UI 模块 ：

1，美术提供一个效果图
2，美术将效果图 切成碎图 。
3，程序拿到碎图 打成一个大图 。
4，导入到Unity3d 里面 TextureImporter 自动切割成碎图 。
一个界面一个大图 。如果是AB加载 将 真个界面用到的素材打到一个AB包里面.

图片的大小：
低端机： 1024*1024 iphon4 同期 以前的机型 。内存 2G 以下 单核
高端机： 2048*2048 内存 超过 2G 双核以上 。

AB 包大小： 一般不超过 1M 1M 左右加载速度是最快的 。

分场景 分功能 ：

Loading:

Regist :

Bag 1:
Bag2:

5，拿到碎图拼界面 。

1，调整画布的大小
2，调整屏幕分辨率的大小。
3，以panel 为编程单位 子控件设置相对父类的锚点 。
4，用碎图拼界面。

拼界面的应该注意的地方：
之前没注意到优化问题 导致batch特别高 游戏卡顿

Canvs重上往下渲染

合并批次

Canvs RenderMode ： Screen Space-overLay 永远贴在最前面|开发纯2d -Camera开发2d和3d混合 world Space cancas就是一个3d物体

游戏一般分为两个canvas 一个动态一个静态

DC: 计算发生了几次 传送指令 dc发生在一帧

重建：

重新在传送一次。顶点的位置发生了变化 就需要重新传送。

动态会发生重建

0层级 { 材质球 1{ image1 ,text ,, button1 } 材质球2 { image2 , button2}}

1层级 { 材质球 2{ button1, text , } 材质球3 { image3 , button3}}

最下面的层级号是在它上面的层级好+1 如果他们俩的材质和贴图是一致的

1,Unity会将每一层的所有元素进行一个排序（按照材质、纹理等信息），合并掉可以Batch的元素成为一个批次

2,Text组件会排在Image组件之前渲染。

0: text-->image-->button -->image2 -->button2

1: text-->button -->image3-->button3

3,，Unity会再做一个优化，即如果相邻间的两个批次正好可以Batch的话就会进行Batch

例1:

一个层级为0的ImageA，一个层级为1的ImageB（2个Image可Batch）和一个层级为0的TextC，

0 : ImageA TextC

1: ImageB

textC--> ImageA --> ImageB

1dc --> 1 dc

例2:
一个层级为0的TextD，一个层级为1的TextE（2个Text可Batch）和一个层级为0的ImageF，

0:　　TEXTD　　ImageF
1:　TeXT E

textD--> ImageF --> TextE　　3DC

优化：　　加一个控件　　反而可以减少DC

0:　ImagG　　ImageF
1:　TeXT E　　，TEXTD

ImageF --> TextE --> TextD

尽量的让可以Batch 放在一个层级 。

3,，Unity会再做一个优化，即如果相邻间的两个批次正好可以Batch的话就会进行Batch

DC：drawcall CPU-->GPU 传输一次数据
指标：正常的游戏在50--120之间　　超过这个值 手机无法运行 非常卡顿

1. 传递 材质球　　先传递Shader 再传贴图
2. 传递顶点

两张图片 顶点可以同时发送过去　但是如果材质球或者贴图不是同一个 ，它不会一次打包

为什么要把贴图打成大图
用的是同一个材质球，同一个贴图 ，顶点才会被打包一次性传送给GPU 。

Batch: 用的是同一个材质球，同一个贴图 ，顶点才会被打包一次性传送给GPU 。
计算层级号的算法:
1,如果有一个UI元素，它所占的屏幕范围内（通常是矩形），如果没有任何UI在它的底下，那么它的层级号就是0（最底下）；
Batch: 用的是同一个材质球，同一个贴图 ，顶点才会被打包一次性传送给GPU 。
2,如果有一个UI在其底下且该UI可以和它Batch，那它的层级号与底下的UI层级一样；
3,如果有一个UI在其底下但是无法与它Batch，那它的层级号为底下的UI的层级+1；
4,如果有多个UI都在其下面，那么按前两种方式遍历计算所有的层级号，其中最大的那个作为自己的层级号。

遇到过的问题 在电脑上能运行，但是发布到手机上就不能运行非常非常的卡顿

在我的大学 一个同事没注意到这个问题 犯得低级错误没有把美术给的碎图打成大图
Batch: 用的是同一个材质球，同一个贴图 ，顶点才会被打包一次性传送给GPU 。
层级计算 总结:
1,有相同材质和纹理的UI元素　　如果不能Batch的UI元素叠在一块，就会增加Drawcall开销
2,尽量让同一个材质球上的东西 放在同一级上
3,有些情况可以考虑人为增加层级从而减少Drawcall，比如一个Text的层级为0，另一个可Batch的Text叠在一个图片A上，层级为1，那此时2个Text因为层级不同会安排2个Drawcall，但如果在第一个Text下放一个透明的图片（与图片A可Batch），那两个Text的层级就一致了，Drawcall就可以减少一个。
4,应该尽量避免使用Mask，其实Mask的功能有些时候可以变通实现，比如设计一个边框，让这个边框叠在最上面，底下的UI移动时，就会被这个边框遮住；　　Mask 模板测试
5, z 值　尽量保持一致　 默认都是0

DC: 计算发生了几次 传送指令
重建：
重新在传送一次。顶点的位置发生了变化　 就需要重新传送。
血条，布局组件 ，小地图 等 。
整个画布发生重建 。

针对重建 做优化：

a. 使用尽可能少的UI元素；在制作UI时，一定要仔细查检UI层级，删除不不必要的UI元素，这样可以减少深度排序的时间（上图中的2）以及Rebuild的时间（上图中的3，4）。

b. 减少Rebuild的频率，将动态UI元素（频繁改变例如顶点、alpha、坐标和大小等的元素）与静态UI元素分离出来，放到特定的Canvas中。
动静分离 。

c. 谨慎使用UI元素的enable与disable,因为它们会触发耗时较高的rebuild（图中的3、4），替代方案之一是enable和disableUI元素的canvasrender或者Canvas。

d. 谨慎使用Text的Best Fit选项，虽然这个选项可以动态的调整字体大小以适应UI布局而不会超框，但其代价是很高的，Unity会为用到的该元素所用到的所有字号生成图元保存在atlas里，不但增加额外的生成时间，还会使得字体对应的atlas变大。

e.谨慎使用Canvas的Pixel Perfect选项，该选项会使得ui元素在发生位置变化时，造成layout Rebuild。（比如ScrollRect滚动时，如果开启了Canvas的pixel Perfect，会使得Canvas.SendWillRenderCanvas消耗较高）

f. 使用缓存池来保存ScrollView中的Item,对于移出或移进View外的的元素，不要调用disable或enable,而是把它们放到缓存池里或从缓存池中取出复用。 无限滚动 。

有限的单元格子 展示 很多内容 。M 层的数据发生了变化 。

g. 除了rebuild过程之外，UGUI的touch处理消耗也可能会成为性能热点。因为UGUI在默认情况下会对所有可见的Graphic组件调用raycast。对于不需要接收touch事件的grahic，一定要禁用raycast。对于unity5以上的可以关闭graphic的Raycast Target而对于unity4.6，可以给不需要接收touch的UI元素加上canvasgroup组件。

unity5.x

unity4.6

重绘：
把界面打成overdraw
在同一个像素位置 不断的 填充 缓存区。

为了降低overdraw,可以做如下优化：

1. 禁用不可见的UI，比如当打开一个系统时如果完全挡住了另外一个系统，则可以将被遮挡住的系统禁用。

2. 不要使用空的Image,在Unity中，RayCast使用Graphi作为基本元素来检测touch,在笔者参与的项目中，很多同学使用空的image并将alpha设置为0来接收touch事件，这样会产生不必要的overdraw。通过如下类NoDrawingRayCast来接收事件可以避免不必要的overdraw。

```
3. public class NoDrawingRayCast : Graphic
4. {
5.         public override void SetMaterialDirty()
6.         {
7.         }
8.         public override void SetVerticesDirty()
```

```
9.        {
10.       }
11.       protected override void OnFillVBO(List<UIVertex> vbo)
12.       {
13.             vbo.Clear();
14.       }
}
```

常见问题：

Q1：我在UGUI里更改了Image的Color属性，那么Canvas是否会重建？我只想借用它的Color做Animation里的变化量。

如果修改的是Image组件上的Color属性，其原理是修改顶点色，因此是会引起网格的Rebuild的（即Canvas.BuildBatch操作，同时也会有Canvas.SendWillRenderCanvases的开销）。而通过修改顶点色来实现UI元素变色的好处在于，修改顶点色可以保证其材质不变，因此不会产生额外的Draw Call。

Q2：Unity自带的UI Shader处理颜色时，改_Color属性不会触发顶点重建吗?

在UI的默认Shader中存在一个Tint Color的变量，正常情况下，该值为常数(1,1,1)，且并不会被修改。如果是用脚本访问Image的Material，并修改其上的Tint Color属性时，对UI元素产生的网格信息并没有影响，因此就不会引起网格的Rebuild。但这样做因为修改了材质，所以会增加一个Draw Call。

Q：动静分离或者多Canvas带来性能提升的理论基础是什么呢？如果静态部分不变动，整个Canvas就不刷新了？

在UGUI中，网格的更新或重建（为了尽可能合并UI部分的DrawCall）是以Canvas为单位的，且只在其中的UI元素发生变动（位置、颜色等）时才会进行。因此，将动态UI元素与静态UI元素分离后，可以将动态UI元素的变化所引起的网格更新或重建所涉及到的范围变小，从而降低一定的开销。而静态UI元素所在的Canvas则不会出现网格更新和重建的开销。

Q：UWA建议“尽可能将静态UI元素和频繁变化的动态UI元素分开，存放于不同的Panel下。同时，对于不同频率的动态元素也建议存放于不同的Panel

**中。”那么请问，如果把特效放在Panel里面，需要把特效拆到动态的里面吗？**

通常特效是指粒子系统，而粒子系统的渲染和UI是独立的，仅能通过Render Order来改变两者的渲染顺序，而粒子系统的变化并不会引起UI部分的重建，因此特效的放置并没有特殊的要求。

**Q：多人同屏的时候，人物移动会使得头顶上的名字Mesh重组，从而导致较为严重的卡顿，请问一下是否有优化的办法？**

如果是用UGUI开发的，当头顶文字数量较多时，确实很容易引起性能问题，可以考虑从以下几点入手进行优化：

1. 尽可能避免使用UI/Effect，特别是Outline，会使得文本的Mesh增加4倍，导致UI重建开销明显增大；
2. 拆分Canvas，将屏幕中所有的头顶文字进行分组，放在不同的Canvas下，一方面可以降低更新的频率（如果分组中没有文字移动，该组就不会重建），另一方面可以减小重建时涉及到的Mesh大小（重建是以Canvas为单位进行的）；
3. 降低移动中的文字的更新频率，可以考虑在文字移动的距离超过一个阈值时才真正进行位移，从而可以从概率上降低Canvas更新的频率。

三、界面切换

**Q1：游戏中出现UI界面重叠，该怎么处理较好？比如当前有一个全屏显示的UI界面，点其中一个按钮会再起一个全屏界面，并把第一个UI界面盖住。我现在的做法是把被覆盖的界面 SetActive(False)，但发现后续 SetActive(True) 的时候会有 GC.Alloc 产生。这种情况下，希望既降低 Batches 又降低 GC Alloc 的话，有什么推荐的方案吗？**

可以尝试通过添加一个 Layer 如 OutUI， 且在 Camera 的 Culling Mask 中将其取消勾选（即不渲染该 Layer）。从而在 UI 界面切换时，直接通过修改 Canvas 的 Layer 来实现“隐藏”。但需要注意事件的屏蔽，禁用动态的 UI 元素等等。

这种做法的优点在于切换时基本没有开销，也不会产生多余的 Draw Call，但缺点在于“隐藏时”依然还会有一定的持续开销（通常不太大），而其对应的 Mesh 也会始终存在于内存中（通常也不太大）。

以上的方式可供参考，而性能影响依旧是需要视具体情况而定。

**Q2：通过移动位置来隐藏UI界面，会使得被隐藏的UIPanel继续执行更新（LateUpdate有持续开销），那么如果打开的界面比较多，CPU的持续开销是否就会超过一次SetActive所带来的开销？**

1，Setactive

2，CanvasGroup

3，移出屏幕

4，增加一个outlayer 层 相机不渲染层 。

这确实是需要注意的，通过移动的方式“隐藏”的UI界面只适用于几个切换频率最高的界面，另外，如果“隐藏”的界面持续开销较高，可以考虑只把一部分Disable，这个可能就需要具体看界面的复杂度了。一般来说在没有UI元素变化的情况下，持续的 Update 开销是不太明显的。

**Q3：如图，我们在UI打开或者移动到某处的时候经常会观测到CPU上的冲激，经过进一步观察发现是因为Instantiate产生了大量的GC。想请问下Instantiate是否应该产生GC呢？我们能否通过资源制作上的调整来避免这样的GC呢？如下图，因为一次性产生若干MB的GC在直观感受上还是很可观的。**

准确的说这些 GC Alloc 并不是由Instantiate 直接引起的，而是因为被实例化出来的组件会进行 OnEnable 操作，而在 OnEnable 操作中产生了 GC，比如以上图中的函数为例：

上图中的 Text.OnEnable 是在实例化一个 UI 界面时，UI 中的文本（即 Text 组件）进行了 OnEnable 操作，其中主要是初始化文本网格的信息（每个文字所在的网格顶点，UV，顶点色等等属性），而这些信息都是储存在数组中（即堆内存中），所以文本越多，堆内存开销越大。但这是不可避免的，只能尽量减少出现次数。

因此，我们不建议通过 Instantiate/Destroy 来处理切换频繁的 UI 界面，而是通过 SetActive(true/false)，甚至是直接移动 UI 的方式，以避免反复地造成堆内存开销。

切换界面 ： 后面用的上的界面 隐藏 。 用不上的才销毁。

界面采用的预加载：

四、加载相关

**Q1：UGUI的图集操作中我们有这么一个问题，加载完一张图集后，使用这个方式获取其中一张图的信息：assetBundle.Load (subFile, typeof (Sprite)) as Sprite; 这样会复制出一个新贴图（图集中的子图），不知道有什么办法可以不用复制新的子图，而是直接使用图集资源 。**

经过测试，这确实是 Unity 在 4.x 版本中的一个缺陷，理论上这张“新贴图（图集中的子图）”是不需要的，并不应该加载。 因此，我们建议通过以下方法来绕

过该问题：

在 assetBundle.Load (subFile, typeof (Sprite)) as Sprite; 之后，调用

Texture2D t = assetBundle.Load (subFile, typeof (Texture2D)) as Texture2D;

Resources.UnloadAsset(t);

从而卸载这部分多余的内存。

1，内存镜像 。图片，资源。AB 包

2，解压镜像。 图片　　解压后面要用的到的资源 马上将第一块内存释放。

3，场景中的物体

**Q2：加载UI预制的时候，如果把特效放到预制里，会导致加载非常耗时。怎么优化这个加载时间呢？**

**UI和特效放在一起了**

UI和特效（粒子系统）的加载开销在多数项目中都占据较高的CPU耗时。UI

界面的实例化和加载耗时主要由以下几个方面构成：

1. 纹理资源加载耗时

UI界面加载的主要耗时开销，因为在其资源加载过程中，时常伴有大量较大分辨率的Atlas纹理加载，我们在之前的Unity加载模块深度分析之纹理篇有详细讲解。对此，我们建议研发团队在美术质量允许的情况下，尽可能对UI纹理进行简化，从而加快UI界面的加载效率。

2.

采用中间的分辨率

320*480　　764*876　　1024*768

3. UI网格重建耗时

UI界面在实例化或Active时，往往会造成Canvas（UGUI）或Panel（NGUI）中UIDrawCall的变化，进而触发网格重建操作。当Canvas或Panel中网格量较大时，其重建开销也会随之较大。

4. UI相关构造函数和初始化操作开销

这部分是指UI底层类在实例化时的ctor开销，以及OnEnable和OnDisable的自身开销。

上述2和3主要为引擎或插件的自身逻辑开销，因此，我们应该尽可能避免或降低这两个操作的发生频率。我们的建议如下：

1. 在内存允许的情况下，对于UI界面进行缓存。尽可能减少UI界面相关资源的重复加载以及相关类的重复初始化；

B: 单独提取出来 。

Loading : A, Regist: ,C

2. 根据UI界面的使用频率，使用更为合适的切换方式。比如移进移出或使用Culling Layer来实现UI界面的切换效果等，从而降低UI界面的加载耗时，提升切换的流畅度。

1. 对于特效（特别是粒子特效）来说，我们暂时并没有发现将UI界面和特效耦合在一起，其加载耗时会大于二者分别加载的耗时总和。因此，我们仅从优化粒子系统加载效率的角度来回答这个问题。粒子系统的加载开销，就目前来看，主要和其本身组件的反序列化耗时和加载数量相关。对于反序列化耗时而言，这是Unity引擎负责粒子系统的自身加载开销，开发者可以控制的空间并不大。对于加载数量，则是开发者需要密切关注的，因为在我们目前看到的项目中，不少都存在大量的粒子系统加载，有些项目的数量甚至超过1000个，如下图所示。因此，建议研发团队密切关注自身项目中粒子系统的数量使用情况。一般来说，建议我们建议粒子系统使用数量的峰值控制在400以下。

**Q3：我有一个UI预设，它使用了一个图集， 我在打包的时候把图集和UI一起打成了AssetBundle。我在加载生成了GameObject后立刻卸载了AssetBundle对象， 但是当我后面再销毁GameObject的时候发现图集依然存在，这是什么情况呢？**

这是很可能出现的。unload(false)卸载AssetBundle并不会销毁其加载的资源 ，是必须调用 Resources.UnloadUnusedAssets才行。关于AssetBundle加载的详细解

释可以参考我们之前的文章：[你应该知道的AssetBundle管理机制。]

1，　内存镜像，图片
2，　镜像解压，图片
3，　实例化到场景　　Resources.UnloadUnusedAssets　　释放依赖内存

五、字体相关

**Q1：我在用Profiler真机查看iPhone App时，发现第一次打开某些UI时，Font.CacheFontForText占用时间超过2s，这块主要是由什么影响的?若iPhone5在这个接口消耗2s多，是不是问题很大？这个消耗和已经生成的RenderTexture的大小有关吗？**

Font.cacheFontSize()
Font.CacheFontForText主要是指生成动态字体Font Texture的开销,一次性打开UI界面中的文字越多，其开销越大。如果该项占用时间超过2s，那么确实是挺大的，这个消耗也与已经生成的Font Texture有关系。简单来说，它主要是看目前Font Texture中是否有地方可以容下接下来的文字，如果容不下才会进行一步扩大Font Texture，从而造成了性能开销。

常用的文字 只有 5000个　　总共文字有1万多　。

软件：
FontMaker　　FontFactory　　将不常用的文字 剔除掉 。

## Lua 遇到的坑：

1、不支持continue，因为没有这个关键字，有时候写代码有些麻烦
3、一个较大的数调用tostring()之后，再调用tonumber()，得到的值不是原来的值了。从代码可以看出只能容下14位小数位，所以有时候数字会被截断变小。

```
#define LUA_NUMBER_FMT "%.14g"
#define lua_number2str(s,n)　　sprintf((s), LUA_NUMBER_FMT, (n))
```

4、函数调用层次、参数个数等等经常会有一个最大数的限制，如果不小心超过了。。。。。

TextAsset

5、LUA的TABLE分为数组和HASH两个部分，很多的库函数是针对把TABLE当作严格的数组时起作用的。如果误用的话，结果通常是不准确的。　　len　　#　　for 循环 代替

6、LUA 5.1的变量默认是全局的，很容易在函数中的变量没有加上local而覆盖全局的变量，而导致全局数据变化，逻辑不正确。　　local　tmp

7、LUA代码执行如果出现错误会停止执行函数后面的代码，会导致需要严格控制的流程只执行了一部分，产生逻辑错误。例如在网游中购买商品的过程中，如果先给物品，再扣钱。代码在给完物品后出现错误，就不会执行扣除金钱的操作，导致可在无限地利用这个漏洞刷物品。 对于这类情况，一般是先检测金钱数是否够，条件是否满足，先扣钱，再给物品，并记录下日志。这样如果出现错误，玩家损失了金钱，但是可以找客服进行解决。