性能优化工具文档

# PPT资料

<https://docs.google.com/presentation/d/16phV0YLh873CtWnBoWoq8t-UuN93uzK5O9MONB4PMOI/edit>

# 分支

KoA：master

注意：

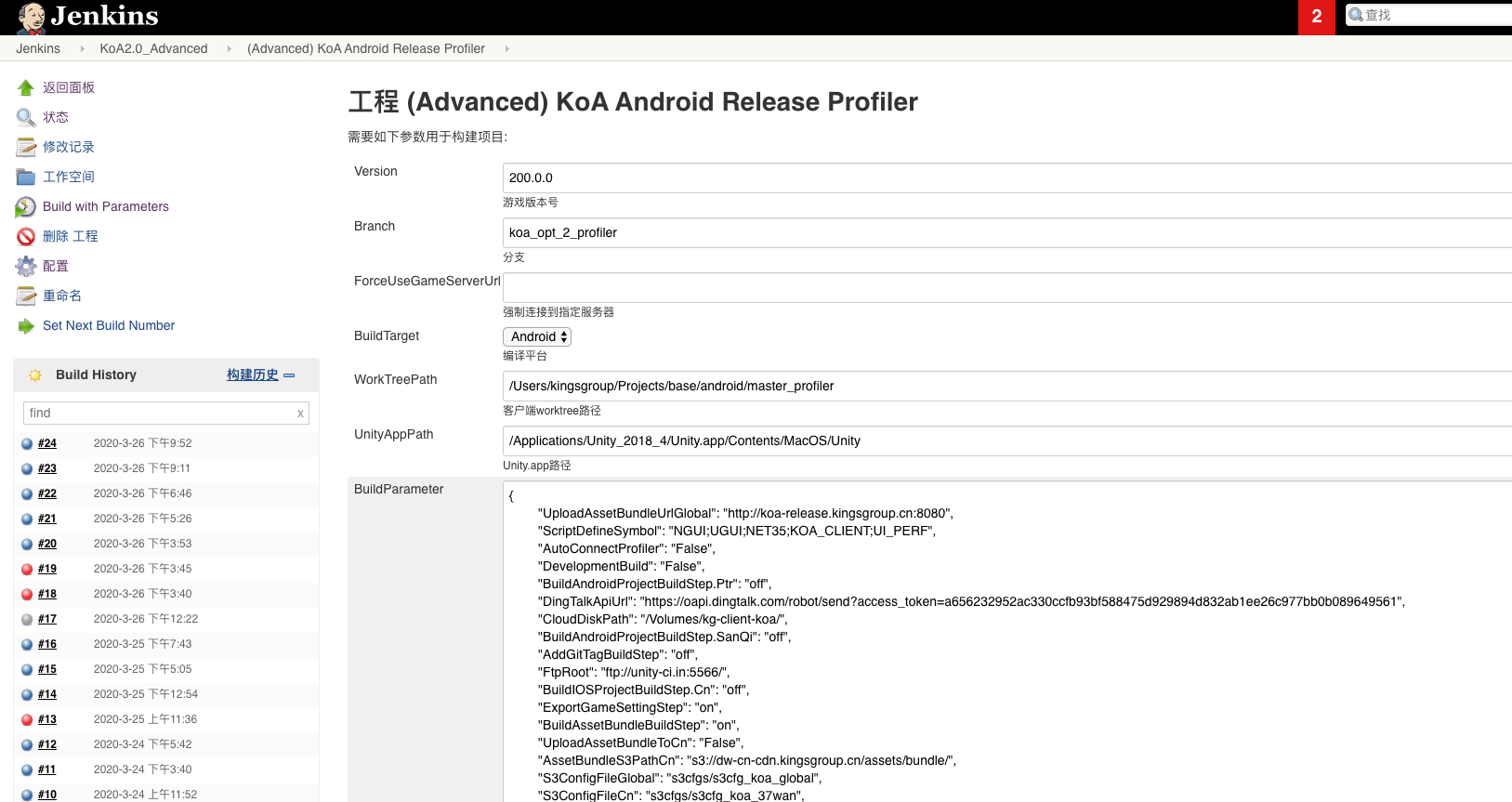
做性能优化的时候，对每个界面，从Master分支拉出专门的优化分支做修改，改好后经过QA测试后再合并回Master

# 出包方法

KoA：<http://10.0.1.164:8080/view/KoA2.0_Advanced/job/(Advanced)%20KoA%20Android%20Master%20Profiler/>

注意：

1. 确保分支选择的是正确的
2. 确保ScriptDefineSymbol中含有UI\_PERF
3. 确保DevelopmentBuild是False，有一定影响
4. 确保DEBUG\_LEVEL\_LOG、DEBUG\_LEVEL\_WARN、UNITY\_DEBUG这三个宏被移除
5. 如果有Bug（例如性能采样的堆栈无限变深）请与潘宇频联系修复。



# 性能测量工具

## 数据采集

出包后，在真机上安装APK，进游戏进行操作。



LoadAllConfig按钮：**读取所有配置表。**移除后面开启界面时配置表读取的消耗影响

* 有些配置表读取需要4秒；全部配置表读取完成在Edtior里面大约4秒；真机数十秒。

SavePerfSate：保存性能测量结果

PerfUpdate：勾上后会开始测量每帧的性能。建议进入想测量的界面后再打开本面板勾选这个，然后几秒后就取消勾选。Update性能测量会产生大量的记录，没必要测量很久。

**以上是老的操作。现在移除了。**

**保存：保存测试结果到Persistent目录下**

**UIOpen：是否测量UI开启时间**

**UIEvts：是否测量UI操作回调**

**PerfUpdate：是否测量一段时间的Update**

自由操作游戏，开启你想看的界面、测量想测试的性能。

结束后，点击保存，会把性能测量结果输出到文件

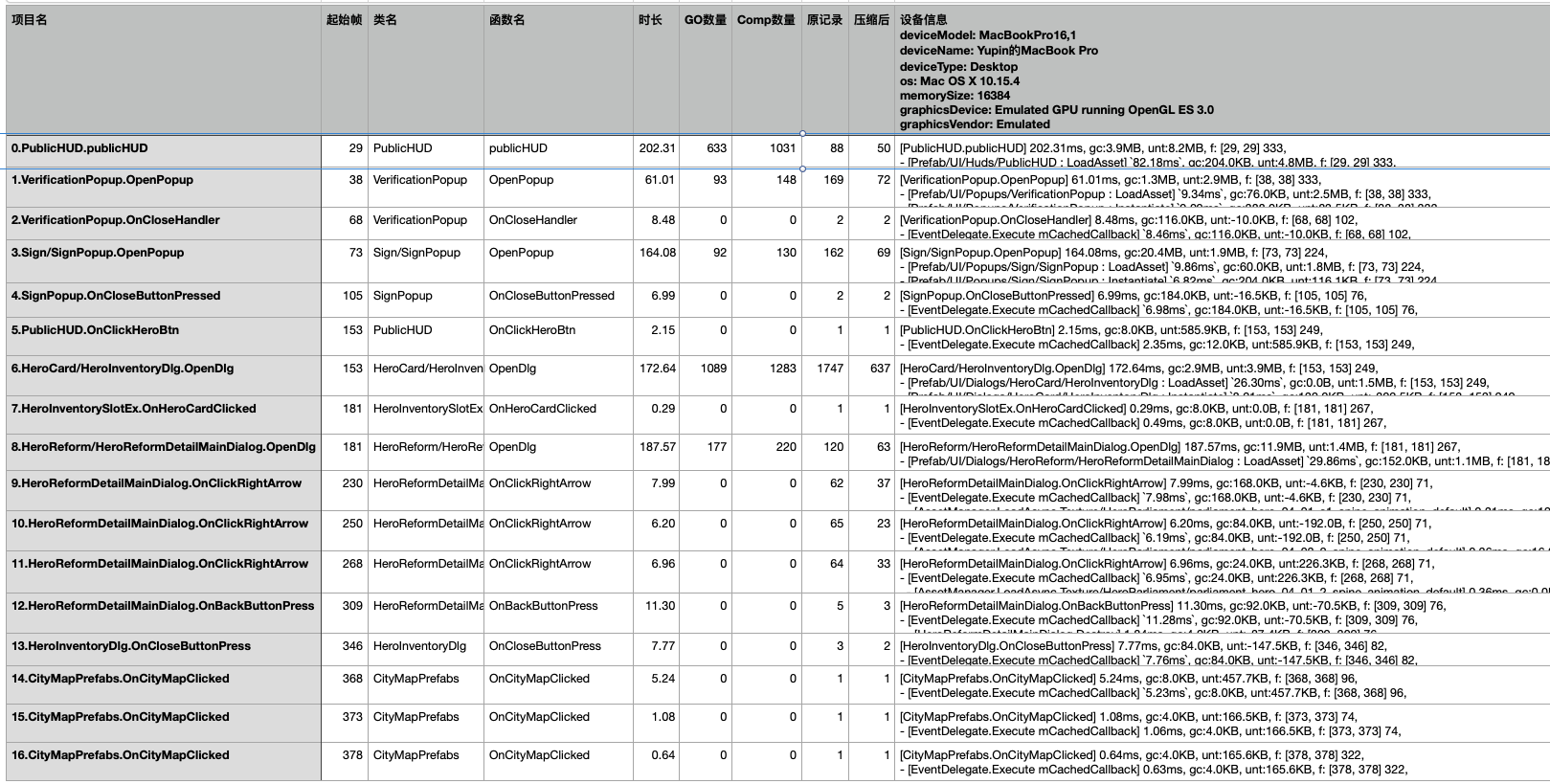
进入真机的Persistent目录，获取数据采集结果

Galaxy J3/Phone/Android/data/com.diandian.gog/files/perf\_stat/

如果没有真机，可把包下载地址告知潘宇频或QA，委托进行测试。

## 数据分析

用GoogleSheet或Excel打开stat.csv文件，可看到每个界面（Dialog和PopUp）开启的耗时、还有每个UI控件的响应函数调用耗时。

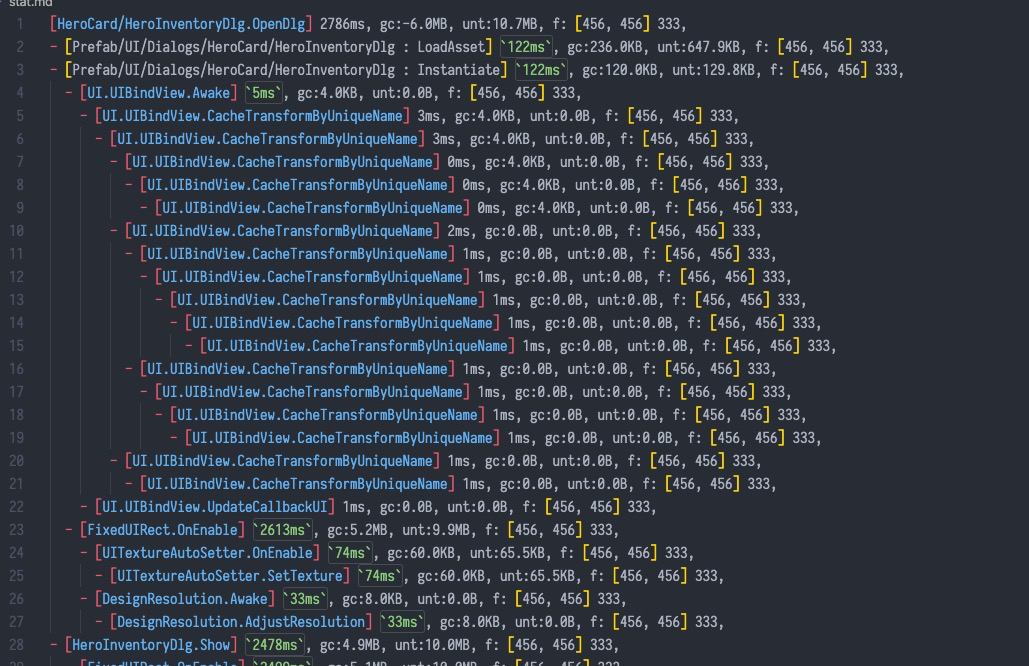


进入stat目录，可以查看具体每一项的函数调用堆栈耗时。

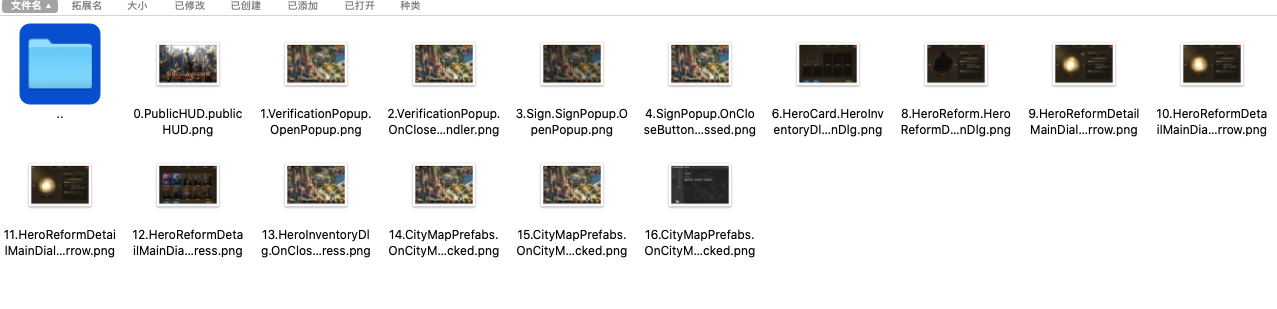


用vscode等文本编辑器打开，开启markdown语法高亮，可以比较方便地查找瓶颈。

输出示例：<https://docs.google.com/spreadsheets/d/1ASXOi5YcsUM_xCtTU6YtmoZiEnP0Z1QAnPv1wAt_fbI/edit#gid=1119580275>



screenshot目录中，是每个界面打开、或UI响应函数完成后的一瞬间的截图，方便查找瓶颈在哪个操作发生



## 测量更多数据

### 代码注入采集——MonoBehaviour默认采集

分析工具会默认采集MonoBehavior子类的所有成员函数。匿名函数除外。

你可以设置项目的黑白名单，来决定哪些MonoBehavior被测量：

* 打开Assets/Plugins/Performance/Editor/CodeInject/UIPerfCodeInject.cs文件
* autoIncludeFilePrefix：这些前缀的cs文件会被采集，其它忽略
* autoExcludeFilePrefix：这些前缀的会被略过

### 代码注入采集——打标签

如果想测量的类不是继承自MonoBehaviour，那么需要手动打上标签、注入性能测试代码。

* 在想采集的类或函数之前，加上[UIPerf]标签
* 在不想采集的类或函数之前，加上[NoUIPerf]标签

快速入门：如果想采集这个类的所有函数，只要在类的开头加上[UIPerf]标签即可；如果这个类有某几个函数调用过于频繁不希望采集他们的数据，那么在这些函数的开头加上[NoUIPerf]标签即可。

注意：目前只有Assembly-CSharp.dll程序集里面定义的类会被注入

### 手动采集

如果希望分析一个很大的函数体内部每一段代码的性能，可以手动加入代码。

用using (PerfMan.Perf("xxxx")) 把想测量的代码包裹起来即可。只有一行无需大括号；多行需要大括号包裹

* 字符串可以固定值，也可以是拼接的结果。可以拼接字符串传入你想关注的信息（例如资源的名字），查看在哪些情况下代码会非常费。

注意：这种采集可以对Assembly-CSharp-firstpass.dll中的类进行，例如Plugins目录下的NGUI

## Editor下获取性能数据

Editor模拟运行并不能反映真实的真机效率。但可以用来查看性能检测代码是否正确注入，看看自己想要的数据是否会被测量到。

建议在真机出包前先本地跑一波看看。

PlayerSettings中，Scripting Define Symbols加**：UI\_PERF**;

在点击顶部菜单：Tools/UI Perf/Inject Code：会自动注入性能采集代码。

点击Play运行游戏，在结束后去礼包码界面点击输出结果

结果输出路径：/Users/yourname/Library/Application Support/FunPlus/Guns of Glory/perf\_stat

### AB模式运行：

首先Jenkins出一个包，记下自己发包的版本号，例如100.0.0

* 目前KoA是8.0.0，GoG是5.3.0

修改本地的Assets/StreamingAssets/version.json文件，改为自己刚才发的版本号，例如：

* {"data":{"minor":"0","major":"1"},"code":{"minor":"0","major":"100.0.0"}}

如果自己在优化、测试过程中改了Prefab内容，那么必须执行上面两步，否则AB不是最新的。如果自己没有改过任何Prefab资源，那么上面两步可以忽略。

PlayerSettings中，Scripting Define Symbols加**：USE\_BUNDLE\_ANDROID**

### Editor模式运行：

Editor模式运行无法测出准确的资源加载的性能（因为资源是直接读取Assets目录下的文件，有的资源读取更慢、有的会更快）

但是用来测量业务逻辑的性能还是有一定价值，可以测试优化前后的代码是否变得更快。

# 引用统计工具

## Prefab引用统计

Prefab的统计形象采用Excel格式输出。MacOS上使用前需要做一些处理。

1. 命令行安装HomeBrew，输入：ruby -e "$(curl -fsSL <https://raw.githubusercontent.com/Homebrew/install/master/install>)"
2. 安装libGDIPlus：brew install mono-libgdiplus
3. 打开Unity的MonoConfig文件，文件位于：/Applications/Unity/Hub/Editor/X.X.X/Unity.app/Contents/MonoBleedingEdge/etc/mono/config，其中X.X.X是Unity版本，加入下面两行：X.X.X是你本地安装的libgdiplus的路径。

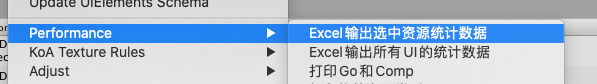
<dllmap dll="gdiplus" target="/Library/Frameworks/Mono.framework/Versions/X.X.X/lib/libgdiplus.dylib" />

<dllmap dll="gdiplus.dll" target="/Library/Frameworks/Mono.framework/Versions/X.X.X/lib/libgdiplus.dylib"/>

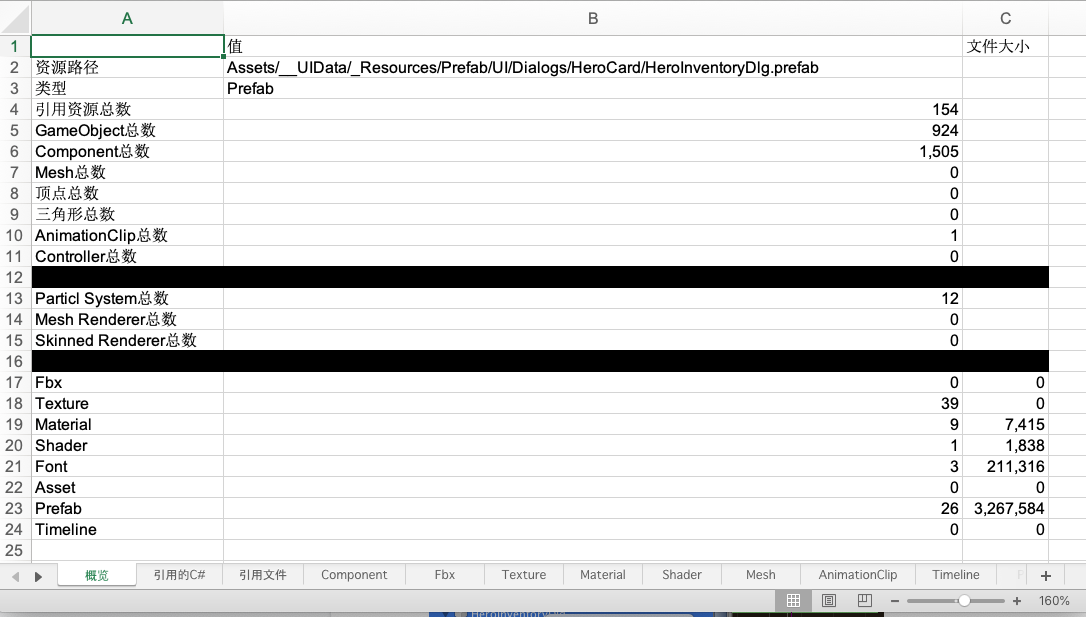
参考：<https://stackoverflow.com/questions/20820139/unity-and-system-drawing-on-os-x>

安装完成后，在Prefab上点击右键，**Performance→Excel输出选中资源统计数据**，即可输出。

Excel输出所有UI资源统计数据，可以批量输出Assets/\_\_UIData/\_Resources/Prefab/UI目录下所有Prefab的统计信息



输出结果示例：

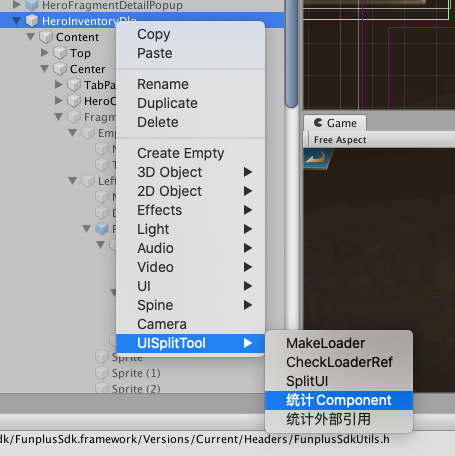


重点关注GameObject、Component、Texture的总数。如果不符合预期，请用下面的方式排查。

## GameObject复杂度统计

在UI场景中欲导出的Prefab根节点上，右键→UISplitTool→统计Component，可以统计这个GameObject及其所有子节点的复杂度，结果会输出到控制台。

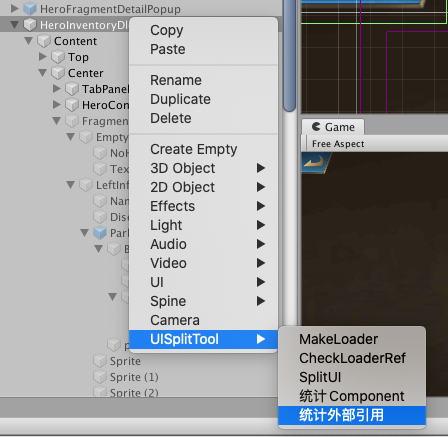
建议在导出前做一次细致统计，确保GameObject和Component数量不超过200。

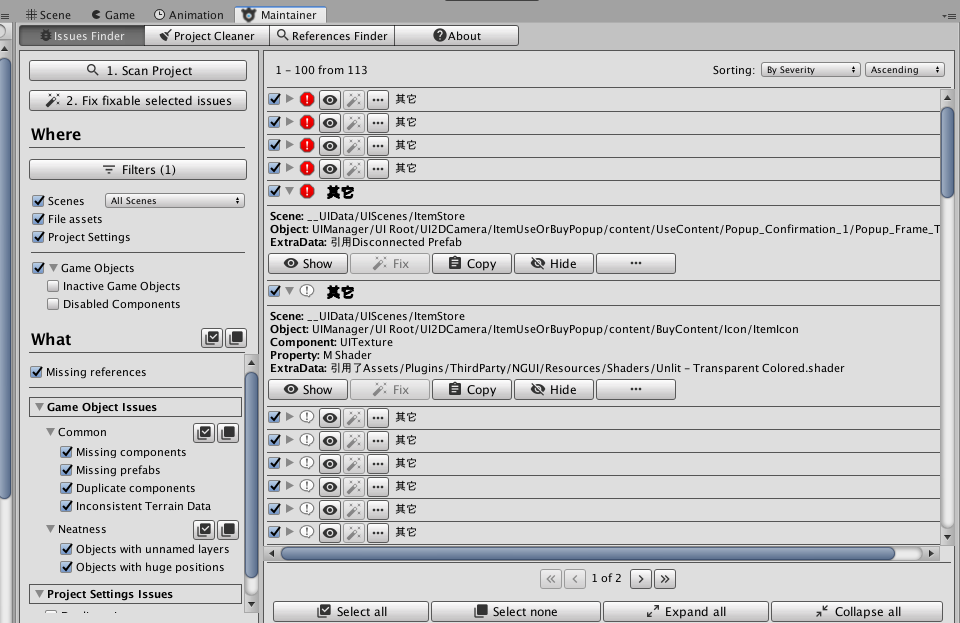


## GameObject外部引用统计 & Prefab错误引用统计

Prefab有两种方式引用外部资源，一种是Component序列化属性中对Object的引用；一种是层次结构上拽入了其它Prefab的节点（有可能不是根节点）。

在GameObject上右键 → UISplitTool → 统计外部引用，会把该GameObject及其递归子节点对外部的直接引用全部输出到Maintainer面板。





点击Show按钮可以跳转到有外部引用的对象。

仔细查找这里面是否有预期外的引用。例如引用不相关的Prefab、不相关的Texture等等。

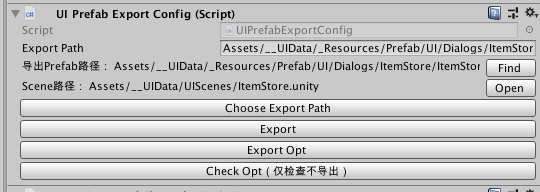
当在GameObject树状结构上引用了其它Prefab的时候（此时GameObject在Hierarchy中是蓝色方块），请一定要引用这些Prefab的根节点，而不能引用子节点。否则会导致引用资源变多、影响加载效率。

使用本工具统计外部引用的时候，如果树状结构中有不正确的Prefab引用，会在控制台输出错误信息。

目前KoA所有UIPrefab中，不正确的Prefab引用统计：<https://drive.google.com/file/d/17fzx3sQZV62dqQDgdO02_e5yli0pfNaO/view?usp=sharing>

# UI导出工具

## 说明



现在UI导出提供了2个新的按钮：【CheckOpt】和【Export Opt】。

Check Opt会对UI做全面的检查，但不会做任何导出Prefab的操作（会自动修复一些错误，建议检查完后存储一下，特别是版本特别老的界面）；

Export Opt会先执行Check Opt，没有Error的情况下再做导出Prefab。

在保留原有的Export功能的前提下，增加了如下新增功能：

1. 导出前会执行错误检查，错误未解决前无法导出
2. 禁止Disable的GameObject（目前是警告，不影响导出；以后会换为Error，程序必须用代码控制）
3. 禁止Disable的Component
4. 禁止Missing的Prefab引用
5. 禁止Missing的Component
6. 禁止同一GameObject下有重复的Component
7. 所有Component的属性中不能有Missing的引用
8. 禁止引用其他Prefab的子节点
9. 禁止Disconnected的Prefab引用
10. 自动移除冗余的资源依赖
11. 禁止引用本场景内、但是非该UI节点下的其它对象（防止在场景中编辑UI的时候胡乱拖拽）（UIManager内置的对象除外）
12. 引用了非static+default AB包中的纹理会给出警告
13. GameObject数量超过200会给出警告
14. 检测除了Common图集外其它Texture引用数量、大小是否超出上限
15. 引用纹理的ReadWrite检查、压缩格式是否正确
16. Prefab必须位于Component目录下
17. 允许Component的白名单，不认识的Component会给警告
18. 有些Component允许多个
19. 与最新版的Maintainer插件集成，错误和警告会输出到列表，可快速跳转到出错的GameObject或Component。部分错误提供一键修复功能。



如上图，点击Show按钮会打开场景、选中出错的GameObject或Component。点击Fix会进行自动修复（如果可以自动修复的话）。

这套框架可检查任意Prefab或Scene是否符合程序规范，非常适合做成按钮给策划或美术使用。文档后面会补充。

# 界面拆解优化方案

界面的加载速度由GameObject和Component的数量、以及引用的其它资源（如Texture）的复杂度决定。

在制作UI界面的时候，UI美术会倾向于把相关联的界面都做在一个场景中，这样很方便排版和预览最终效果。但这个结果是导出的Prefab非常巨大，在低端机上加载耗时超过500ms。

通过统计、分析现有的UI面板，我们拟定优化策略如下：

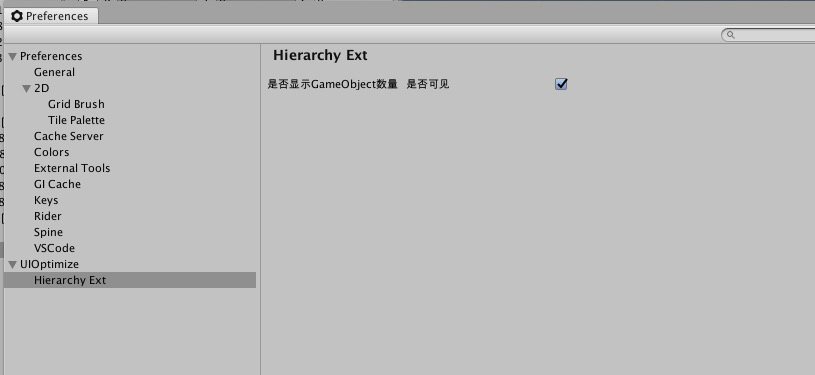
1，对复杂的界面（GameObject数量超过200）做拆分评估。如果界面中含有关联性较小的子面板（例如多个Tab），则优先对这些界面做拆分。关联性非常复杂的面板降低其优化优先级。

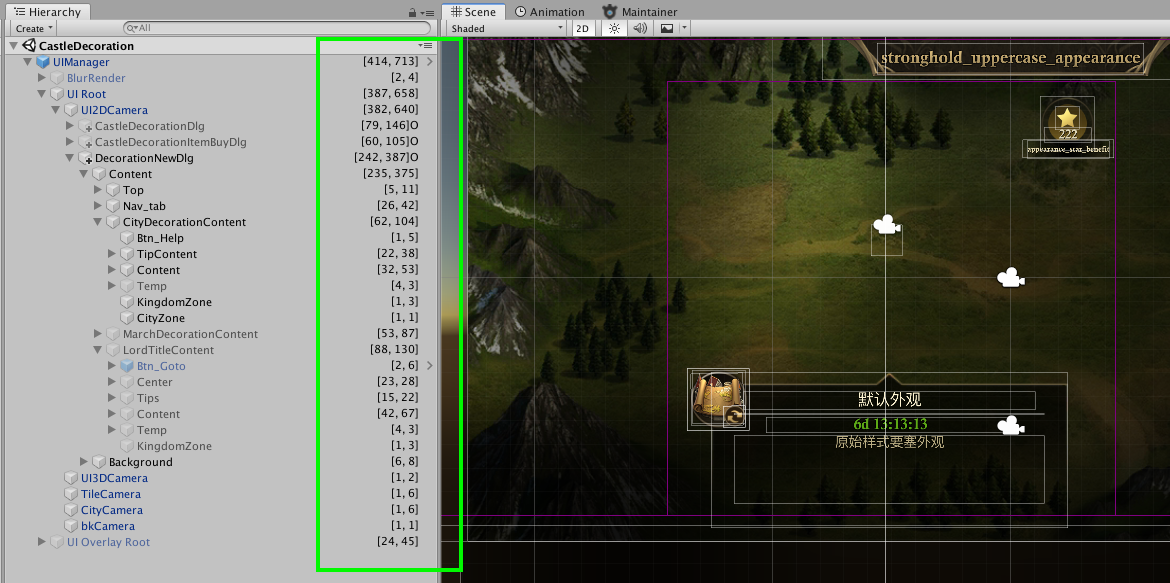
2，开始拆分的时候，首先用工具标记处面板上的子面板、导出成离散的多个Prefab。然后修改代码，引用初始化的时候为空，当需要用到某个子面板的时候，同步加载这个子面板

3，为了更好地体验，可排期把同步加载改为异步加载

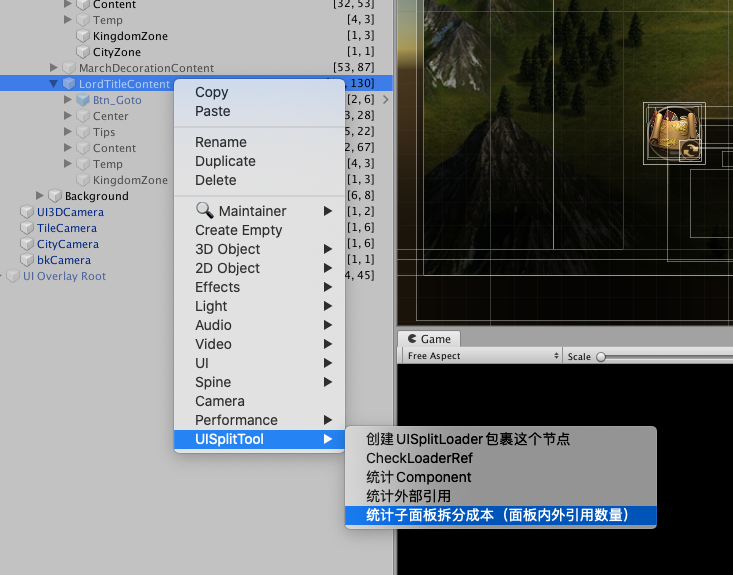
## 拆分复杂度评估

目前有一个编辑扩展，可在场景层级面板中显示GameObject和Component的数量，方便查找热点。可以在编辑器设置中开启这个功能。

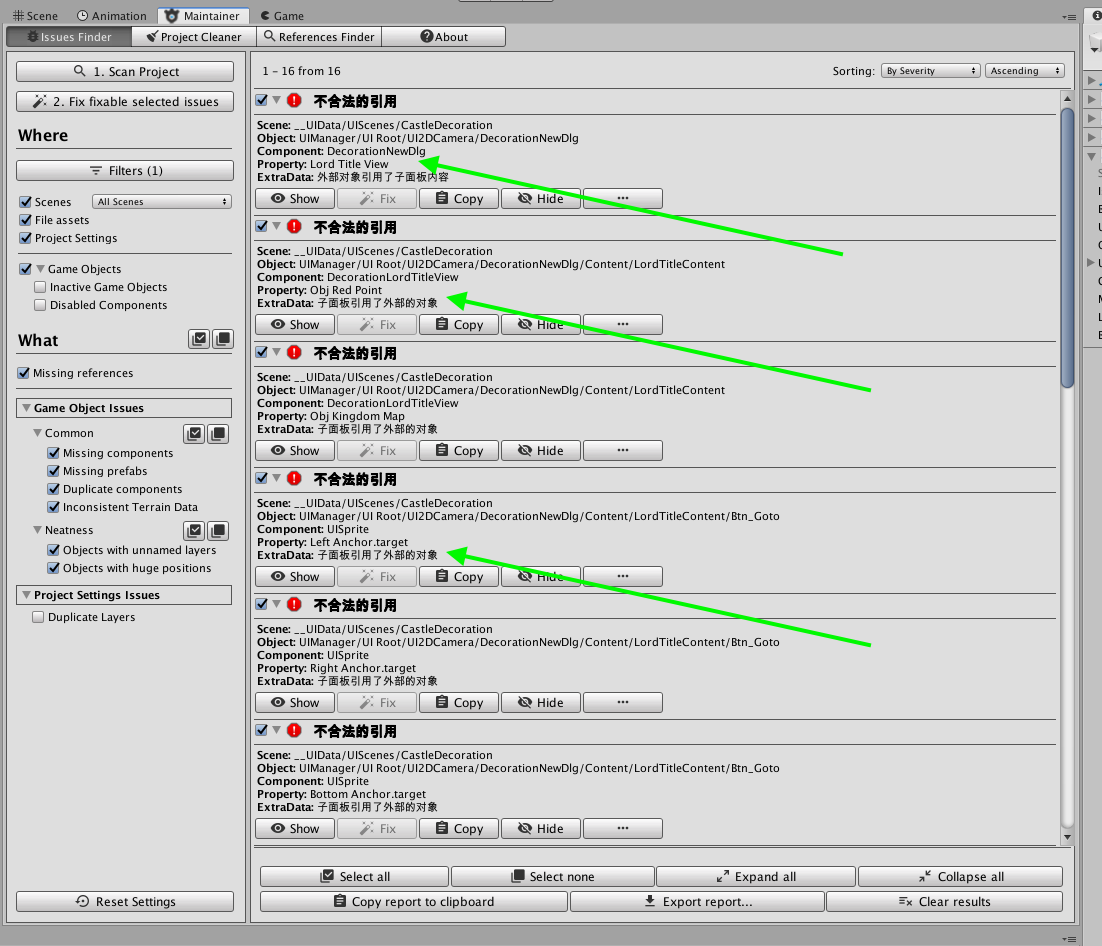




当找到潜在的子面板时，右键点击节点，选择“UISplitTool/统计子面板拆分成本（面板内外引用数量）”



这个命令会分析这个子面板有哪些属性引用了外部的对象；以及外部对象有哪些属性引用了它里面的对象。



比较典型的引用：

* 根面板有属性指向了子面板中的对象，这是因为子面板的对象要在Dlg、Popup的MonoBehaviour类中用到。
  + 优化方案：使用代码生成工具自动生成代码
* 子面板的布局Anchor属性引用了外部的UIPanel等对象
  + 优化方案：子面板增加额外的UIPanel后导出
* 子面板有一个父面板的引用。
  + 切断这些引用，手工加载后赋值

如果除了Anchor之外只有5处以下的内外引用，那么修改的成本是很低的。一般用Tab划分的界面都会很容易处理。

## 界面拆分原理

我们让制作界面的场景还是保留原来的复杂度，方便UE进行界面拼接和排版。

不同的是，在导出的时候我们自动把子面板和父面板分开导出（导出前我们保证父子面板之间没有任何相互引用防止出错）

我们实现了UISplitLoader和UISplitLoaderContainer两个MonoBehaviour类。

这两个类都挂载在父面板中（自动挂上）。UISplitLoaderContainer管理所有的UISplitLoader，UISplitLoader负责把子面板加载出来。

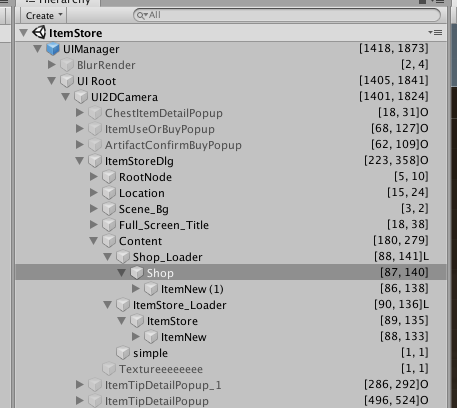
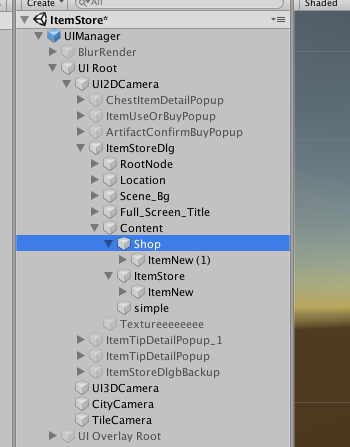
程序在实现面板逻辑的时候，在恰当的时机（例如切换Tab）调用LoaderContainer去加载子面板即可。

Loader支持同步和异步的加载。

## 界面拆分流程

首先找到要拆分的子界面的根节点，右键，点击 UISplitTool/创建UISplitLoader包裹这个节点



这一步会在子面板的节点之外创建一个GameObject把子面板包裹起来，其中会有一个UISplitLoader。如下图：

Loader中是这样的：这些值会在导出的时候填充，不用管它



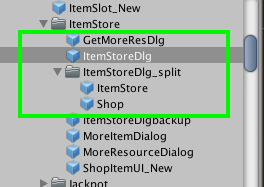
当点击CheckOpt 或Export（Opt）按钮进行UI面板导出的时候，导出工具会在根节点上建立与Loader们的引用关联，方便代码去调用

现在UIBinder、PopUp、Dialog、BaseView等类都已经继承了UISplitLoaderContainer，如果根节点上有这些类就没有问题。

如果根节点上的类不是这些类型，那么需要手动挂上去一个UISplitLoaderContainer

注意：每个Loader下有且只能有1个子节点；同一个子界面中的Loader不可以重名

导出后，在父面板同级目录下会建立名为“父面板\_split”的目录，子面板会放在这里。



在导出后，子面板必须通过代码才能加载。加载代码可以这样写：通过Loader的名字找到Loader对象去加载；

var loader = gameObject.GetComponent<UISplitLoaderContainer>().GetLoader("Shop");

var go = loader.LoadAndInstantiateInScene();

var mShopDlg = go.GetComponent<ShopDlg>();

也可以用下文的UIBinder导出工具，把Loader的GameObject的名字前面加上下划线，直接导出成变量进行访问：

shop = shop\_Loader\_UISplitLoader.LoadAndInstantiateInScene().GetComponent<ShopDlg>();

实例代码可以参考Assets/\_Script/items/ItemStoreAndShopDlg.cs类，这个面板在初始化的时候只加载了一个tab。当用户点击了切换按钮的时候才加载另一个。

## 要点

* 如果子面板在打开的时候不需要Enable，那么一般这个子面板就可以拆出去延迟加载。
* UISplitLoaderContainer.LoadAndInstantiateInScene是直接加载并实例化；UISplitLoaderContainer.LoadAsset是只加载资源，而不进行实例化。后者适合用在Item上。
* Item有时候也是很大、很复杂的，应该放进用到它的子面板里面。或者当做一个单独子面板导出。
* Loader层次结构之间支持嵌套，可以一层一层在运行时按需加载
* Anchor类对父面板的引用，建议在子面板的根节点上加上一个UIPanel组件，以这个UIPanel为anchor做布局

# UI Binder工具

## 说明

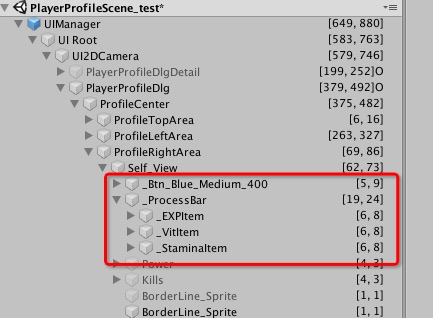
现有的UI制作过程中，需要先在C#代码中声明变量。然后再在Inspector中拖拽，以让代码访问到对象。这种流程有一些缺陷：

* 手工声明变量、手工Editor中拖拽比较繁琐
* Inspector中的引用允许为空，难以判断是丢失了还是刻意为之
* 当界面被UE重构后，对象可能会被删除、引用可能会丢失置空。这样代码可能会访问到null。这种错误直到运行时才会报出。

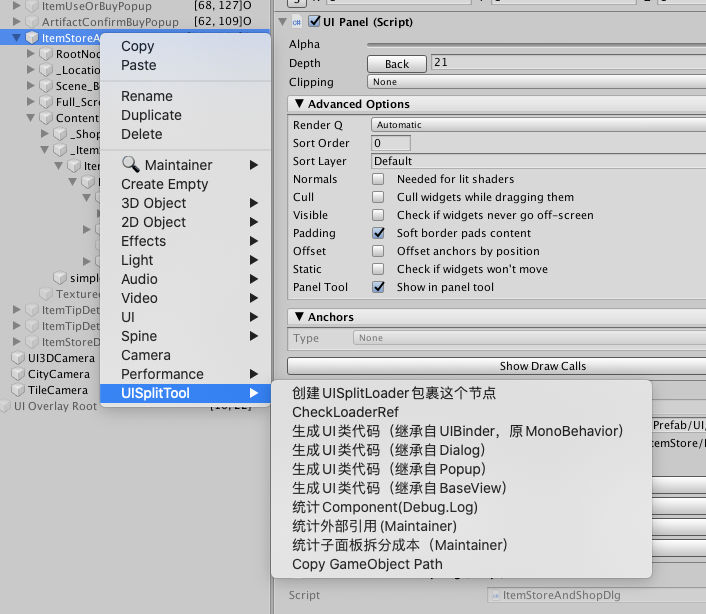
UI Binder工具用来自动建立Prefab中的Object（GameObject或Component）与代码中的变量的关联。它根据Prefab的结构生成C#类代码。在运行时可在代码中方便地访问控件。

## 操作流程

1. 标记想要被代码访问的GameObject。只要GameObject以\_或**m\_**开头即会被生成代码。在UE初次把拼好的UI面板拿给程序的时候，务必做一次这个操作，并告知UE以后不要随意修改已有GameObject的名字。



1. 选择一个GameObject右键移动到UISpliteTool，会弹出多个生成代码的选项：



* 继承自UIBinder：直接继承自UIBinder类。如果原来这里的类继承自MonoBehavior，可以用这个
* 继承自Dialog：如题
* 继承自PopUp：如题
* 继承自BaseView：如题

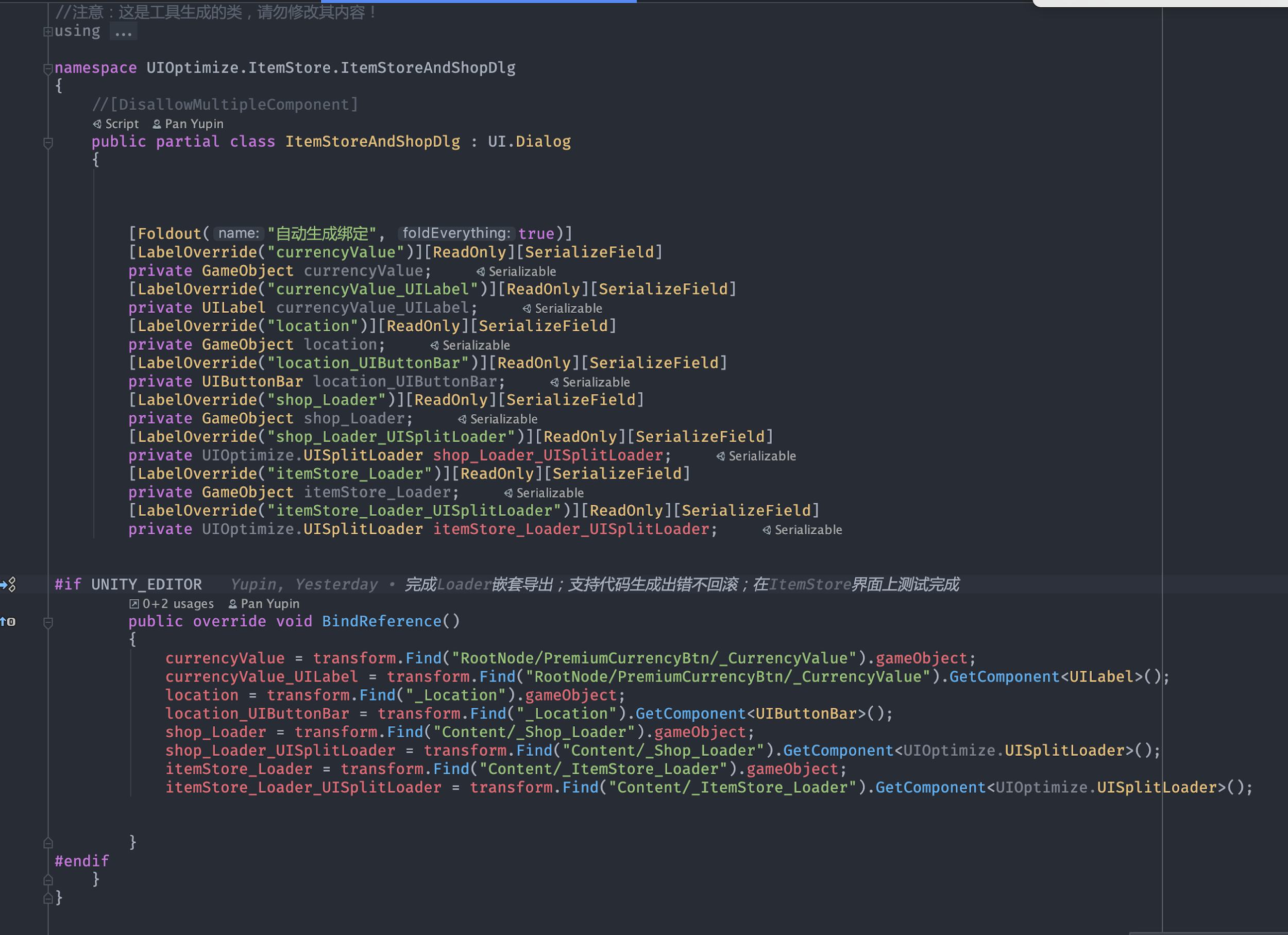
点击后会生成C#类代码、并自动绑定控件。如下图：



生成的类文件会放在这里：Assets/\_Script/UI/UIBinders\_Gen/UI场景名字/导出的根节点的名字/这个GameObject的名字.cs

命名空间是：UIOptimize.UI场景的名字.导出的根节点的名字

实例：



目前生成代码的规则如下：

* Transform、Collider等过于常见的Component（有一个黑名单维护）不会被导出。除此之外的所有Component都会被导出生成代码
  + TODO：未来会改为白名单
* 导出的变量会以 [gameObjectName]\_[ComponentClassName]命名
* 如果同类型的Component有多个，会在变量名后面加上\_x，其中x是序号
* 如果层次结构遍历中遇到了Loader且Loader命名符合导出规范，则Loader对象以及其上挂载的Component会生成Field、其下子节点会被忽略
* 如果层次结构遍历中遇到了UIBinder节点、且UIBinder节点命名符合导出规范（即这个节点既要生成类代码、又会被上层的生成类访问），那么只会生成这个UIBinder节点的GameObject的Field，它本身挂载的Component不会生成Field（这些Field会生成在它自己导出的类上面）

注意：当首次用右键给GameObject生成C#类的时候，同时也会在这个GameObject身上绑上一个BinderGenerator\_XXXX类。



这个组件请不要删除！它表示的是在这个节点上需要生成C#代码、且记录了这个C#类继承自哪里（上图中继承自UI.Dialog）。

后面做UI检查的时候可能要用这个组件去重新生成代码、验证界面的正确性。

每当界面发生改变的时候，可以重新生成绑定的C#类。既可以在GameObject上右键生成，也可以点击上面这个组件的**重新生成UI绑定类**按钮生成，后者方便一些，不用去考虑基类的名字。

## 为老面板生成代码的方法

1. 把GameObject的名字改为C#类的名字（如果这两者不一致的话。有的老面板确实是不一致的。记得改完之后去UIManager里面修改面板的名字，否则Open的时候找不到）
2. 直接在GameObject上右键生成UI类代码。老的界面是什么类这里就选择什么类，MonoBehavior就选择UIBinder。
3. 生成代码后，GameObject上应该就是已经绑了这个Component了，把它删了。
4. 修改代码，把老的C#类放入生成的类的命名空间中，类改为Partial。这样生成的类就和老的类合二为一了（虽然在两个CS文件中）。
5. 回到编辑器，重新点击右键生成UI类代码。这会重新生成C#代码，并建立关联。回Inspector中可以查看是否正确。正常的话，老的面板的引用是不会丢失的。

## 总结

优点

* 省去手动声明类成员的过程
* 省去手动拖拽控件的过程
* 保证所有控件的引用存在、正确
* 在界面被UE和程序来回修改的过程中降低出错的概率。可要求每次UIPrefab修改完界面后必须重新生成一次代码（可以程序做，也可以UE自己做，代码编译不过表示界面被改坏了），可以集成进CI系统中在打包前验证界面的正确性。
* 与运行时根据Attribute或反射查找Field进行运行时赋值相比，这种绑定方法性能更好一点。
* 支持嵌套地生成Binder类，嵌套地挂载Loader类。

坏处

* 自动生成代码耗时较长，因为C#代码太多了，全在Assembly-CSharp.dll里面

# UI公共组件编辑器

UI公共组件指的是位于Assets/\_\_UIData/UIComponents目录下的Prefab。它们可以被拖进UIScene里面，并且会被多个UIScene共用，例如回退、关闭、确认按钮等。

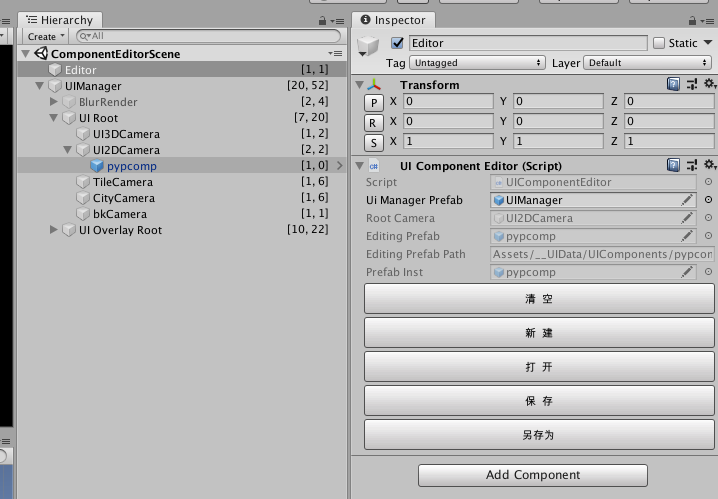
之前没有专用的场景来编辑这些Prefab，也无法对他们做正确性检查。

新的UI制作流程实现了一个专门的编辑器，用来处理这些公共组件，可以用这个编辑器进行创建、修改、保存等工作。

## 编辑流程

编辑器场景的位置：Assets/\_\_UIData/UIComponents/ComponentEditorScene.unity，双击打开。

选中最顶层的Editor节点，Inspector中有一排按钮，提供了编辑器的全部功能。



## HowTo：修改已有的Prefab

选中Editor，点击打开按钮，选择要修改的Prefab，

在Editor中修改它

改完后再选中点击保存即可。注意：现在修改了老的Prefab保存后会报出很多错误，Editor也有一定概率崩溃。但是不影响修改的结果的生效。

## HowTo：创建新的Prefab

选中Editor，点击新建按钮，可以从0创建一个Prefab；也可以点击打开按钮，从一个旧的Prefab修改出新的。

在Editor中编辑、修改它

改完后点击另存为，选择一个目录和目录名保存出去即可。

## 规范性检查

每次保存前都会做规范性检查，检查内容跟UI面板导出基本一致，请在Maintainer面板中仔细阅读、修改错误和警告。

# 特殊规则

UITextureAutoSetter：如果UITexture图片是ota的，那么必须有autosetter；或者被场景引用（因为要设置值）

AutoLoader：挂载特效。如果是ota的，那么应该用AutoLoader加载，否则资源冗余且不能热更新。

# 老界面性能优化指南

* 逻辑优化：用前文的数据分析工具，仔细查找哪些函数调用次数过多、耗时过长，在满足业务需求的前提下优化这些函数的耗时。例如建立缓存、减少列表遍历等等
* 界面拆分：如果LoadAsset耗时过长，则说明界面Prefab过大，需要做拆分。参考上文创建Loader的方法、以及UIBinder类生成的方法，对老界面做改造。
  + 创建Loader包裹，用来分块加载
  + 用UIBinder生成子面板的类代码
  + 可参考ItemStoreAndShopDlg
* 列表改造：UICustomContainer类增加了EnableFraming这个属性，可以让它分帧地创建Item（每帧运行时长不会超过10ms）。把性能较差的列表都改为这个类实现。
  + 有的Item过大的话，可以把Item也用Loader包起来，按需加载放入列表。特别是有的面板中有好几种Item
  + 如果Item上绑了TweenAlpha，UICustomContainer开启分帧后会自动做一个渐现动画
  + 可参考HeroInventoryDlg
* 任何老界面的改造务必使用Export Opt按钮做导出，确保优化后的界面没有任何问题！

# 进包资源检查工具 & 流程

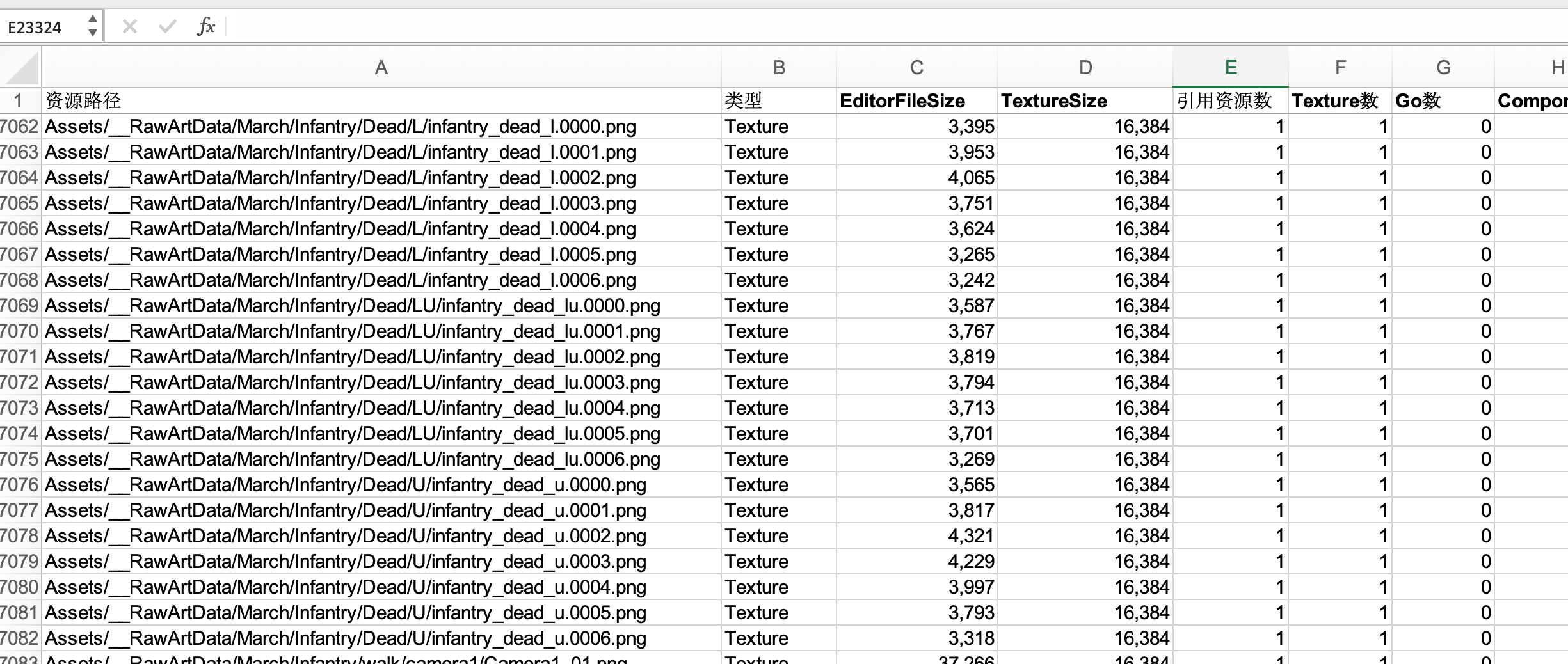
当新功能合并到主干后，有可能会遇到安装包过大的问题。这个工具用来检查到底是什么资源造成了包体过大。

注：给资源做规范性检查、在CI中产出报告更加稳妥。目前还没做到这点，所以遇到包体过大的问题时还是需要专门处理。

在菜单中点击Tools/AB统计工具/统计Static包资源（所有）

该命令会对ABName设置为static+default的所有Asset、以及它们的引用做统计，结果输出到Excel文件。

Excel文件内容如下所示：



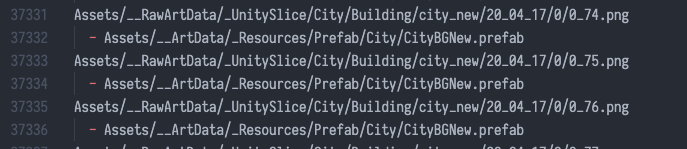
其中，TextureSize是纹理的内存（其实就是显存）占用大小，通过获取Importer里面的format、纹理的长宽进行计算得到。它不等于Assets目录下png或jpg文件的大小（磁盘上的文件是压缩的），也不等于APK中占用的资源大小（同样是压缩的）。但一般包体过大的原因大部分都是由于引入了过多、或者过大的纹理。

此外，如果这个纹理被设置了Packing Tag，那么它会被打进Atlas。请在Window/2D/Sprite Packer工具中把所有图集打出来预览，查看最终的图集大小。打成图集后Alpha为0的部分会被移除，合图大小会比单个散图之和要小一些。

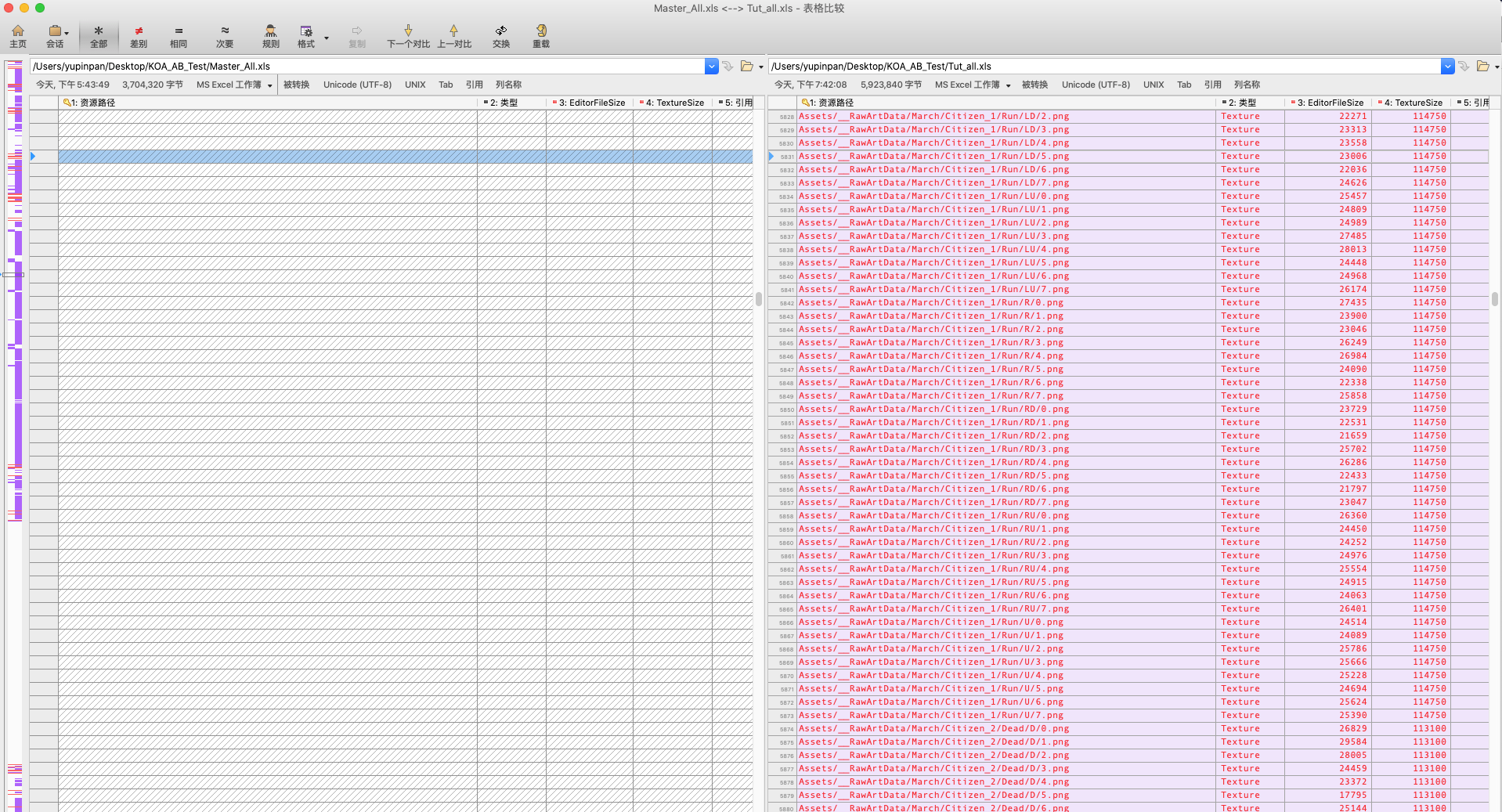
EditorFileSize是这个资源文件占用的磁盘大小。对于Animation、Prefab而言，它的大小与这个资源在AB中占用的大小、加载后在内存中的大小也是不同的（这三者互不相等）。但是一定程度上也能反映资源的复杂度。

如果Animation很大，那么可考虑看看里面是不是有非常密集的关键帧、或者是非常长的动画。Prefab同理。

导出Excel的同时还会输出一个md文件，这个文件记录了每个资源是因为被谁引用而打进AB中。方便查找问题。



对两个不同的版本的项目工程应用上述工具，生成两份Excel文件，然后可以在Beyond Compare中对比两个版本的Excel表格差异。



然后可以去Editor中看这些新增的资源为什么占用这么大的空间。典型的原因：

* 无用的资源进包。解决：切断对这些资源的依赖；清空不需要进包的资源的ab name。
* 资源过大或过多。解决：查看美术制作这些资源的方式是否正确、资源规格是否正确。必要的话应该开发工具检查、或者自动纠正问题。

# 附录

## UI规范

## Prefab规范

* 不允许有Missing的Reference（GameObject、MonoBehaviour）
* 允许的Component
* GameObject上限
* Component上限
* 纹理大小上限
* Particle上限
* 不允许引用其它Prefab的非根节点

## C#代码规范

不允许单个函数低端机运行时长超过50ms（异步、分帧、多线程初始化）

统一规范的写法分配、释放资源（回调、Asset等）

## 性能优化辅助库

UICustomContainer

轻量级协程分帧

## UniRX用法说明和实例

参考文档：

* 王国金的PPT：
  + <https://docs.google.com/presentation/d/1O-zwtJDFsJZ9llKKdhXMeyULFngQxG9uobkQhaEMwjk/edit#slide=id.p>
* UniRX精讲
  + <https://lianbai.github.io/2019/10/25/UniRx%E7%B2%BE%E8%AE%B2/UniRx%E7%B2%BE%E8%AE%B2-001-%E2%80%94%E2%80%94%E7%AE%80%E4%BB%8B/>