# 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；

### 去掉的功能

1. 取消虚拟布局；

## 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

皮肤机制是基于失效验证机制的；