新特性

这篇文章主要介绍EUI库的新特性。在EUI库中，我们对这部分做了全面的优化：清除冗余计算，精简嵌套层次，减少类继承深度，并让包体最终降到了原先版本的40%。EUI目前的版本gzip后，包体仅有36k大小，对加载量已构不成压力。

以下是GUI体系时比较集中的几个痛点：

1.异步问题。失效验证机制带来的延迟计算优化，导致组件宽高无法立即得到，立即访问会返回0。

2.不统一的显示列表。两套显示列表使用不便，开发者需要额外判断addElement()和addChild()用在什么地方。

3.皮肤分离机制过于严格。皮肤分离机制在定义可复用的外观上更有优势。但对于只使用一次的外观定义不够方便。

新版本的EUI库在提高新手易用性上做了大刀阔斧的重构，以上痛点在EUI中也都得到了彻底解决，在以下的新特性解读中，将会逐一覆盖到。

组件立即返回宽高

我们在分析这个问题的过程中，得出了一条避免造成初学门槛的重要结论：子类不应该改变父类行为。

先简单介绍失效验证机制：就是属性发生改变时，不立即应用改变，而是先标记下来，延迟一定时间片后再统一计算，这样可以有效避免重复的计算量，从而提升运行时性能。尤其在使用自适应流式布局时，失效验证能起到强力的性能保障作用。而组件的宽高是需要动态测量的，所以也需要用失效验证机制延迟计算，因此实例化组件后，立即访问宽高将会返回0.

跟开发者深入沟通后，我们发现真实的需求都是：希望能把GUI对象当做普通显示对象来用,统一显示列表. 抽象出来质上几乎本都是相同的问题：子类改变了父类的行为

基于这个结论，我们重构了所有子类改变父类行为的设计。确保每个EUI组件都能直接当成普通显示对象来用。现在访问EUI组件宽高时，也会跟原生显示对象的表现一致，立即能得到包含子项的宽高值。

统一的显示列表

之前版本的GUI库引入了addElement()系列方法，用于替代addChild()系列方法，在GUI范围内调用addChild()方法将会报一个错。这么做的主要原因是自适应流式布局是层层向上测量，层层向下布局的。如果GUI显示列表中间混入了一个普通显示对象，将会造成自动布局体系断层而在那一层失效。所以想要正常启用自适应流式布局，就应该让GUI组件添加在一起，中间没有断层。但是这其实应该算一种最佳实践，不应该在框架层级去强制限制，开发者还是会有混合添加普通显示对象的需求的。之前是提供了UIAsset组件作为普通显示对象的包装器加到GUI显示列表，但这还不够方便。

基于以上得出的结论，两套显示列表并屏蔽addChild()系列方法的设计，显然是违反了「子类不能改变父类行为」原则的。现在EUI里只有addChild()系列方法，已经不存在addElement()方法。任意EUI组件和普通显示对象都可以互相混合添加。这里我们需要讨论的是两种添加情况的处理方式：

1.普通显示对象添加到EUI组件里。效果跟设置EUI组件的includeInLayout属性为false类似。布局类在计算布局时会主动忽略它，对它不测量也不布局，这样本身也比较符合预期。那么也就不在需要UIAsset包装器了，任何一个EUI组件只要有addChild()方法，都可以直接添加普通显示对象。

2.EUI组件添加到普通显示对象里。这个在之前的的GUI库里就是允许的。在这种情况下，EUI组件被当做一个普通显示对象来用，你对它设置的left，right等布局属性都无效，因为布局属性是需要父级容器对它布局的。而你的父级不是EUI组件。这个就是所谓的断层处。但是只会影响这一层，在组件内部再添加其他EUI组件，是可以正常布局的。

所以结论是，开发者一开始可以不使用自动布局功能，把所有EUI组件仅当做普通显示对象来操作。若需要开始用自动布局，可以再采用最佳实践的方式，将EUI组件都组织到一起。

EXML支持内部类

首先简单介绍皮肤分离的机制：皮肤分离机制就是将原本一个组件拆分成两个。一个逻辑组件只管代码控制，一个皮肤组件只负责外观。运行时将皮肤组件附加到逻辑组件上，变成一个完整组件。皮肤组件并不是显示对象，实际上更类似一个持有外观信息的数据对象。这样做的好处比较多，例如：方便代码解耦，方便复用外观，方便可视化编辑，等等。

而传统的UI方式，通常是只有一个组件，在组件上直接修改预设的外观属性。这个带来的问题是你必须在UI组件上声明非常多的 外观属性，例如文本颜色，背景色等，而且不管声明再多，通常也都还是会不够用，导致可自定义的部分比较有限。而皮肤分离的模式，逻辑组件上基本不声明任何外观属性。完全交给另一个皮肤组件去决定外观。这样可以完全自定义，扩展性上比较灵活。

之前的GUI库里必须将皮肤声明为一个独立的EXML文件，再引用它。对于需要复用的皮肤，这种方式比较理想。而对于只用一次的外观，则会比较不便，显得文件也特别多。现在EUI里已经支持EXML的内部类定义方式，可以直接嵌套写在节点内。通常有两种节点支持作为内部类：Skin和ItemRenderer，如下：

```
<?xml version='1.0' encoding='utf-8'?> <e:Skin class="skins.DemoSkin" xmlns:e="htt
p://ns.egret.com/eui" minHeight="230" minWidth="470"> <e:Button label="按钮"> <e:Ski
n states="up,over,down,disable"> <e:Image source="image/button_up.png" includeIn="u
p" width="100%" height="100%" /> <e:Image source="image/button_down.png" includeIn=
"down" width="100%" height="100%" /> <e:Label id="LabelDisplay" left="20" right="20"
 top="10" text="{data.label}" /> </e:Skin> </e:Button> <e:List id="list" cacheAsBitm
ap="false"> <e:itemRendererSkinName> <e:Skin states="up,down"> <e:Image source="imag
e/button_up.png" includeIn="up" width="100%" height="100%" /> <e:Image source="imag
e/button_down.png" includeIn="down" width="100%" height="100%" /> <e:Label id="Label
Display" left="20" right="20" top="10" text="{data.label}" /> </e:Skin> </e:itemRend
erSkinName> </e:List> </e:Skin>
```

如上面的例子，这个Button的外观只有它自己使用，那么可以从节点内部可以直接开始描述一个Skin，而不需要另外声明一个ButtonSkin的exml文件。另外一个比较常用的是ItemRenderer，ItemRenderer通常都是直接跟List关联的，很少有复用的情况，现在也可以直接嵌入写在List节点内部。

在代码上看，内部类的作用起来可能只是少创建了一个文件而已。但是在工具层面，这将会是完全不同的操作体验。之前要定义外观前，我们总是得先创建一个皮肤文件，编辑完后回来引用这个皮肤文件。现在的流程可以简化为：拖拽一个按钮到界面上，双击直接进入编辑外观。如果你不需要复用它，那么就结束了。当你需要复用这个按钮皮肤时，再一键将内部皮肤转换为独立EXML文件，变成可复用的。

运行时解析EXML

XML的文件结构描述显示列表有着天然的优势。在之前的GUI库里，EXML文件是在命令行编译阶段被编译为了JS文件，然后作为标准代码加载运行的。我们反复优化了好多次编译结果，始终还是EXML文件本身才是最小的记录方式。在EUI中，我们将EXML文件改为运行时解析，而不再提前编译。这样做的几个好处：

能够减少网络加载量。
减少中间转换过程，降低调试难度。也可以直接在编辑器编辑后拷贝EXML内容到代码中粘贴解析。
由于之前编译器只为TS开发者设计的，现在不再依赖命令行，JS开发者也能直接使用EXML文件。

不过不用担心性能问题，运行解析并不是每次实例化皮肤都解析一次，而是只有第一次解析，会将EXML编译为JS代码，然后使用eval()方法转换为标准的类定义。之后都直接调用类定义快速创建。

另外，之前EXML的模块名是根据所在文件夹路径生成的，现在由于EXML文件变成了运行时解析，有可能只有文本内容，并没有路径信息，因此包名也不再依赖文件路径。我们提供了另一种声明类名的方式：在EXML根节点上设置class属性，class属性的值会被解析并注册为全局类名。若不声明，这个EXML文件解析的类定义会被解析器作为一个临时变量返回。声明方式如下图：

```
<?xml version="1.0" encoding="utf-8" ?> <e:Skin class="skins.ButtonSkin" states="up,
down,disabled" minHeight="50" minWidth="100" xmlns:e="http://ns.egret.com/eui"> <e:I
mage width="100%" height="100%" scale9Grid="1,3,8,8" includeIn="up" source="button_u
p_png" /> <e:Image width="100%" height="100%" scale9Grid="1,3,8,8" includeIn="down"
source="button_down_png" /> <e:Image width="100%" height="100%" scale9Grid="1,3,8,8"
 includeIn="disabled" source="button_disable_png" /> <e:Label id="labelDisplay" top=
"10" bottom="10" left="20" right="20" verticalCenter="0" horizontalCenter="0"/> </e:
Skin>
```

EXML描述非皮肤对象

之前版本的GUI库里，EXML的根节点被限制为Skin皮肤节点，只能用于描述皮肤，实际上还有较多显示列表初始化需求，是直接使用Group等不可定义皮肤组件作为根容器初始化的。这样的情况就只能使用代码方式手动编写显示列表初始化代码，而不能使用EXML这种更加简便的描述方式。现在这部分代码，在EUI里也可以直接用EXML描述了。EXML的根节点不再必须是Skin，可以为任意组件。这个特性全面提升了EXML的适用范围，能够简化普通容器的显示列表创建过程。解析后的对象是一个继承自根节点的自定义类。定义了ID的节点，会在自定义类上以ID名声明一个成员变量持有该节点的引用。

动态数据绑定

数据绑定一直都是呼声相当高的一个便捷功能。现在EUI里也已经对其提供了支持。注意前文「EXML支持内部类」一节中的配图，可以发现ItemRendeerer内的Label节点已经使用了数据绑定功能：

text=”{data.label}”

它表示Label的text属性与ItemRenderer的data.label属性绑定。当列表的数据源改变时，ItemRenderer里的Label会自动刷新显示的文本内容。而不用手动写刷新的逻辑代码。在EXML中，开发者只需要简单地使用一对{expression}即可完成数据绑定，大括号内的expression表示根节点组件上的属性或当前EXML内定义的ID（实际上也是根节点上的属性）。注意在这个ItemRenderer的例子中，由于ItemRenderer是内部类，根节点就是ItemRenderer自身。

数据绑定功能相当于给静态的XML语法加入了部分动态刷新的功能，能够极大程度减少逻辑代码的编写量，在配合列表的ItemRenderer视图刷新尤其方便。之前我们通常要写一个ItemRenderer的逻辑类，覆盖dataChanged()方法，访问data属性，然后重新赋值刷新所有相关的视图组件。现在只需要简单地定义一个数据绑定标签，无需任何繁琐的过程。

另外值得一提的是，EUI里提供的数据绑定功能是基于setter的方式，改变的时候才会触发一次，并不是定时刷新检查的。得益于JavaScript的动态语言特性，所有的Object对象都可以实现动态数据绑定，并不限定于egret引擎内的对象。

自动布局兼容旋转缩放

这个特性也相当有用，之前版本的GUI库里，如果组件设置了旋转或缩放，自动布局的时候，依然是按照未变换之前的矩形来计算的，会造成诸多不便。现在EUI里对这部分也实现了完美兼容，不仅旋转缩放，对EUI组件使用Matrix进行的任意变换，都可以按照实际显示的矩形区域被正确测量和布局。