件，并且Unity会自动调用OnClick方法。有问题的界面里，UIEvent脚本在前，先得到了OnClick方法，关闭了界面，而ButtonTween.cs脚本在后，后收到OnClick方法，同一帧里OnClick里将点击标志位置为true，但后面的动画因为界面已经关闭不再执行。当再次打开问题界面时，ButtonTween.Update方法里发现点击标志位true，执行了点击的缩放动画。这个bug独立查到打印日志，只是出于对日志惊奇，导致没仔细分析日志背后的逻辑，如果仔细查看日志的调用堆栈，也许会发现这个SendMessage广播事件的机制。遇到bug，不要惊奇于现象，越惊奇越是暴露出自己知道的东西少，当明白了bug的来龙去脉，当储备了引发bug的相关知识点，那bug就不再是bug了。

94：测试发来一个图，看到是HpNameItem\_Monster.ResetTarget方法里出错了，错误是访问的字段\_target为nil，看了代码后发现，前面也有用到\_target的地方，为啥不报错。开始惊奇于bug现象，怀疑是执行到这句问题代码时，\_target已经为nil了，后来查\_target会在OnDie里置为nil。难道是在ResetTarget方法执行的过程中，外部又调用了OnDie方法？惊奇了一阵子后，确认了自己这边没啥问题后想着让主程帮忙看下问题，问他是否会出现一个方法执行过程中，而另一个方法会调用，导致前后状态紊乱。主程在玩游戏，所以没去问，又查了下ResetTarget方法里的问题代码的前一行用到\_target字段的代码，是个方法SetHp。最后发现当ResetTarget方法传来的target血量值为0时，就会必现这个bug。也算是自己代码写的不完善，整个ResetTarget方法都在用\_target字段，而在中间调用的SetHp方法却会有几率的修改该字段，算是代码逻辑不严谨了。最后把SetHp方法的调用放到了ResetTarget方法的最后。即使传来的target血量值为0，那么执行完ResetTarget方法后，最后执行设置血量，直接OnDie，算是完整的一个流程结束。

95：查看了lua的tonumber方法源码，在lbaselib.c库里。如果是number，则直接返回，如果是字符串则尝试转换为number，转换成功则返回，转换失败则不处理，方法最后是pushnil。如果参数为nil则会在checkany里生成一个error字符串，但这个字符串只是被压栈，并没有被取出和打印出来。

96：查看了lua的tostring方法源码，在lbaselib.c库里。尝试调用元表方法\_\_tostring，如果成功则直接返回，如果不成功则继续判断。如果是number类型，则转成字符串后压栈，如果是字符串类型则直接压栈，如果是nil则压”nil”字符串，如果是bool则压”true”或”false”字符串，默认处理是返回类型名字+地址组合字符串。

97：tolua.isnull在ToLua.cs的OpenLibs方法里，里面全是ToLua.cs提供的一系列工具类方法，除了isnull，还有print，typeof，toarray等常用方法。方法内部判断类型是luanil类型则返回true，如果是C#对象则判断是否为null，来返回true/false。理论上，tolua.isnull可以判断lua里的所有对象，只是在判断lua原生对象时不如lua自身的语法方便和效率。在lua代码里对一个C#对象赋值nil，nil压栈，在C#里通过ToObject获取C#对象，如果获取不到则会返回null。

98：运行protoc-gen-lua-master.cmd，cmd里是python gen.py。gen.py的作用是通过protoc.exe生成proto协议文件对应的lua版本和python版本的pb文件。因为lua的pb文件里用的local变量方式存储字段，而lua里一个文件的local变量数量超过200会有警告，所以对于lua版本的pb文件，又经过了一个protoc-gen-lua-tools.exe（这个工具是项目内部的）处理，最终以table的形式取代了local形式。会在工程目录的lua/proto目录下生成lua.pb，在数据表格的ProtoGen/export目录下生成了python版本的pb，python.pb文件用于导表工具使用。

99：protoc-gen-lua.exe是云风团队针对proto协议做的lua版本。protoc.exe是google提供的生成proto协议文件的工具，里面默认支持各种主流的语言版本，但没有lua的。但protoc.exe本身接受一个外部plugin，这个plugin必须是exe形式才行，plugin可以和protoc.exe同目录，也可以指定plugin的目录给protoc.exe。protoc.exe解析proto协议文件，最后启动plugin。在protoc-gen-lua.exe里运行了protoc-gen-lua.py，protoc-gen-lua.py里收到来自protoc.exe解析proto协议文件生成的描述结构，通过plugin\_pb2.py来完成导出lua.pb工作。也就是说protoc-gen-lua.exe工具的核心部分在protoc-gen-lua.py和plugin\_pb2.py两个文件。

100：数据表格/ProtoGen目录下的export.py是导表工具的核心，导表.bat配置各种参数，最后调用export.py。在export.py里，解析子表生成xml，最后合并xml，读取xml生成DialogData.py文件，最后运行DialogData.py文件。在DialogData.py文件里的export方法，对于每一行excel数据都生成了一个data，并用xml里的数据填充data，最后data序列化成字符串，以二进制形式写入了bytes文件。因为Dialog\_pb.py文件和Dialog\_pb.lua文件对应的是同一份proto协议，可以相互序列化和反序列化。

101：冲锋特效一加载，unity就崩溃，最后清空library目录，并将所有的plugins都reinport都不行。最后发现是unity版本的问题。

102：很多UI文件里，既有class，也有local方法，最好让class放到前面，local方法放到后面，class设计成独立的类，不和外部的local和全局方法有强关联，即使用到，也最好是通过外部传递参数给class，这样可以保证class的高内聚。而且local方法里可能会访问到这些class对象，所以，顺序上是class，local方法

103：UI上，在OnEnable方法的最开始注册事件，在OnDisable的最后删除注册事件

104：Behaviour脚本的enabled字段，用来标识该脚本是否处于激活（可用）状态。该字段会影响GameObject.SendMessage方法的结果，SendMessage只会找该GameObject上那些继承自Mono且enabled=true的脚本，在那些脚本里找对应字符串的方法名。通过脚本对象调用某一个方法不受enabled字段的影响。

105：NGUI的UICamera脚本里有处理UI上的各种事件，如OnClick，OnDoubleClick，OnPress等，通过ProcessPress和PrecessRelease两个方法完成。通过各种检测最后落到某一个事件逻辑内，通过unity自带的GameObject.SendMessage方法广播出去，GameObject.SendMessage(funcName, param, options)，该GameObject上的所有继承自Mono的脚本（且其enabled字段处于true）里，定义了funcName方法的都会收到通知并执行该funcName方法。通知顺序是按照在Inspector面板的前后顺序。NGUI自带了一个UI事件脚本，UIEventListener，为某一个GameObject添加各种UI事件，并且UIEventListener内部定义了所有的UI事件方法，这样该GameObject需要什么事件，就添加什么事件，一旦该GameObject上发生了特定事件，特定事件对应的监听方法就会被通知并且执行，前提是该UIEventListener脚本处于enabled = true状态。

106：LUA\_API是lua源码内部的核心api，LUALIB\_API是lua的库代码api，比如lstrlib.c里有一个被LUALIB\_API修饰的方法luaopen\_string。tolua.c里也有很多LUALIB\_API，但同时也有一些LUA\_API。在C#代码里的LuaDLL.cs类里所有调用的extern方法，都是lua的LUALIB\_API。

107：lstate.c里的方法LUA\_API lua\_State \*lua\_newstate (lua\_Alloc f, void \*ud)，通过该方法生成一个lua\_State（Lua虚拟机，lua状态机，lua执行环境，建议使用类名描述，任何汉语描述都可能因语境不同而导致语义不同，即lua\_State结构）。一个lua\_State结构和一个global\_State结构一一对应。对于global\_State来说，该lua\_State是其mainthread，mainthread的lua\_State不会被回收，除非调用LUA\_API void lua\_close (lua\_State \*L)方法。在一个lua\_State里通过执行void luaE\_freethread (lua\_State \*L, lua\_State \*L1)方法得到一个新的lua\_State，在lua中使用协程，其实就是创建一个新的lua\_State。lua\_State也是lua中的一个称为LUA\_TTHREAD的基本类型，其他非mainthread的lua\_State都有生命周期，当不再被其他代码使用时会被回收。global\_State结构处理字符串（保证一个lua\_State内相同的字符串只有一份存储），内存分配策略，注册表，UpValue，垃圾回收机制，lua基本类型的元表。Lua\_State结构处理栈，方法调用，hook，错误处理，还有一个l\_gt（该结构是lua语法里的G全局表）。一个global\_State对应多个lua\_State，其中有一个lua\_State是mainthread。每一个lua\_State都是一个独立的lua代码执行环境，他们拥有各自独立的栈空间。

108：toptipwidget代码里，跑马灯播放一次后，位置没有重置，tweenposition设置了ResetToBeginning，也设置了transform的位置为起始位置。可每次执行完完自右往左的移动动画后，transform的位置就不断的往左移动。最初以为是self.\_contentTrs.localPosition和self.\_fromPos之间是同一个pos，而tweenpos会改变self.\_contentTrs.localPosition进而改变了self.\_fromPos，以至于动画结束后，重置位置没有效果。但vector3是值类型，不存在这种是同一个pos的问题。最后发现self.\_toPos = self.\_fromPos这句代码出的问题。在lua里，这句代码使得self.\_fromPos和self.\_toPos是同一个对象。而在开始移动动画时，根据self.\_fromPos计算了self.\_toPos，这样self.\_fromPos也就成了目标点位置，动画结束后，重置位置确实成功了，只是那时候self.\_fromPos的位置并不是最初的位置

109：最初多立绘对话的镜头移动，经常会出现对话结束后，关闭界面，恢复镜头，之后FreeCamera的三个速度参数非常快，并没有被恢复过去 。可看lastSpeed确实被执行了赋值。问题出在了在对话结束时lastSpeed已经是targetspeed了。最初的做法是，对话开始时，转动镜头，设置结束回调，在回调里恢复三个速度参数。可有些对话很短，快速点击跳过会非常快的又执行了镜头恢复。而对话开始时的镜头转动回调还没执行，再次执行镜头恢复，则第一次的回调再也不会执行了。当镜头恢复时，FreeCamera.Speed参数就是非常大的值，这时候lastSpeed也就成了最初的targetSpeed。即使对话关闭，执行了恢复速度的代码。也并没有真正恢复三个速度参数。

110：在lua里写transform:GetComponent(“UIWidget”)，之前在C#里写的都是获取类型，最初以为是对GetComponent做了重载，点开Component的源码发现，GetComponent方法有重载不同的参数类型，字符串和Type都可以，Wrap文件只是针对不同的重载版本生成了不同的方法。

111：UIPanel组件的sortOrder默认情况下是0，该值是设置给了UIDrawCall的mSortOrder字段。UIDrawCall里用mSortOrder赋给了MeshRender.sortingOrder ，sortingOrder是Renderer类的重要字段。用来控制渲染顺序，和Camera.depth是类似的东西，只是Renderer.sortingOrder的控制优先级较低。对于NGUI来说，控制渲染顺序的有Renderer.sortingLayerName（几乎不用），Renderer.sortingOrder（只能作用于”片”类型的Mesh，一个3D物体的渲染顺序不受Renderer.sortingOrder控制），Material.renderQueue（unity里通用的渲染顺序字段）。渲染顺序控制优先级大概有：Camera.depth > Renderer.sortingLayerName > Renderer.sortingOrder > Material.renderQueue。

112：立绘图片的临时资源（白板）在游戏里显示上和右有边线，最初以为是图片在边缘处有多余像素，让美术改完后，再次导入发现还是有问题。美术那边确认图没问题，应该是图片的设置问题了，最后发现是Texture的设置参数问题，WrapMode默认设置是Repeat，图片的上和右边线是来自下和左边界，之所以之前没遇到这个问题，是因为立绘图片的下和左边界是填满像素的，因为WrapMode是Repeat模式，导致图片在游戏内的显示里上和右边界有了像素。最后改为Clamp（夹紧）模式问题解决。

113：FreeCamera类里，当相机在自动寻路模式中时，对话调整镜头会出现问题，因为UpdateCamera方法里naving为true则执行镜头跟随角色的移动逻辑，所以当naving为true时，对话镜头无效。最初是想在对话镜头调整过程中屏蔽naving设置，最后FreeCamera类的负责程序将updatesmooth字段放到了和naving一并判断，当updatesmooth为true时不进入跟随角色逻辑，只有对话镜头移动结束后，UpdateCamera方法才会进入跟随角色的逻辑里。这样相当于updatesmooth的优先级高于naving。

114：服务器实例化一个消息结构并填充数据，序列化后，把数据流发给客户端。客户端收到后实例化对应的消息结构并反序列化，对于客户端收到的数据流来说，服务器没有赋值的字段，数据流里没有该字段的信息。客户端实例化的消息结构里访问数据流里没有的字段时，返回该字段类型的默认值。在与服务器的消息里如此，在表结构里如此，访问一个表的某一行数据的某一列，如果没有数据，则返回的是该字段类型的默认值。之前一直以为表里没填的数据，lua访问得到的是nil，是看的调试信息里显示的nil，调试信息的变量详细界面里确实也没有该字段的值，但访问的时候会在没有值时返回默认值。

115：在某一次启动unity时，右下角的技能按钮的锚点错位了，那次突然不行了，但其他人并没有出这个问题。最后发现是unity启动前，如果显示的不是Game视图会出现这个问题。Unity获取Game视图分辨率时失败，导致锚点失效。

116：在UI\_XXX代码里，封装了UI组件，这些XXXItem最好不要直接访问Item之外的变量或方法，保持其独立性，降低耦合，需要用到的通过构造函数传参过来。

117：lua里不要用pairs访问proto的repeated字段，因为message里包含各种描述信息，所以其对应的lua表更像是一个字典，而不是一个数组。Repeated也保证了不会在list中出现nil，不影响ipairs的遍历

118：尽量减少代码内的update，最好有一个总update，其他的往里注册，在一个update内循环调用，以保护模式循环调用，保证一个出错影响其他的update。效率上更好

119：UI功能保证：

1表格错误，服务器数据错误不能报错，报log

2强制关闭界面，再次打开要能正常显示功能

3GameInit.InitManager里不能打开UI

4UI打开取数据显示，注册事件，处理事件，刷新UI

5目前，界面关闭时并没有被销毁，对应的lua脚本也处于存活状态，开关界面只是在执行onenable和ondisable，并没有执行oncreate。如果以后界面被关闭后，其lua脚本被销毁了，再次打开要重新执行oncreate，现有代码也要正常工作，不受影响。原则：oncreate方法里只做获取组件并初始化文件前头的local组件和变量的工作，界面的初始设置放到一个方法里在onenable里调用，关闭前的设置放到一个方法里在ondisable里调用。

120：代码准则：

偏基础的模块代码，多检查参数合法性，不返回nil值，返回一个该类型默认值。这样在被上层逻辑调用时，上层代码不用去一味的对参数判空处理了。甚至可以对上层代码保证，返回的值永远是一个合法的值，但不一定是正确的值，非正确返回值时附上一个Log就完美了。

上层业务逻辑代码在获取到一个方法的返回值时，多检查返回值的合法性，不对返回值作任何的假设。上层业务逻辑代码在执行逻辑时尽量不出现nil访问，尽量用log追踪逻辑流程

121：逻辑和资源的关系：

原则上逻辑不依赖资源加载状态，尤其是纯表现作用的资源，在使用的地方执行一次播放操作，代码逻辑对该资源的了解仅限于此，至于该资源是否被加载了，是否准备好播放操作了等等，把细节都屏蔽在资源管理内部。对上层调用来说是透明的。对于和代码逻辑强耦合的资源，需要慎重对待，资源加载回调没回来，则逻辑不展开执行。

123：Mgr和数据的关系：

PuzzleMgr.Init.InitData里用到了PuzzleData，但因为异步加载，访问为nil，数据加载改成同步，或者全部加完给事件

124：