# UI解说

## UIComponent

最基本的UI组件，可使用原生的子对象API管理子对象，没有应用遮罩，框架以外的自定义显示对象可以添加到UIComponent中在添加到需要显示的组件中，这是由于框架已经屏蔽了组件容器的原生的子对象API；

UIComponent是不包含皮肤和状态的实现的；

### 事件

构造方法侦听Event.ADDED事件，当该对象被添加到任意显示对象后，才会播放UIEvent.INITIALIZE事件并调用createChildren方法；

当组件的3阶段延迟渲染都完成后，会播放UIEvent.CREATION\_COMPLETE和UIEvent.UPDATE\_COMPLETE两个方法，不同的是UIEvent.CREATION\_COMPLETE仅播放一次，而UIEvent.UPDATE\_COMPLETE则每次3阶段延迟渲染都完成后都会播放；

### 实现的功能

1. 框架的底层实现，systemManager和框架的初始化；
2. 延迟渲染；
3. 布局元素；
4. 工具提示；
5. 拖拽功能；

### 常用模版方法

下面这3个方法是具体组件的布局覆盖方法：

1. commitProperties：应用设置过的属性；按嵌套深度由外向内执行；
2. measure：测量当前原件的尺寸，如果手动设置过尺寸则该方法会被忽略；按嵌套深度由内向外执行；
3. updateDisplayList：应用设置的尺寸来更新显示列表；按嵌套深度由外向内执行；

### 尺寸详解（仅看宽度）

\_width：记录组件的宽度，显示修改width属性会改变该值，同时当组件被父层容器改变宽度时，一般是调用setLayoutBoundsSize方法后会设置新的宽度用该变量记录；

\_explicitWidth：记录组件显式指定的宽度，修改width属性会改变该值；

\_measuredWidth：通过measure方法测量出来的宽度，measure方法会在没有显示指定尺寸时被调用；

\_oldPreferWidth：辅助用；

layoutWidthExplicitlySet：记录宽度是否被容器布局对象更改过的标志；

\_percentWidth：百分比宽度，相对于父层，和top，bottom一样，由父层来确定其实际宽度；

\_minWidth：测量组件尺寸时会固定测量出的值在设定的范围内；

\_maxWidth：同上；

setActualSize：方法，直接设置\_width和\_height的值，多用于父层设置子对象的尺寸时；

scaleX：和width属性解耦，设置后会缩放组件，同时父层会重新布局；

preferredWidth：按照: 外部显式设置宽度 -> 测量宽度的优先级顺序返回宽度. 包含scale的数值；

layoutBoundsWidth：按照: 布局宽度 -> 外部显式设置宽度 -> 测量宽度的优先级顺序返回宽度. 包含scale的数值；

setLayoutBoundsSize：方法，父层设置子组件尺寸的方法；

组件尺寸按照 布局设定的尺寸 -> 显示定义的尺寸 -> 测量的尺寸 来设定；

## Group

最基本的容器组件，屏蔽了组件容器的原生的子对象API，实现了一套用于组件的子对象管理API，而UI组件除了UIComponent外也都屏蔽了原生的子对象API，这样的原因是：Flash API里没有提供一个既有鼠标事件又不是容器的显示对象基类。所以框架里所有组件都继承自Sprite。也就是说所有组件不管是不是容器，都含有addChild()方法。这样对组件体系会造成混乱，你可以随意给非容器添加子项，结果就是添加的子项都无法自动布局，调试起来也会更加困难。而要从代码层面避免这种错误使用，就需要引入一套新的容器接口。只有是容器的组件才具有它们。另外一个原因是，addElement()虽然底层还是调用的addChild(),但是它的参数不是显示对象，而是接口，更具有扩展性。这样你就可以把非显示对象的也当做子项添加到显示列表。Flex里主要是为了兼容绘图元素，这里主要是为了将来精简显示列表嵌套层级做准备(皮肤可以是非显示对象，这样就不存在多一层嵌套了)。扯了这么多,最后记住一句话：在框架范围内，只要你使用了addChild()等方法，不要想了，肯定是写错了！

容器类使用 group 来作为内部容器而不使用 Sprite，主要原因是利用了 group 是继承 UIComponent 的特性，拥有延迟渲染的功能，不会造成组件断层，否则 mesure 向上执行会被中断；

### 实现的功能

1. 布局逻辑，遮罩位于布局类中；
2. 框架的子组件管理；
3. 透明背景绘制；

## SystemManager

一个应用程序应只有一个SystemManager，HammercGlobals的\_systemManagers列表是用来记录通过外部加载到本的程序的SystemManager的，Air的多窗口不在我们的讨论范围内，其有自身的另一套实现方法。

### 两种方式添加SystemManager

1. 直接把SystemManager作为文档类，然后实例化一个UIComponent添加到SystemManager里，再把游戏场景addChild进UIComponent。
2. 不改变原有的文档类结构。实例化SystemManager然后把它addChild到游戏场景的上面盖住场景即可。UI都在SystemManager里开发。游戏仍然在传统显示列表里开发。

### 实现的功能

1. 虚拟容器管理，用来实现容器分层；
2. 鼠标事件过滤这点是为了弥补FP原生事件的缺陷，FP抛出的所有的鼠标事件对象默认都是不可取消的，也就是在构造函数里传入了cancelable=false。而在框架内有很多组件是需要调用event.preventDefault()来实现阻止某个操作发生的功能。所以在根容器的鼠标事件捕获阶段加了层过滤，把相关的鼠标事件对象转换为可以取消的再重新抛出。
3. 自动跟随舞台改变大小这个非常好理解。FP原生只不带自动布局功能的。所以这里让桥接用的SystemManager跟随舞台大小而改变，从而通知相关子项层层重新布局。你只需把SystemManager实例化好添加到显示列表。它就会自动监听舞台事件，然后始终保持自己的尺寸跟舞台完全一致。此时SystemManager的x，y，width，height设置都是无效的。当然，也有一些特殊的情况下，你不想跟随舞台。比如要做的游戏是固定场景尺寸的。这时你可以将SystemManager.autoResize设置为false。即可关闭这个功能。

### 去掉的功能

1. 自加载的perloader；

# 皮肤实现

皮肤和组件分离：分为逻辑组件和皮肤组件，逻辑组件是皮肤组件的宿主；

## 实现原理

一个组件的具体绘制和子组件的创建都是在皮肤组件中完成的，逻辑组件和皮肤组件都有同样名称的公共属性，逻辑组件能拿到皮肤组件里同名属性的引用；

逻辑组件和皮肤组件的同名公共属性都必须是组件内部子组件的名称，皮肤组件负责实例化这些组件，在添加到逻辑组件后逻辑组件的同名属性会持有这些组件的引用；

### 子件规则

1. 使用公共属性而非公共setter/getter的寄存器属性；由于组件的设计中所有公开属性都必须是setter/getter方式定义的，所以公共属性可以很方便的拿到；
2. 过滤掉基本数据类型；
3. 该属性必须是逻辑组件和皮肤组件都存在的属性；

## 逻辑组件相关方法：

partAdded：如果添加的皮肤是ISkin接口，一个皮肤子件注入后会调用该方法；

partRemoved：如果添加的皮肤是ISkin接口，一个皮肤子件移除引用前会调用该方法；

createSkinParts：如果皮肤为空或为非ISkin接口，则会调用该方法添加所有皮肤子件；

removeSkinParts：如果皮肤为空或为非ISkin接口，则会调用该方法移除所有createSkinParts方法添加的皮肤子件；

## 主要皮肤相关类解说

### UIComponent

可使用原生的子对象管理API，是组件基类，没有皮肤属性；在UI中添加自定义的任意显示对象都使用它；

### UIAsset

屏蔽了原生的子对象管理API的非容器组件（即不能任意添加子对象）和内部对象的鼠标交互事件，添加了皮肤属性，可以将解析后的皮肤对象添加到自身；可以通过设置皮肤名称来显示位图和影片剪辑；

### DefaultSkinAdapter

默认的皮肤名称解析器，支持处理的皮肤名称有：Class（实例化后返回）、String和ByteArray（异步加载后返回）、BitmapData（用Bitmap包装后返回）、其它类型直接返回；

### SkinnableComponent

支持ISkin类型皮肤的逻辑实现，同名公开属性的实例注入逻辑；

实现组件状态的逻辑；

### SkinBase

实现了ISkin接口和IStateClient接口的皮肤类，继承自Group组件，所有标准的皮肤类都应该继承自该类；

### SkinLayout

针对非ISkin接口的皮肤类的布局类；

### SkinnableContainer

继承自SkinnableComponent实现IUIContainer接口，内部添加了一个Group容器，所有实际添加的组件都添加到该Group对象中；

### DataGroup

可以设置项呈示器后，传人列表对象呈现多个同样项目的容器；

### SkinnableDataContainer

可以添加皮肤对象的DataGroup；

### ViewStack

可添加多个子件，但始终只显示其中一个子件的容器，无皮肤类；

# 状态实现

IStateClient接口应用在皮肤类上，通常在会皮肤类中的构造函数中指定所有的状态，同时对应的逻辑组件的方法getCurrentSkinState()会返回当前的状态字符串，和皮肤类中的状态一一对应；逻辑组件的validateSkinState()方法会把逻辑组件的当前状态赋值到皮肤类的currentState属性上，最后皮肤类的updateDisplayList方法中会取得currentState属性来绘制组件。

状态改变时要在逻辑组件上进行操作则需要覆写validateSkinState方法。

## 关于enabled这个属性

一般来说enabled设置为false后应该是无法和鼠标事件交互和加上黑白滤镜，但是有部分情况除外，比如一个按钮禁用的时候还是希望鼠标移上去能显示工具提示则不能禁用鼠标事件。

UIComponent中的enabled属性仅仅是记录该属性的值和抛出”enabledChanged”事件，没有具体的处理逻辑；

Group等非带有皮肤组件由于没有进行支持所以设置enabled无效，需要禁用则要直接设置mouseEnabled属性；或者可以使用SkinnableContainer类作为容器，则enabled属性有效。

SkinnableComponent类中处理了enabled属性，同时添加属性autoMouseEnabled确定enabled为false时是否接受鼠标事件，ui类库中仅ButtonBase分支该属性为true，即enabled为false时不禁用鼠标交互；

SkinnableComponent中设置enabled为false时如果存在状态则由皮肤类处理表现，如果没有状态则会添加黑白滤镜到本组件上；

# 一些未实现的组件

## Image

如果仅仅是加载外部图片并显示，直接使用UIAsset即可，有特殊需求可以自行创建该类。

## Menu

游戏中常用的菜单为弹出菜单且只有1级，使用list封装一下即可。

# 样式

按照皮肤分离的逻辑来看，逻辑组件真正控制皮肤的属性仅仅存在一个（label等文本组件有更多属性），就是skinName属性，所有有关布局和样式的设置都应该存在于皮肤组件中；

这样导致的问题（其实不能称为问题，这是皮肤分离一开始就设定好的规则）或不方便是：当我需要扩展组件时会比较麻烦，比如当我的按钮需要显示的是图形文本时，必须扩展ButtonBase类和创建对应的逻辑类（否则只能一个特定的文本图片创建一个皮肤类，这样创建有n个逻辑几乎一致的皮肤类）；或者我的下拉框的长度每个都一样，也需要扩展出逻辑类和皮肤类来完成。

所以这里引入了Flex3的样式设置方法，即逻辑类添加一个setStyle的方法来直接控制皮肤类的任意样式；这样对处理皮肤样式有了更多的选择；

## 详解

样式仅控制应用了ISkin的皮肤类，所以该功能是SkinnableComponent中加入而不是UIComponent中加入的，如果添加的皮肤不是ISkin接口则无效；

**StyleManager.globalStyleDeclaration**

全局的样式，所有的组件样式都会继承自该样式；

**styleName**

可以注册一个样式，每个组件的styleName属性赋值为该注册的样式的话会继承该注册的样式对象的所有样式；

每个组件都会有一个默认的样式名称，为其类名；

注意：每个组件的styleName属性应该在创建该组件后就进行设置，下一帧后即会创建组件样式描述对象，再修改该属性无效；

**SkinBase.styleProperties**

请在皮肤类的构造函数中指定其接受的所有样式属性名称，在逻辑组件中被设置或为设置都会回调皮肤类的styleProperties方法；