

LeUI consult book

LeUI 框架思想与使用示例-v1.1

目录

一、概述.....	3
二、框架设计思想.....	4
基本设计思想.....	4
LeUI 核心接口设计：	5
装饰机制.....	6
资源库管理.....	9
三、特点与优化.....	9
全局共享事件机制.....	9
异步渲染机制.....	9
性能上的一些优化.....	11
四、使用 LeUI 组件.....	12
配置样式库.....	12
使用 LeUI 组件.....	19
扩展 LeUI.....	40
五、 UI 编辑器的使用.....	45
新建 UI.....	45
添加新组件.....	48
删除组件.....	50
保存 UI.....	51
打开 UI.....	52
导出文件.....	53
更换资源库.....	55
六、小结.....	56

一、概述

Actionscript3.0 目前仍是主流的网页游戏开发语言。对网页游戏的前端开发者来说，通常有近一半的时候是编写 UI 界面，UI 是如此耗时，以至于一款好的 UI 框架变得相当重要。

Adobe 官方的 flex 框架带有设计编辑工具，组件也很齐全，但是使用 mxml 编译出的文件略显臃肿，因此，相比游戏而言，flex 还是更加适合于做商务 web 项目；而且也有一部分开发者（包括本人在内）更钟情于使用纯 as3 做开发。

另一个比较知名的 UI 库 Aswing，脱胎于同样大名鼎鼎的 swing（java）框架，整个框架设计水平令人佩服，可以用来作为学习设计模式的良好参考，而且使用纯 as3 编写，深得广大 aser 的喜爱，本人曾经参与过两个项目都是使用的 Aswing 做的 UI。但它是一种面向多种使用目标情景而设计的，也就是说你可以用它来做 web App，也可以用来做游戏，这样的好处是显而易见的，用途广泛；不过同时也意味着有相当一部分功能是游戏中用不到的。

此外，许多人用过 ghostCat 库，里面包括一套 UI 框架（本人当前参与的游戏项目就在使用），同样非常棒，简化了一些常见的设计模式，比如数据和组件不再刻意解耦，使用起来挺方便的。通过反射技术，链接资源与实例，将资源打包成某种格式(比如 swf)，动态加载反射链接为 UI 类，十分有利于模块化的游戏实现。

除了上述几个较为知名的 UI 库，不少公司都是自己开发自己使用，也陆续有一些新的 UI 库开源，这里就不多说了。

仅就本人使用过的这几个 UI 库而言，虽然设计优秀，各有千秋，但并没有一款框架是针对游戏开发的特点而架构，因而不能方便快捷地适用于游戏开发。游戏 UI 开发具有以下特点（个人看法）：

- 1.游戏开发主要使用资源图拼接界面，而不是使用 as 的绘图 api。
- 2.游戏开发的资源图经常需要放入资源库中复用。
- 3.游戏开发需要频繁的版本升级，因此需要解决资源库的缓存问题。
- 4.游戏开发需要好用的 UI 编辑器来提高开发效率。
- 5.游戏开发需要 UI 组件与事件管理、log 系统对接。

LeUI 就是针对游戏开发的这些特点而设计的！具有以下特点：

- 基于资源图来装饰组件；
- 简化了样式设置方法；
- 多种方式管理资源库；
- 支持全局共享事件机制；
- 异步渲染机制；
- 性能上的许多优化；
- 为将来配套的 UI 编辑器预留了便利的接口；
- 开放源代码，方便使用者根据实际情况修改和进一步优化。

二、框架设计思想

基本设计思想

LeUI 的基本设计思想是聚合/组合，其实绝大多数程序设计语言的数据结构设计中也直接体现聚合/组合的思想。比如程序语言一般会提供元数据类型，复杂的数据类型则使用元数据类型的聚合/组合实现。而现实中，最能体现这一思想的物品我认为是积木玩具，因此以下以积木玩具来类比说明本 UI 库。

积木玩具特点：基本元素块带有色彩可以作为独立的对象存在；基本元素块通过不同形式的组合可以形成不同的图案；通过不同层级的聚合，可以形成复杂的形状。

类比于 UI：简单组件可以由资源图像独立装饰；复杂组件亦可由简单组件通过组合或聚合构成。许多 UI 库也多少表现出了这种思想。而 LeUI 将这种思想作为架构的根本。

一块元素，可以是三角形，可以是四边形.....；可以是红色，可以是蓝色.....。在 LeUI 设计思想中，称之为可装饰元素(IStyle)：



图 2-1 可装饰元素

多块元素，可以通过组合的模式，来构成复杂的组件：

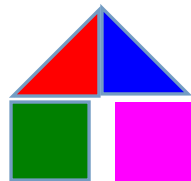
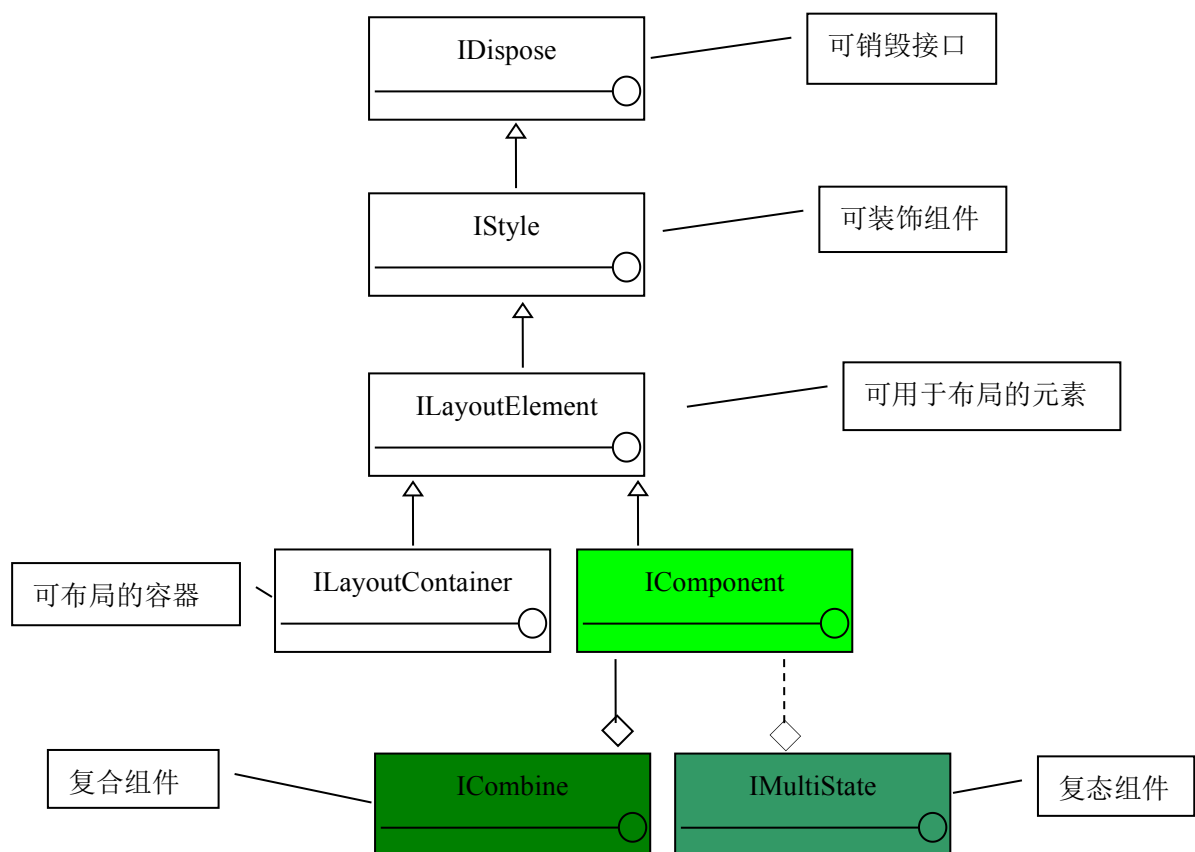


图 2-2 由可装饰元素构成复杂组件

本套 UI 库命名 LeUI，以此来向著名的积木玩具品牌 LEGO 所承载的组合/聚合思想致敬。LeUI 中的各级组件类名均以“L”开头，如组件基类 LComponent。

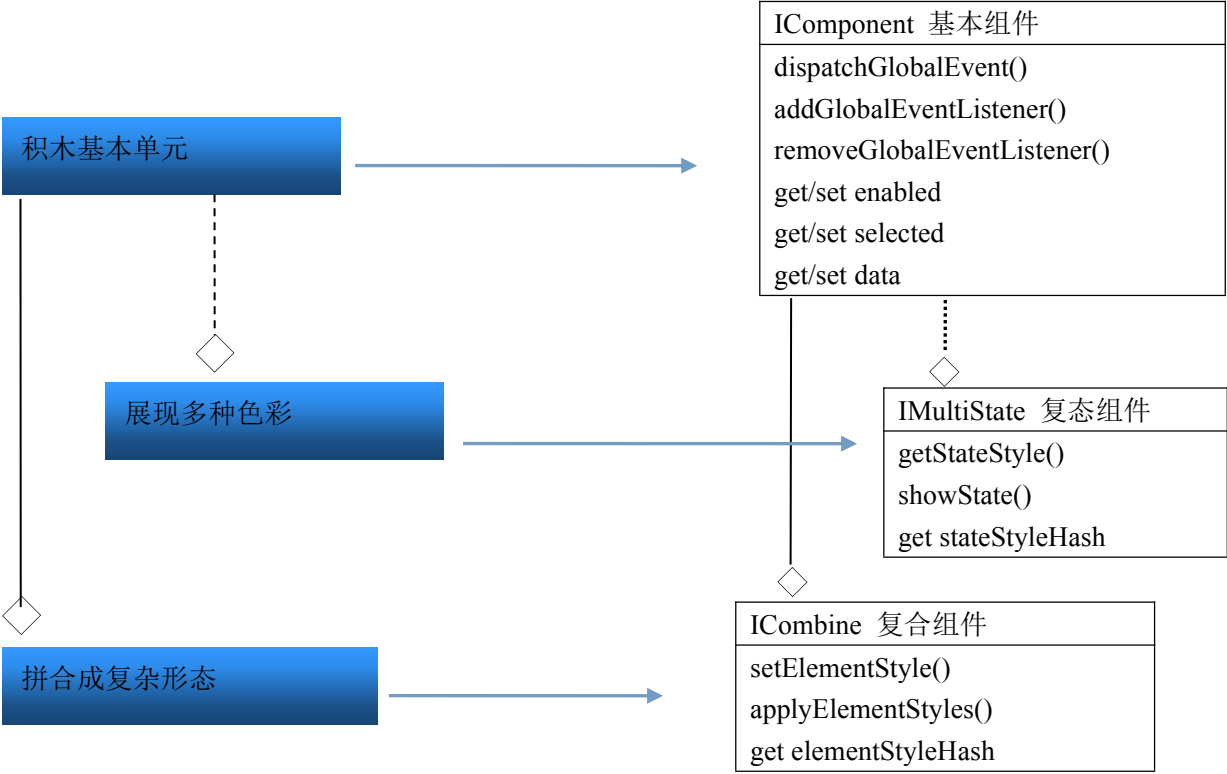


LeUI 核心接口设计：



上图显示了 LeUI 核心接口设计，图中带底色的三个接口：**IComponent**、**IMultiState**、**ICombine** 是 LeUI 设计思想的直接体现。

装饰思想与 LeUI 设计思想的映射关系如下图：



装饰机制

上图中，显示了设计思想中积木特征与 LeUI 中接口继承体系相对应的关系。而实际上，IComponent 承载了更多的功能性设定，而非仅仅是对应于“积木基本单元”。真正对应于“积木基本单元”的接口设计，是 IComponet 的上层接口 IStyle。

IStyle 定义了“可装饰元素”：

IStyle 可装饰元素
get/set style
getDefaultStyle()
resetStyle()
setBg()

前文提到过，游戏内的 UI 绝大多数是基本位图的，因此在 LeUI 的“可装饰元素”接口定义时，所指的“装饰”，意思即使用显示对象作为组件的背景，平铺至组件设定的尺寸(使用九宫格缩放)。由方法 `setBg(asset:DisplayObject)`完成这一工作。

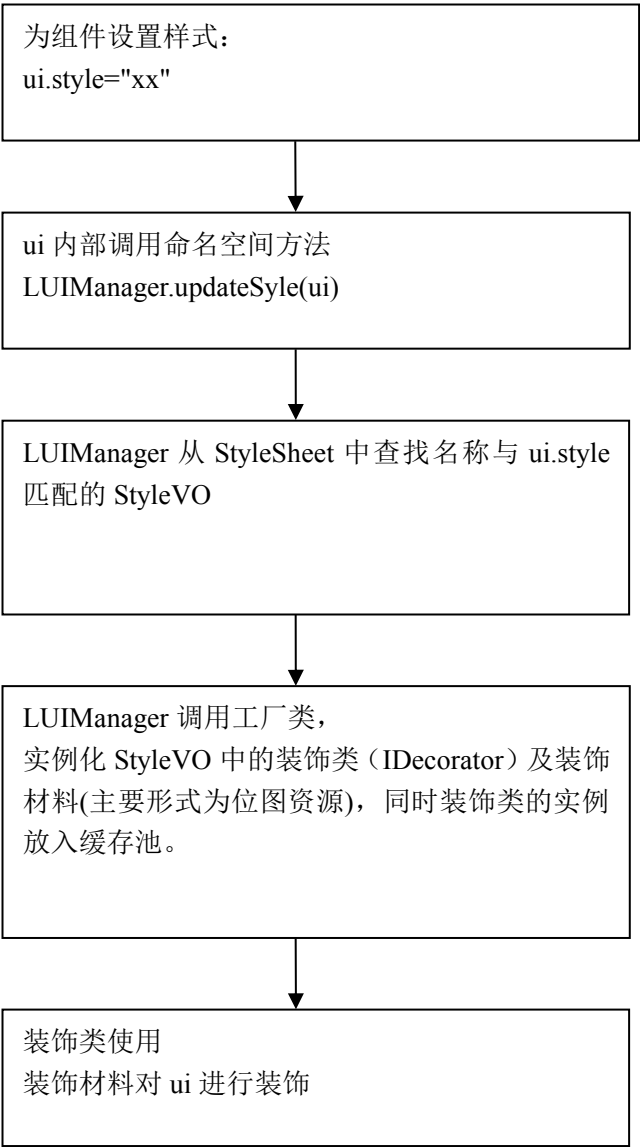
LeUI 中定义了一个资源库接口 `IStyleSheet`，及样式信息包装类 `StyleVO`。LeUI 中为 `IStyleSheet` 提供了一个实现类 `LStyleSheet`，可称谓样式表。`IStyleSheet` 定义如下：

IStyleSheet 样式表
<code>putStyleVO()</code>
<code>getStyleVO()</code>

`StyleVO` 被设计成动态类，原因是为了复态组件及复合组件的样式信息在包装上能够统一，也是为以后的 UI 编辑器处理起来方便。样式包装类 `StyleVO` 定义如下：

StyleVO 样式信息
<code>styleName</code>
<code>decoratorClass</code>
<code>assetClass</code>

一个可装饰组件（`IStyle`）的装饰流程如下：

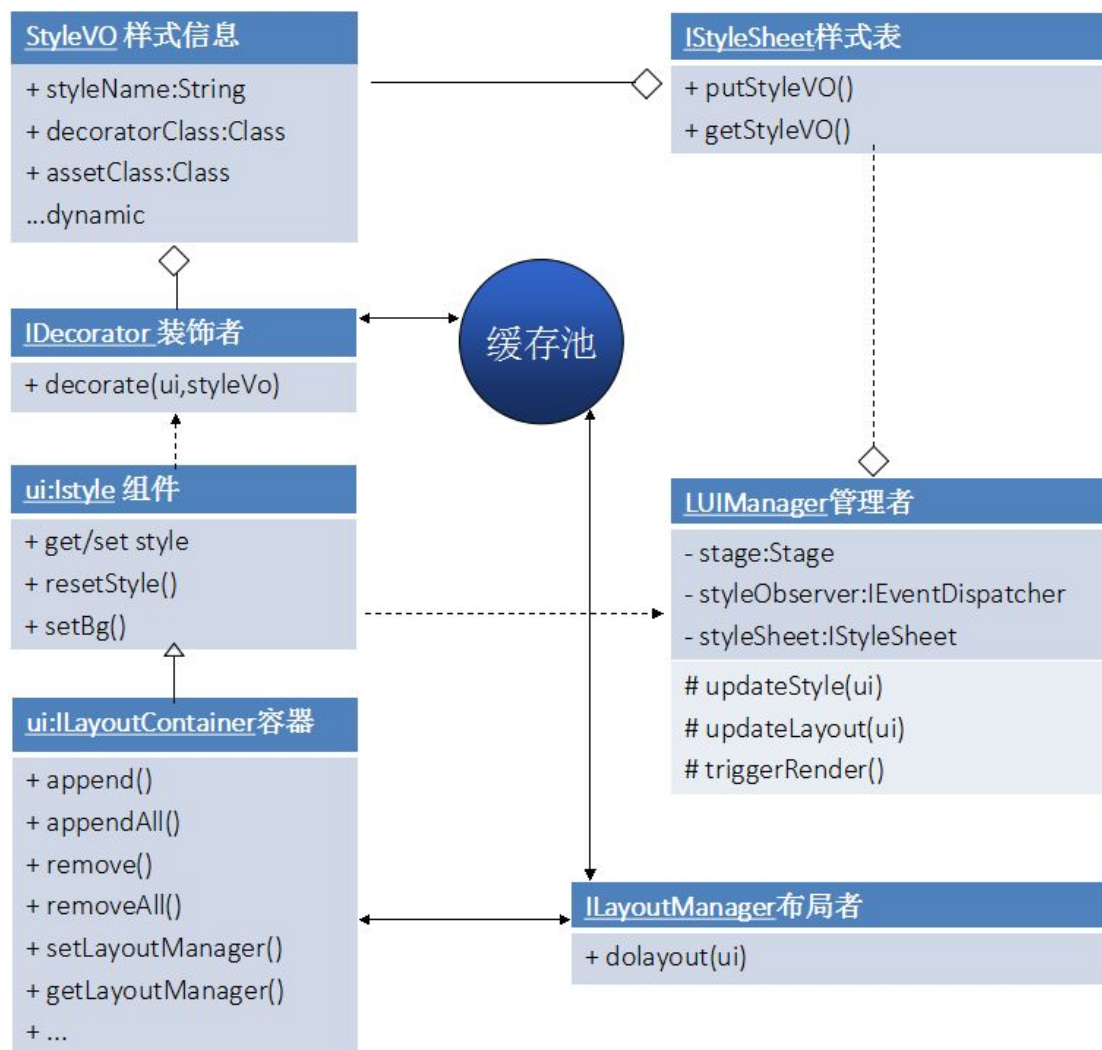


装饰过程由 LUIManager 控制，调动 IStyleSheet、StyleVO、IDecorator 及位图资源，完成对 IStyle 组件的装饰。

在 LeUI 核心接口设计的图示中，可以看到，IComponent 是组件基类 LComponent 的直系接口，而从 IStyle 到 IComponent 之间还有一个 ILayoutElement，称之为可布局元素，它有两个子级接口：IComponent 和 ILayoutContainer，因此，LeUI 中的所有组件都是可布局元素，包括可布局容器 ILayoutContainer（下称容器）。

容器对显示列表中子对象的位置、尺寸的管理称为“布局”。与可装饰组件的装饰流程类似，容器的布局管理也借助 LUIManager，调动 ILayoutManager 实例（从缓存池中存取）对容器的子显示对象设置尺寸和位置。所以在设计理念上，布局与装饰是一脉相承的，可将之统一划归到 LeUI 装饰机制，具体可以参考下图，阅读源码。

LeUI 装饰机制：



LeUI 从开始发起到现在，框架设计上经过多次推倒重构（当然最初连名字都不是现在这个），设计时主要参考了 Aswing 和 ghostcat 的 UI 框架，因此熟悉这两个 UI 库的 aser 如果使用 LeUI 应该会感到亲切。目前 LeUI 的框架设计能够实现快速方便的扩展自己的复合

组件，且只需在样式表中利用已有的子元素样式快速配置出新组件的样式，这对于游戏开发来说能较大程度的提高效率。后面的组件使用例子中会详细介绍。

资源库管理

在 LeUI 的规划中，LeUI 编辑器包括两个部分：**库编辑器**和 **UI 编辑器**，资源库的管理通过**库编辑器**来实现，**库编辑器**提供切图、九宫格等资源库工具，将图片（通常是 Png 和 jpg）资源编辑后存入一个列表，然后生成资源库文件（格式为：swc 或 swf 或自定义的二进制文件），并将资源样式名映射到对应的样式表文件（xxStyleSheet.as）。目前，**库编辑器尚未开发**，暂时先提供了一个 **python 脚本**，将资源图片映射成嵌入类文件（后面会提到）。

借助 LeUI 设计时的另一特点：全局共享事件机制，可以在程序运行时，更换一套全新的资源库。例如一个游戏中，主城是一种风格的 UI，进入某个大的副本时，可以动态的设置一个新的 UI 资源库给 LeUI，则所有的 LeUI 组件会自动使用新的资源库更新样式。

三、特点与优化

全局共享事件机制

全局共享事件机制，顾名思义，就是提供一个全局共享的事件派发器实例，以方便不同组件之间的通信。页游客户端框架里通常会使用 MVC 模式，通过 MVC 来完成各模块之间的通信解藕，但每一种设计模式都非完美，只是在使用时的相对最优选择而已。LeUI 设计全局共享事件机制，是为了对游戏内各模块通信提供一种便利的补充。IComponent 接口中设计了三个方法来支持这一机制：

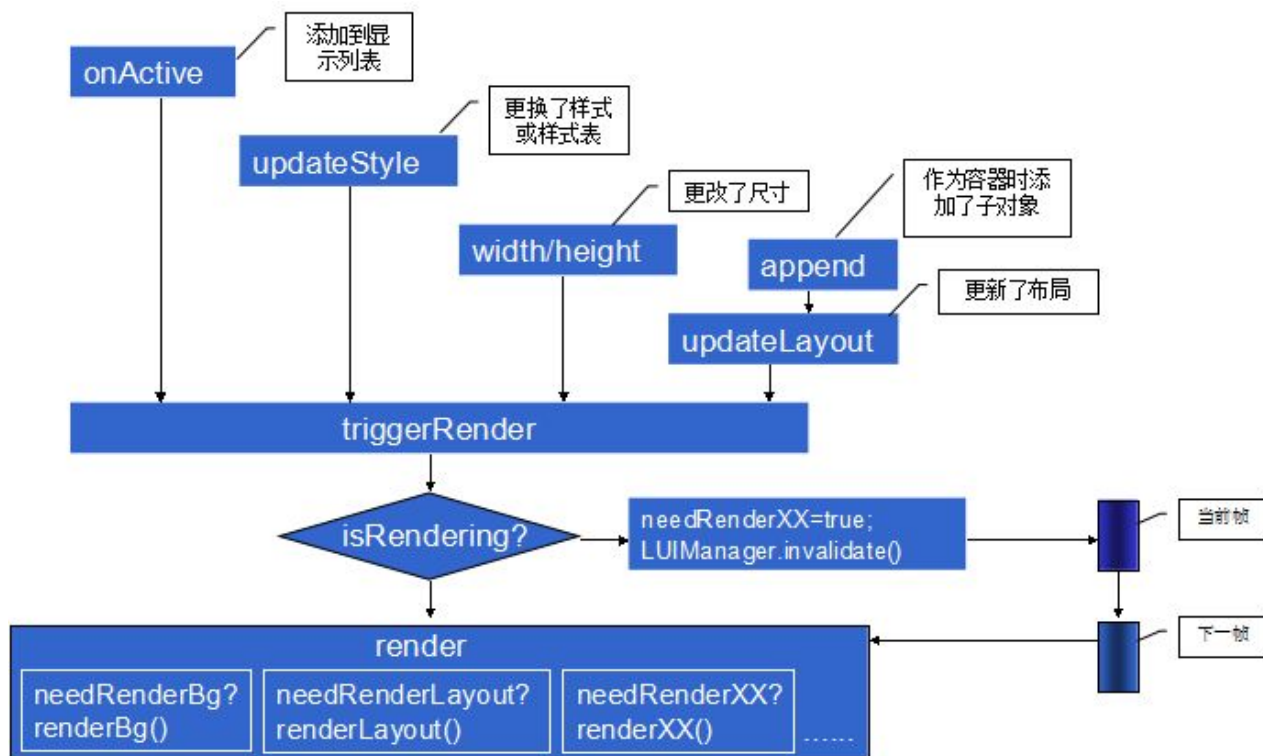
```
dispatchGlobalEvent() //派发全局事件
addGlobalEventListener() //监听全局事件
removeGlobalEventListener() //移除全局事件的监听
```

异步渲染机制

所谓的异步渲染，是早期 flashplayer 版本时流行起来的一个概念，认为每帧执行时，对显示对象的渲染默认是即时的，优化方法是在每帧的后半部分有一个固定的渲染阶段，把渲染延迟到这个固定的渲染阶段来做，可以降低消耗，提升性能。与之相关的，有人提出了帧执行时的“弹性跑道”理论（此处不再详述，具体可以百度），这一理论没有官方的正式认可，但在 adobe 编写的《flash 平台优化》这本小册子里描述的也差不多。实际操作方法一般是基于 Event.RENDER 事件，将对显示对象属性的改变异步到这个事件的监听函数里去执行。LeUI 初始版本也依此设计了一套所谓的异步渲染机制（该设计在当前版本已被干掉，

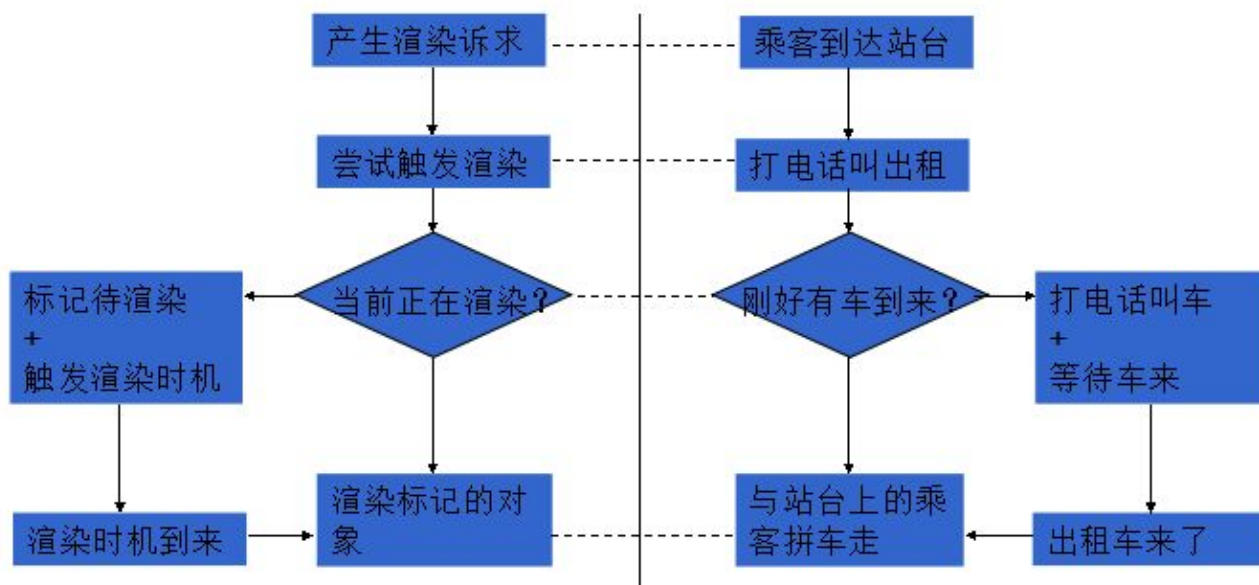
至于原因，后文会提及)。LeUI 在每帧初始时将一个全局变量 isRendering 设置为 false,即将进入渲染阶段时将 isRendering 设置为 true.....

在该机制下，对组件的某些属性或状态的修改，会记录一个待渲染状态，然后在触发 Event.RENDER 事件时，再真正的应用对属性的修改，原理如下图：



上述设计可以用一个打车的类比来说明：

LeUI异步渲染机制——类比说明



图中将异步渲染类比为在站台打车。此类比中，乘客去往同一目的地，因此拼车更节省资源。

那么为何在正式版中将这一机制干掉呢，因为实际测试发现，这种所谓的异步渲染，并不能达到提高性能、节约开销的目的。原因简述如下：

早期版本的 flash player 的渲染机制如何已不好做太多考证，但在早期的 flash player 中，异步渲染机制确实存在于很多框架中，包括 adobe 官方的 flex 框架，以及 aswing。而在近期以来的版本中（flash player 10.x 以上版本），每一帧里，对显示对象的渲染分为两种：交互对象和非交互对象，典型如 Sprite 和 Shape，根据本人的测试结果，交互对象触发鼠标事件时，渲染是即时的，而非交互显示对象的渲染则只在每帧的最后阶段执行一次。

因此，在目前的测试结论下，异步渲染已不能达到提升性能的目的，因为对于交互对象，交互时始终执行即时渲染，同时对于非交互对象，如今的 flash player 已经自动对其应用了异步渲染机制。如果 LeUI 仍然统一对所有显示对象通过监听 Event.RENDER 事件来实现所谓的异步渲染，则当一个 Sprite 触发了渲染进而执行 Event.RENDER 事件监听函数时，原来不需要即时渲染的 Shape 也被强制进行了即时渲染，反而比 adobe 目前默认的机制更加浪费资源，有鉴于此，LeUI 果断放弃了上图上所设计的渲染机制。

性能上的一些优化

LeUI 针对性能的优化，包括以下几个方面：

1.数据的及时销毁：LeUI 组件的最顶层接口为 IDispose，接口唯一的方法 dispose()。该方法需要用户自行调用，此方法的作用是：移除此组件的所有子显示对象、删除组件附带的数据、移除组件内所有的事件监听，如果此组件是容器，则会对子组件递归执行 dispose()。

2.为组件设置激活开关：UI 组件的激活与否是根据其是否在显示列表中判定的，当未被添加到显示列表时，处于未激活状态，此时会移除事件监听函数，且对组件的样式设置及布局设置只记录所设置的信息，而并不实施变更，这样就能减少 flash player 事件流中无谓的遍历，同时减少无用的渲染检测；当组件被添加到显示列表时即变成激活状态，事件监听被重新添加，同时样式、布局进行一次更新。

3.为常用事件添加统一监听接口：flash player 的事件机制中，事件触发后，通过显示列表遍历过程中，会对每个节点检测其是否有监听函数，如果有，则执行监听函数。LeUIManager 为较为常用的 ENTER_FRAME 事件和 KEY_DOWN 事件提供了统一的监听接口，方便用户将这类事件的监听集中处理，以减少事件流对显示列表遍历执行的时间。

4.复合组件等屏蔽自动组装子元素的样式：LeUI 组件激活时，会自动进行一次样式、布局的组装（根据已设置的样式/布局 或 默认的样式/布局），而在复合组件中，由于激活是根据是否被添加到显示列表判定的，Event.ADDED_TO_STAGE 会被对所有实例执行监听，这就导致子组件自动组装一次，复合组件本身进行样式组装时，会对子组件又进行一次样式组装，如果复合组件的子组件也是复合组件，则会嵌套调用多次组装操作，性能浪费严重，所以目前先屏蔽子元素默认样式组装。所以对于复合组件，需要在样式表中对其子元素的样式

进行显式配置，否则你可能看不到正确的结果；类似的列表类容器如 LList、LGrid 等也屏蔽子元素的默认组装，需要在创建子元素时为它设置样式，或先将子元素添加到显示列表以激活其样式组装，再将它添加到容器中。 PS:此机制以后可能会用更好的方式优化。

四、使用 LeUI 组件

LeUI 组件可以通过手动调用 `setBg()` 方法装饰其背景，因此你可以在任何时候、任何途径下给它设置“样式”。

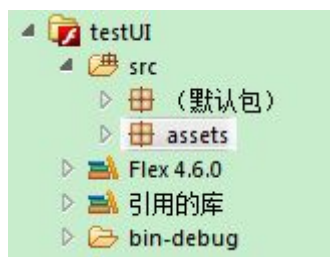
但是 LeUI 的目标用户是游戏开发者，游戏 UI 在设计时，通常会有一个统一的视觉规范，这个统一的规范落实到资源层面就可称之为资源库，资源库是根据一定的 UI 规则、从设计效果图中分离出来的一系列图片资源，程序里的通用组件使用 UI 规则+图片资源来还原统一的设计效果。这种机制与 LeUI 中相对应的就是样式表 `LStyleSheet` 与样式信息 `StyleVO`。本文档下面将讲解如何使用 LeUI 设定的标准方式来配置 `LStyleSheet` 与 `StyleVO`。

配置样式库

首先介绍如何手动配置资源库；等 LeUI-Builder 完工后，再补充使用编辑器的配置方法。手动配置资源库，为保证下面的方法有效，请依照本文档描述的方式操作：

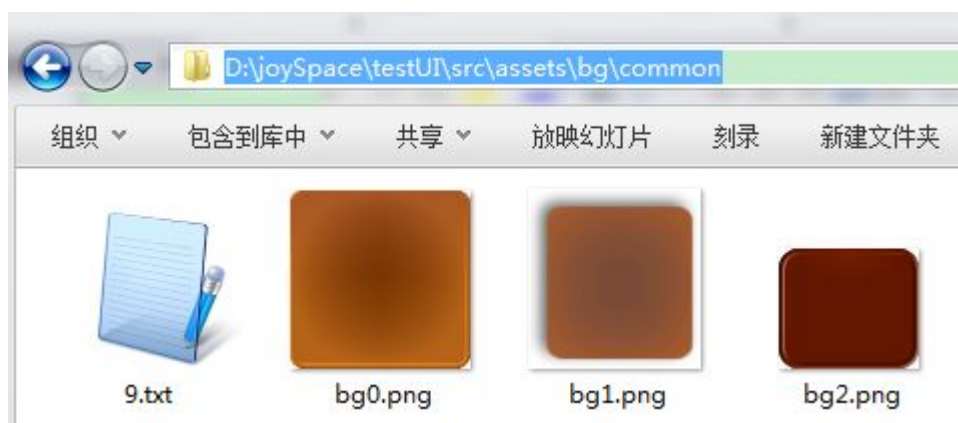
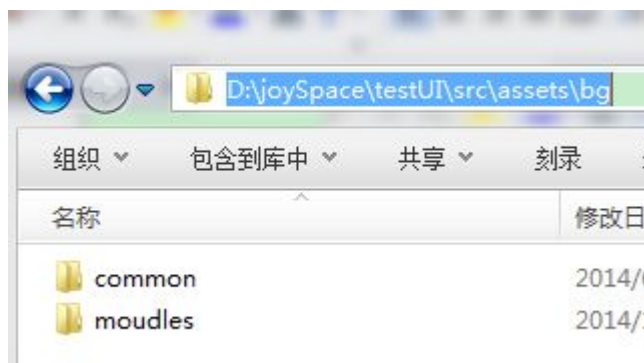
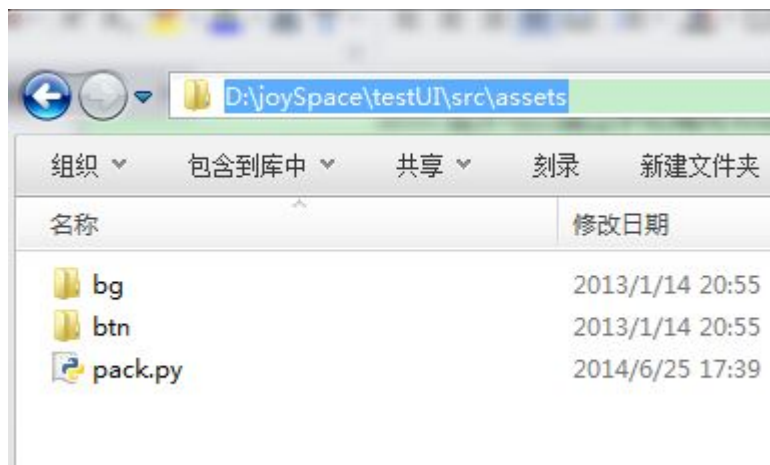
本例项目下载地址：<https://github.com/swellee/testUI/>

1. 在你的 as3 项目的 `src` 文件夹下，建立一个资源目录，该资源目录中放置所有图片资源及对图片资源九宫格配置信息文件。（本例中，新建 as3 项目“testUI”，`src` 文件夹下用于放置资源的文件夹命名为“assets”，如下图所示：）



2. 在资源文件夹 `assets` 中，放置脚本文件 `pack.py` (示例项目中已有，单独下载地址：<https://github.com/swellee/LeDoc/blob/master/pack.py>)，脚本会以自己所在的路径为起始，深度遍历当前及子级路径，将所有要处理的图片文件（脚本中可以配置哪些格式的图片文件会被处理），配合图片同级目录下的九宫格配置文件的信息，生成资源的嵌入代码到一个 `as` 文件中，该 `as` 文件即可作为资源库，为配置样式表做装备。脚本运行需要 `python 2.7.x` 环境，请确保你先安装了环境。

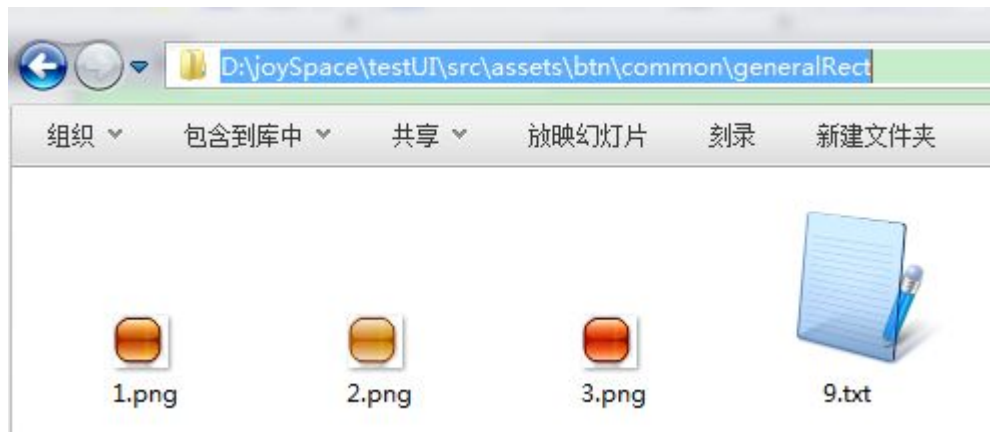
3.将你的资源（图片文件）放置在 assets 下或其子路径下。本例项目中，在 assets 下建立了两个子文件夹“bg”和“btn”，分别用于存放面板类组件资源和按钮类组件资源，子文件夹可以有更丰富的层级，如图：



在上面的图中可以看到，图片资源的同级目录中，有一个“9.txt”文件，该文件中存储着对同级目录中某些图片的九宫格配置信息，脚本会将配置信息对应资源名生成嵌入代码。打开“9.txt”，配置信息格式：包括两个字符串的数组形式，第一个元素是图片名（不带扩展名），第二个元素是九宫格信息，如图：


```
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
["bg0", "scaleGridTop=55, scaleGridLeft=55, scaleGridBottom=64, scaleGridRight=65"]
["bg1", "scaleGridTop=56, scaleGridLeft=55, scaleGridBottom=65, scaleGridRight=65"]
["bg2", "scaleGridTop=20, scaleGridLeft=30, scaleGridBottom=40, scaleGridRight=40"]
```

如果该路径中的所有图片使用相同的九宫格信息，则图片名使用“all_same”（该字符串可在脚本中自定义），如在一个放了按钮资源图片的路径下，按钮资源有 3 个图片，这三个图片使用相同的九宫格信息：



```
9.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
["all_same", "scaleGridTop=7, scaleGridLeft=15, scaleGridBottom=17, scaleGridRight=18"]
```

注意，不允许在一个“9.txt”文件中同时出现具体化的图片名及“all_same”，这样可能会导致脚本生成的代码混乱。

OK，下面来打开 assets 路径下的 pack.py 脚本，简单介绍下脚本的变量设置，当然了，如果你的项目结构跟示例完全一致，则无需改动脚本即可.....。

```
#-----global config infos-----
#bitmaps extention
bmpx = ['bmp', 'jpg', 'png']

#config scale info file name
cfg = '9.txt'
'''in current directory, if all the bitmap file use the same config info,
use the follow tag str as the info key, this model was usually take to cfg a butt
eg: ["all_same", "scaleGridTop=55, scaleGridLeft=55, scaleGridBottom=64, scaleGridR
cfg_info_ally_tag = "all_same"

#create code file
codefile = './Assets.as'
code_indent = '\t\t\t'

#whether show the "Done" word when progress finish, if not, the progress will
#exit automaticly, default is not
show_done_info = False#True

#-----functions definition-----#
```

这段代码区域，共有 6 个变量可配置：

bmpx 变量，此变量配置将被处理的图片格式，如果要添加新格式，向数组中添加相应的扩展名即可；

cfg 变量，此变量指出用于配置九宫格信息的文件名；

cfg_info_ally_tag 变量，此变量指出用于表示“该目录下所有资源使用统一的九宫格信息”的标签名；

codefile 变量，此变量指出生成嵌入代码的 as 文件路径（使用脚本所在路径的相对路径），修改此值需要在后面脚本中，写入 as 文件头时，相应的修改 as 文件头的包名。

code_indent 变量，此变量指出生成代码时的每行的缩进。

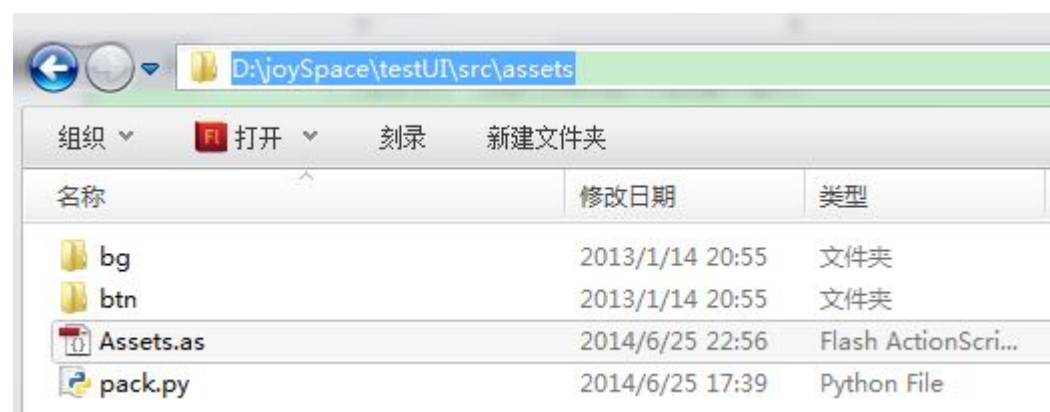
show_done_info 变量，此变量标识是否在脚本执行完成时，打印一个“Done”并等待用户按回车键才关闭脚本，默认 False，则完成时自动关闭脚本。

其他修改，请自行阅读脚本的后半部分：

```
#-----enter progress-----#  
  
if __name__ == "__main__":  
    import os  
  
    asset_root = os.getcwd()  
    asset_root_name = os.path.basename(asset_root)  
  
    '''script file head:'''  
    file_head = 'package ' + asset_root_name+'\n\  
{\n\t/**\n\t* @author leui\n\t*/\n\tpublic class Assets\n\t{\n\t\t'''script file tail'''  
    file_tail = code_indent+ 'public function Assets()\n\t}\n\t}\n\t'
```

.....

4.执行脚本，双击 pack.py，如果配置正确，脚本执行完后会生成一个 as 文件，本例中，生成了一个 Assets.as 文件：



5.打开生成的 as 文件，可以看到，资源名已相应地映射成了嵌入类代码，如路径 assets\bg\common 下的 bg0.png，映射成的嵌入类名为 bg_common_bg0。

```

package assets
{
    /**
     * @author leui
     */
    public class Assets
    {
        [Embed(source= "../bg/common/bg0.png", scaleGridTop=55, scaleGridLeft=55, scaleGridBottom=64, scaleGridRight=65)]
        public static const bg_common_bg0:Class;
        [Embed(source= "../bg/common/bg1.png", scaleGridTop=56, scaleGridLeft=55, scaleGridBottom=65, scaleGridRight=65)]
        public static const bg_common_bg1:Class;
    }
}

```

6.现在有了资源嵌入类，就可以使用它来为创建样式表提供支持。新建一个样式表类，本例项目中样式表类命名 MyStyleSheet，继承自 LeUI 的样式表类 LStyleSheet：

```

public class LStyleSheet implements IStyleSheet
{
    private var styleHash:LHash;
    public function LStyleSheet()
    {
        styleHash=new LHash();
        initStyleSet();
    }
}

```

LStyleSheet 类提供了对 IStyleSheet 接口的基本实现，子类重写 initStyleSet()方法来配置具体的样式信息。

```

public class MyStyleSheet extends LStyleSheet
{
    public function MyStyleSheet()
    {
        super();
    }
    override protected function initStyleSet():void
    {
    }
}

```

LeUI 中样式信息可从概念上分为：

基础样式信息--与资源直接对应的样式信息，对所有 LeUI 组件可用；

复态样式信息--使用基础样式信息作为动态属性（映射成状态样式），适用于复态组件（如 LButton）；

复合样式信息--使用基础样式信息作为动态属性（映射成子元素样式），适用于复合组件（如 LScrollBar）；

三者配置时以 styleVo 的 decoratotClass 来区分。

第 5 步生成的 Assets.as 文件中，除了生成资源的嵌入类代码外，还生成了一个保存所在嵌入类名称的数组，此时可以在 MyStyleSheet 类重写的 initStyleSet()方法里使用 for 循环语句将所在的嵌入类导入为基础样式信息，这些基础样式信息的 decoratotClass 配置成 LDecorator，并使用与资源类相同的名称作为样式名，如 bg/common/bg1.png 生成的嵌入类名为 bg_common_bg1，对应的基本样式信息 StyleVO 的 styleName 也同样为 bg_common_bg1：


```
//基本组件样式全部导入
var assetClasses:Array= Assets.assetCls;
for (var i:int = 0; i < assetClasses.length; i++)
{
    var st:StyleVO=new StyleVO();
    var assetName:String=assetClasses[i];
    st.styleName=assetName;
    st.assetClass=Assets[assetName]as Class;
    st.decoratorClass=LDecorator;
    putStyleVO(st);
}
```

实例化一个 LComponent 后，就可以应用这些基本样式。

如： *

```
var comp:LComponent = new LComponent();
comp.style = "bg_common_bg1";
addChild(comp);
```

当然，你也可以不使用与资源类相同的名称作为样式名：

比如，自定义一个名为“bg2”的基本样式信息：

```
paneSet=new StyleVO();
paneSet.styleName="bg2";
paneSet.assetClass=Assets.bg_common_bg2;
paneSet.decoratorClass=LDecorator;
putStyleVO(paneSet);
```

7.除了配置基本样式信息外，更常见的需求是配置复态组件样式和复合组件样式。首先介绍复态样式的配置：

仍然是 new 一个 StyleVO，设置你想要的样式名，然后将 decoratotClass 配置成

LMultiStateDecorator，通过动态属性添加状态样式：属性名即组件的状态名，属性值即该状态下的样式名。

典型例子--配置一个按钮的样式，LButton 组件有六种状态（普通常态、普通悬停、普通按下、选中常态、选中悬停、选中按下），其中“普通常态”的样式必须要有，其他可以不配，如果没有配，则按上述列举状态的顺序，使用前一种状态。例，如果只配了前三个状态，则选中时，顺位向前应用“普通按下”的状态样式。

下面的代码中，配置了一个名为“milkBtn”的复态样式，它使用动态属性添加了三个状态样式，这三个状态样式依次指向了名为 btn_common_milkRect_1、btn_common_milkRect_2、btn_common_milkRect_3 的基础样式信息（需保证样式表中能找到名为 btn_common_milkRect_1、..... 的 StyleVO）。

```
btnset=new StyleVO;
btnset.styleName="milkBtn";
btnset.user = "LButton,LToggleButton";
btnset.decoratorClass=LMultiStateDecorator;
btnset[LButton.BUTTON_STATE_MOUSE_OUT]="btn_common_milkRect_1";
btnset[LButton.BUTTON_STATE_MOUSE_OVER]="btn_common_milkRect_2";
btnset[LButton.BUTTON_STATE_MOUSE_DOWN]="btn_common_milkRect_3";
putStyleVO(btnset);
```

有了这个配置，实例化一个 LButton 便可应用此样式： *

```
var button:LButton = new LButton();
button.style = "milkBtn";
addChild(button);
```

复合样式的配置:

与复态样式类似, new 一个 StyleVO, 设置你想要的样式名, 然后将 decoratotClass 配置成 LCombineDecorator, 通过动态属性添加子元素样式: 属性名即组件的子元素名, 属性值即该子元素的样式名。

典型例子---配置滚动条 LScrollBar 的样式。打开 LScrollBar 源代码, 可以看到, 这是一个复合组件, 由一个背景条、减量按钮、增量按钮、滑块按钮 四个子元素组成:

```
/*滚动条
*/
public class LScrollBar extends LCombine
{
    /**背景*/
    public var ele_bg:LComponent;
    /**滑块*/
    public var ele_slider:LButton;
    /**增量按钮*/
    public var ele_increase_btn:LButton;
    /**减量按钮*/
    public var ele_decrease_btn:LButton;
```

在样式表中为它配置样式时, 要保证 LScrollBar 子元素的名称作为 StyleVO 的动态属性名, 子元素的样式名作为动态属性的值。如下:

```
//scrollbar
var scrSet:StyleVO=new StyleVO();
scrSet.styleName="LScrollBar";
scrSet.decoratorClass=LCombineDecorator;
scrSet.ele_bg="bg_moudles_scrollbarBg";
scrSet.ele_slider="btn_moudles_scrollBarThumb";
scrSet.ele_increase_btn="btn_moudles_scrollBarDown";
scrSet.ele_decrease_btn="btn_moudles_scrollBarUp";
putStyleVO(scrSet);
```

8.默认样式信息---LeUI 组件被添加到显示列表时, 会自动以自身类名作为样式名进行一次装饰, 如, 实例化一个 LPane, 当它被添加到显示列表时, 自动从样式表中搜索样式名为”LPane”的 StyleVO, 如果找到了, 即使用它的配置内容对实例装饰; 样式表中以 LeUI 组件类名作为样式各的样式信息, 就被称为默认样式信息。强烈建议用户为所有组件配置其默认样式信息!! 当然, 你可以在任何时候通过 xx.style = ‘xx’来应用其他样式。

更多配置样例, 请参考 testUI 项目中的 MyStyleSheet.as 文件。

LeUI 的编辑器将分为样式库编辑器和 GUI 编辑器, 敬请期待。

*注: 所有测试之前, 需先使用样式表实例 初始化 LeUI。初始化方法:

```
LUIManager.initAsStandard(stage,this,new MyStyleSheet());
```

使用 LeUI 组件

当样式表配置完成后，下文将针对主要组件的使用给出示例。

首先，初始化 LeUI。将样式表实例化，并提供给 LUIManager:

```
LUIManager.initAsStandard(stage, this, new MyStyleSheet());
```

标准初始化

参数:

root 舞台

uiContainer UI容器

styleSheet 样式表

SharedEventDispatcher UI组件共享的事件派发/监听对象

LeUI 组件的绝大多数组件，实例化时会默认从 UIConst 类的枚举值中取得一些参数执行初始化，因此，如果想修改这些默认值，就到 UIConst 中查找。

我们开始编写第一个示例:

LComponent:

```
var comp:LComponent = new LComponent();
comp.style = "bg_common_bg1";
comp.setWH(60,40);
comp.setXY(20,30);
addChild(comp);
```

代码行解释:

实例化一个 LComponent;

设置样式;

设置宽高;

设置坐标;

添加到显示列表;

运行截图:



注 1:LComponent 作为 LeUI 组件基类，典型的使用情况是，项目使用了 LeUI 框架，但某些地方需要尽量轻量级的组件来完成自定义的功能；类似的还有容器基类 LContainer。不过这类情况并不普遍，推荐使用已封装好的 LeUI 组件。

注 2：所在 LeUI 组件实例化后，当被添加到显示列表时，默认会从样式表中查找与自身类名一致的样式信息（称为默认样式，如 LButton 的默认样式为一个 styleName=="LButton"的 StyleVO），可以在任何时候给组件更换样式，但强烈建议为所有组件配置默认样式，因为这也是样式库存在的意义之一。

下面将 LeUI 组件按“控件”、“容器与布局”、“辅件”的分类，择其具有代表性的组件给出示例代码，并附上代码解释。更全面更详细的内容，请参阅源码及 api 页面。

LeUI 常用控件有：

```
LText,  
LTextArea,  
LButton,  
LToggleButton,  
LRadioButton,  
LCheckBox,  
LStepperH,  
LStepperV,  
LComboBox,  
LScrollBar,  
LTreeNode,  
LTree,
```

LText，文本组件。

```
var txt:LText = new LText();  
txt.text = "I'm a text";  
txt.setXY(30,20);  
txt.setWH(80,30);  
txt.setAlign(UiConst.TEXT_ALIGN_MIDDLE_CENTER);  
addChild(txt);
```

代码行解释：

实例化一个 LText；

设置文本内容；

设置坐标；

设置尺寸；

设置文本内容排布方式，如果不设置，则默认居中（通过 UiConst 枚举值设置）；

添加到显示列表；

运行截图：



LTextArea 和 **LText**，二者用法基本相同。**LTextArea** 适合于大段文本，且它实现了 **IViewport** 接口，可作为 **LScrollPane** 的视口，后面再细讲。

LButton，继承自 **LMultiState**，有六个状态（普通、悬停、按下、选中普通、选中悬停、选中按下），用来表现状态的组件是一个 **LText**，因此配置状态样式需要是针对 **LText** 类型的样式。第一个状态样式是必须的，其他状态样式如果没有配，则使用第一个状态样式。示例代码：

```
var button:LButton = new LButton();
button.setXY(20,30);
button.setWH(70,30);
button.setText("haha");
addChild(button);
```

代码行解释：

实例化；

（未显式设置样式，因此将自动应用默认样式）

设置坐标；

设置宽高；

设置文本；

添加到显示列表。

运行截图：



LToggleButton，继承自 **LButton**，添加了鼠标点击时对 **selected** 属性的自动切换。

示例代码：

```
var button:LToggleButton = new LToggleButton();
button.setXY(20,30);
button.setWH(70,30);
button.setText("haha");
addChild(button);
```

由于在样式表中，用 LButton 的默认样式信息简单复制了一份，改了下 styleName 作为 LToggleButton 的默认样式：

```
var btnset:StyleVO=new StyleVO;
btnset.styleName="LButton";
btnset.decoratorClass=LMultiStateDecorator;
btnset[LButton.BUTTON_STATE_MOUSE_OUT]="btnbg1";
btnset[LButton.BUTTON_STATE_MOUSE_OVER]="btnbg2";
btnset[LButton.BUTTON_STATE_MOUSE_DOWN]="btnbg3";
putStyleVO(btnset);

btnset = btnset.clone();
btnset.styleName = "LToggleButton";
putStyleVO(btnset);
```

所以此时，运行截图跟 LButton 是一样的。



但是，鼠标点击一下按钮，会发现它切换到了选中状态：



再点击一次又回切换回普通状态。

LRadioButton、LCheckBox 继承自 LCombine，由一个 LToggleButton（充当圆点或复选框）和一个 LText（充当文本）组合而成。圆点的宽高默认使用 UiConst.ICON_DEFAULT_SIZE，并在垂直方向居中；文本的宽高默认为 UiConst.TEXT_DEFAULT_WIDTH,UiConst.TEXT_DEFAULT_HEIGHT。虽然是复合组件，但用法上跟 LToggleButton 极为类似，此处不再例述。

步进器 **LStepperH**（横向）、**LStepperV**（纵向），由一个 LText、两个 LButton 组合而成，用于调整数字、翻页等情形，可以设置步进值、最大、最小值。示例代码：


```
var stp:LStepperH=new LStepperH(30);
stp.setWH(80,25);
stp.setXY(20,20);
stp.maxValue=20;
stp.curValue=2;
stp.minValue=1;
addChild(stp);
```

代码解释：

实例化一个横向步进器；

设置宽高；

设置坐标；

设置最大值；

设置当前值；

设置最小值；

添加到显示列表；

运行截图：



如果改成竖向步进器：



步进器的数值变化时，会派发 `LStepperEvent.VALUE_CHANGED` 事件。

LComBox，下拉菜单框，使用时需要提供用于菜单列表的数据集，示例代码：

```
var cbdata:Vector.<ComboxListVO>=new Vector.<ComboxListVO>;
cbdata.push(new ComboxListVO("good",1));
cbdata.push(new ComboxListVO("good2",12));
cbdata.push(new ComboxListVO("good3",13));
cbdata.push(new ComboxListVO("good4",14));
cbdata.push(new ComboxListVO("good5",15));
cbdata.push(new ComboxListVO("good6",15));
cbdata.push(new ComboxListVO("good7",15));
cbdata.push(new ComboxListVO("good8",15));
cbdata.push(new ComboxListVO("good9",15));
cbdata.push(new ComboxListVO("good10",15));
cbdata.push(new ComboxListVO("good11",15));
cbdata.push(new ComboxListVO("good12",15));
var cbx:LCombox=new LCombox();
cbx.setListData(cbdata);

cbx.setWH(100,22);
cbx.setXY(10,10);
addChild(cbx);
```

代码解释：

实例化一个下拉菜单数据集（`Vector.<ComboBoxListVO>`）；

向数据集中添加若干数据；

实例化一个下拉菜单；

将数据集提供给它；

设置尺寸（未显示下拉项时的尺寸）；

设置坐标；

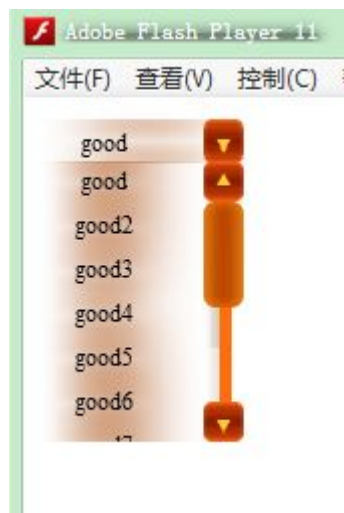
添加到显示列表；

运行截图：

默认显示数据集中第一项的文本。



点击下拉按钮：



注：菜单项的样式可通过 `setListData()` 方法的参数来设置。

LScrollBar 很少单独使用，`LScrollPane` 已集成横向和竖向的 `LScrollBar`，在此不多讲了，想了解的可去看 `LScrollPane` 的源码。

LTreeNode 和 **LTree** 相互依存，用来实现树组件。`LTree` 继承自 `LPane`，但它重写了 `append`、`addChild` 等方法，因此不能作为普通容器使用。下面给出一个树组件的示例代码：


```

var node0:LTreeNode=new LTreeNode("root");
for (var j:int = 0; j < 5; j++)
{
    var node:LTreeNode=new LTreeNode("node_1_"+(j+1));
    node0.appendChildrenNode(node);
    if(j%2==1)
    {
        for (var k:int = 0; k < 3; k++)
        {
            var subNode:LTreeNode=new LTreeNode("node_2_"+(k+1));
            node.appendChildrenNode(subNode);
        }
    }
}

var tree:LTree=new LTree(node0);
tree.setXY(20,20);
tree.setWH(100,200);
addChild(tree);

```

代码解释：

实例化一个树节点，此节点将作为根节点提供给树；

使用 for 循环，创建 5 个一级节点，并给索引为偶数的一级节点添加 3 个二级节点；

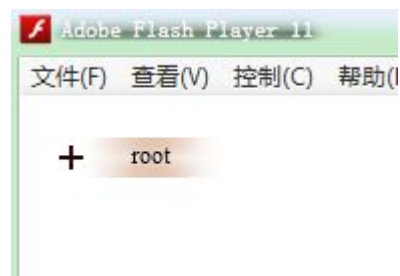
实例化一个树；

设置坐标；

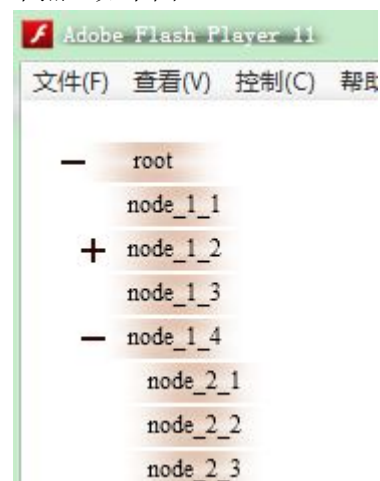
设置尺寸；

添加到显示列表；

运行截图：



点击根节点左侧的“+”图标，可以展开一级节点，一级节点如果有后代，可继续展开后代节点，如下图：



注：如果节点较多，建议将 LTree 作为视口，添加到 LScrollPane 中，以便在展开节点时，通过滚动面板显示完全的树结构。
关于控件，先说到这里，下面讨论容器。

容器与布局

LPane,
LBox,
LList,
LGrid,
LScrollPane,
LWindow,
LSeprator

LPane:普通容器，默认的布局管理器 do nothing，所以你可以放心地对添加进来的子对象设置坐标(而它的子类 LBox/LList/LGrid 容器的布局管理器会抹杀用户对子对象的坐标设置)。示例代码：

```
var pane:LPane = new LPane();  
pane.setWH(200,150);  
pane.setXY(100,30);  
addChild(pane);
```

代码行解释：

实例化一个 LPane；

设置宽高；

设置坐标；

添加到显示列表；

(这次未显式地设置样式，它将使用默认样式)

运行截图：



然后，创建一个按钮，并将它添加到上面的容器中去：

```
var button:LButton = new LButton();
button.setXY(20,30);
button.text = "haha";
button.style = "milkBtn";
pane.append(button);
```

注 1: LButton 通过 UIConst.as 提供了一组默认宽高，因此可以不显式地 setWH();

注 2: 向容器中添加子对象，推荐使用 append(), 当然也可以用原生的 addChild();

注 3: 向容器中添加多个子元素时，强烈建议使用 appendAll 方法，它比重复使用 append 效率更高！

运行截图：



LBox，继承自 LPane，可设置行列数，子对象将自动缩放，以撑满整个容器尺寸。

LBox:

```
var button:LToggleButton = new LToggleButton();
button.setXY(20,30);
button.setWH(70,30);
button.text = "haha";
addChild(button);

var button2:LButton = new LButton("hehehe");
button2.setXY(30,50);

var box:LBox = new LBox(10,10);
box.setWH(200,160);
box.setXY(20,20);
box.appendAll(button,button2);
addChild(box);
```

代码行解释：

实例化一个 LToggleButton;

设置坐标（无效，因为会被 LBox 的布局管理器重置坐标）;

设置尺寸（无效，因为会被 LBox 的布局管理器重置尺寸）;

设置标签文本;

实例化另一个 LButton,并在构造函数中传递标签文本;
设置坐标 (无效, 因为会被 LBox 的布局管理器重置坐标);

实例化一个 LBox, 并在构造函数中传递子元素的间距值;
设置尺寸;
设置坐标;
将两个 LButton 添加进来;
将 LBox 添加到显示列表;

运行截图:



还可以对 LBox 设置布局朝向, 默认为竖向, 如果想改为横向, 添加如下代码:

```
box.direction = UiConst.HORIZONTAL;
```

再次运行, 截图:



LList, 列表容器, 继承自 LBox。它不缩放子元素, 仅仅将子元素依据设置的间隔进行排列。

```

var ll:LList = new LList(0,10);
ll.setXY(20,20);
ll.setWH(200,300);

var aa:Array=[];
for (var i:int = 0; i < 20; i++)
{
    var pp:LPane=new LPane();
    pp.setWH(40,40);
    pp.data=i;
    aa.push(pp);
}
ll.appendAll.apply(null,aa);

addChild(ll);

```

代码行解释：

实例化一个 LList，通过构造函数参数设置子元素的布局间隔，以及是否竖向排列；

设置坐标；

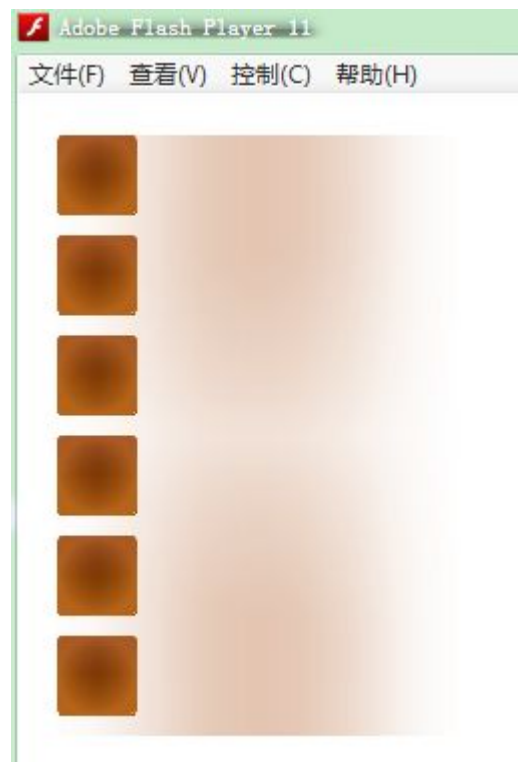
设置尺寸；

通过一个 for 循环，生成 10 个 LPane(LeUI 基类 LComponent 有一个 data 属性，可以为组件临时关联数据提供便利，但建议用户自己设计更系统的数据管理机制)，放入数组；

通过 appendAll 方法，将之前生成的 10 个 LPane 放入 LList 容器。

将 LList 实例添加到显示列表；

运行截图：



LGrid，继承自 LList，为阵列容器，可设置行列数，LGrid 还有一个 canScaleElement 属性，默认为 true，用于全局控制是否对子元素进行统一缩放，以及是否自动修正行列数，另外此值影响 hGap 和 vGap 的意义。详细请参阅 api。先来看一下使用默认会对元素统一缩放的 LGrid:

```
var ll:LGrid = new LGrid(4,4,6,6);
ll.setXY(20,20);
ll.setWH(200,260);

var aa:Array=[];
for (var i:int = 0; i < 20; i++)
{
    var pp:LPane=new LPane();
    pp.setWH(50,30);
    pp.data=i;
    aa.push(pp);
}
addChild(ll);
ll.appendAll.apply(null,aa);
```

代码行解释:

生成一个 LGrid，并通过构造函数参数设置 4 行 4 列及子元素的间距;

设置坐标;

设置尺寸;

For 循环生成 20 个 LPane;

将 LGrid 实例添加到显示列表;

将生成的 20 个 LPane 添加到 LGrid 容器;

运行截图:



观察发现，显示了 4 列 5 行。

注 1: LGrid 会对子元素的行列数进行修正，因为添加进来的元素总数并不总等于设置的行数乘以列数，构造函数中最后一个参数可以设置当二者不相等时，是否以列数为准。例如上例中，列数 4，行数 4，实际添加了 20 个元素， $4 \times 4 \neq 20$ ，以列数为准，自动修正为 4 列、5 行。

注 2: LGrid 会根据自身尺寸及子元素个数，对子元素尺寸进行缩放到统一的元素尺寸，如果不想被缩放，可设置子元素的 `canScaleX` 和 `canScaleY` 属性为 `false`，但不推荐这么做，因为这将使布局后子元素显得不协调。

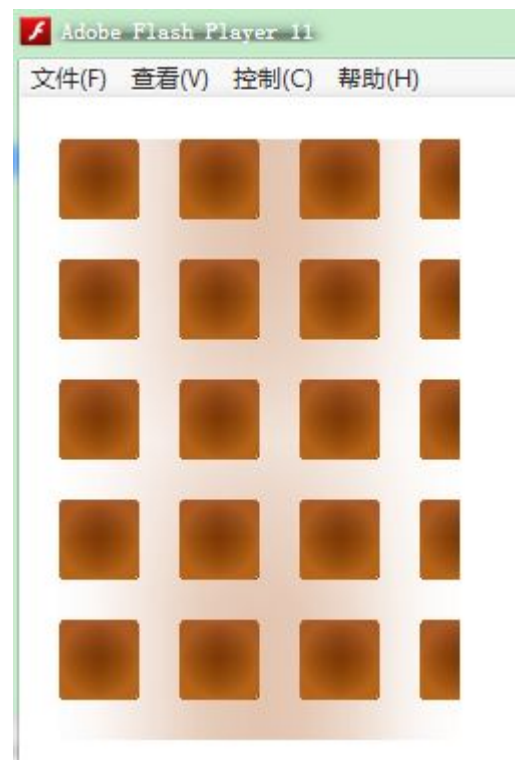
现在，看看如果设置 `canScaleElement=false` 的情况下，如何使用：

```
var ll:LGrid = new LGrid(4,4,60,60);
ll.canScaleElement=false;
ll.setXY(20,20);
ll.setWH(200,300);

var aa:Array=[];
for (var i:int = 0; i < 20; i++)
{
    var pp:LPane=new LPane();
    pp.setWH(40,40);
    pp.data=i;
    aa.push(pp);
}
ll.appendAll.apply(null,aa);

addChild(ll);
```

注意前两行，可以发现，首先是构造函数中，参数的变化；其次设置了 `canScaleElement=false`；运行截图：



至于原因，api 和源码里的注释里可以找到答案，此处不再复述。

LScrollPane，确切的说，它不能算作容器，而是一个为容器服务的滚动管理组件，因为它需要一个视口（IViewport）作为内容物，并响应视口的尺寸变化，以更新滚动条的显隐。LPane 及 LTextArea 实现了 IViewport 接口，因此常用容器（LPane 及其子类）和文本域可以作为视口。

视口的尺寸是由 LScrollPane 的尺寸及视口内容而自动设置的。

首先，使用容器作为视口：

```
var ll:LList = new LList(0,10);
ll.setXY(20,20);
ll.setWH(200,260);

var aa:Array=[];
for (var i:int = 0; i < 20; i++)
{
    var pp:LPane=new LPane();
    pp.setWH(250,30);
    pp.data=i;
    aa.push(pp);
}
ll.appendAll.apply(null,aa);

scrPane=new LScrollPane(ll);
addChild(scrPane);
scrPane.setWH(200,200);
scrPane.setXY(100,50);
```

代码解释：

实例化一个 LList;

设置坐标、尺寸（无效，因为 LScrollPane 会重置视口坐标及尺寸）;

向 LList 中添加 20 个子对象;

.....

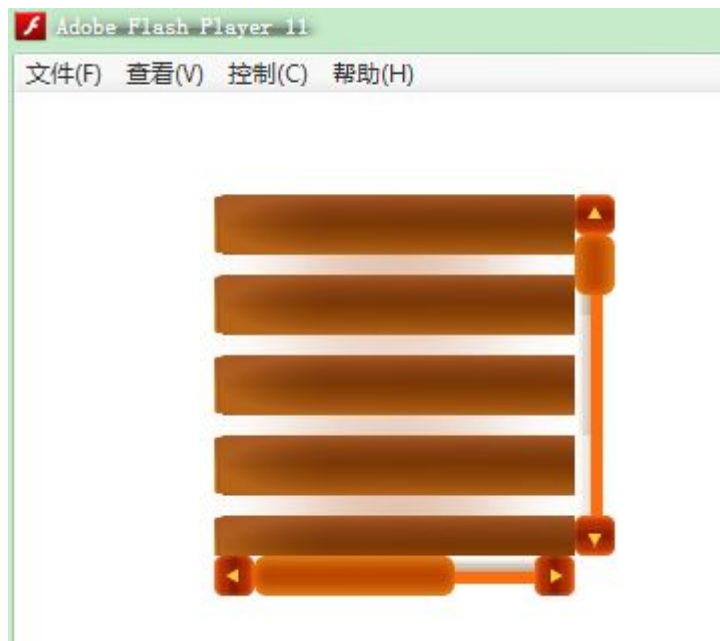
实例化一个 LScrollPane，并通过构造函数参数，将 LList 实例作为视口传入;

添加 LScrollPane 到显示列表;

设置尺寸;

设置坐标;

运行截图：



注：可以设置 LScrollPane 滚动条的显隐策略（自动/从不显示/总是显示，默认为自动）
例如，设置横向滚动条总是显示，代码如下：

```
scrPane.hsbPolicy = UiConst.SCROLLPANE_BAR_POLICY_ALWAYS;
```

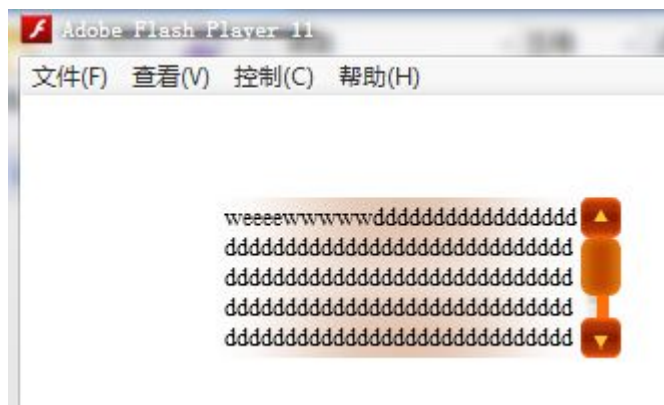
现在，使用文本域 LTextArea 作为视口：

```
var ll:LTextArea = new LTextArea();  
ll.text = "weeeewwwww";  
scrPane=new LScrollPane(ll);  
addChild(scrPane);  
scrPane.setWH(200,100);  
scrPane.setXY(100,50);
```

运行截图：



可以看到，因文字较少，没有显示滚动条。
此时，手动在文本区内输入更多字符，直到当前尺寸显示不下，
滚动条自动显现：



注：文本域作为视口时，只可能显示竖向滚动条。

LSeperator 为分隔线，实例化时需在构造函数里指出是横向或竖向，横向时，height 固定为 UIConst.SEPRATOR_INIT_SIZE，竖向时，width 固定为 UIConst.SEPRATOR_INIT_SIZE。

LWindow，窗体，是 LeUI 中等级最高的复合组件，也是游戏开发当中最直接的组件。由标题条、关闭按钮、内容面板组成。内容面板是一个 LPane 容器，手动向 LWindow 中添加的显示对象，默认会被添加到内容面板中；LWindow 实现了 IPopup 接口，提供 show()、hide()方法用于显示/隐藏窗体，窗体的父容器为 LUIManager 初始化时，用户提供的 uiContainer。

下面给出一个略为复杂的示例，此示例中，初始化一个窗体，向其中添加了一个文本组件、一个树组件、一个竖向分隔线、一个滚动面板，滚动面板以一个 LGrid 作为视口：

```
//实例化一个树节点，作为根节点
var rootNode:LTreeNode = new LTreeNode("武学秘籍");

//一级节点
var shaolin:LTreeNode = new LTreeNode("少林绝学");
//二级节点
var shaolinBook:LTreeNode = new LTreeNode("易筋经");
var shaolinBook2:LTreeNode = new LTreeNode("金钟罩");
shaolin.appendChildNode(shaolinBook,shaolinBook2);

//一级节点
var wudang:LTreeNode = new LTreeNode("武当剑法");
//二级节点
var wudangBook:LTreeNode = new LTreeNode("流星剑");
var wudangBook2:LTreeNode = new LTreeNode("穿杨剑");
wudang.appendChildNode(wudangBook,wudangBook2);

//一级节点
var gaibang:LTreeNode = new LTreeNode("丐帮秘传");
//二级节点
var gaibangBook:LTreeNode = new LTreeNode("打狗棒");
gaibang.appendChildNode(gaibangBook);

//将一级节点添加到根节点
rootNode.appendChildNode(shaolin,wudang,gaibang);

//实例化一个树组件，传入根节点、设置布局间距、节点统一尺寸
var tree:LTree=new LTree(rootNode,10,4,100,24);
tree.setXY(6,10);
tree.setWH(100,300);
```

```

//实例化一个分隔线
var sp:LSeparator = new LSeparator(false);
sp.setWH(-1,260);
sp.setXY(108,6);

//实例化一个文本，放在右侧，当树组件选中一个节点时，
//此文本显示该节点的数据
var treeLabel:LText = new LText("我使用双截棍，叽叽喳喳",false);
treeLabel.setWH(300,26);
treeLabel.setXY(110,5);
//为树添加选中节点时的回调函数
tree.listenSelectedNodeChange(onSelectNodeFun);
//回调函数
function onSelectNodeFun():void
{
    treeLabel.text = tree.selectedNode.text;
}

//实例化一个LGrid作为视口
var ll:LGrid = new LGrid(4,4,190,60);
ll.canScaleElement= false;

//向视口中添加20个小东东
var aa:Array=[];
for (var i:int = 0; i < 20; i++)
{
    var pp:LPane=new LPane();
    pp.style = "bg1";
    pp.setWH(180,50);
    pp.data=i;
    aa.push(pp);
}
ll.appendAll.apply(null,aa);
//实例化一个滚动面板，将ll作为视口传入
var scrIPane :LScrollPane = new LScrollPane(ll);
scrIPane.setWH(330,220);
scrIPane.setXY(114,35);

//实例化一个窗体
var win:LWindow = new LWindow("藏经阁");
win.setWH(460,300);
//将上述几个东东，添加到窗体
win.addContent(tree,sp,treeLabel,scrIPane);

//显示窗体
win.show();

```

运行截图：



点击展开树组件，并选中一个节点：



辅件

LeUI 中将辅助、工具类的复合组件，称为辅件，主要有：

Alert,
LMenu

Alert，其实是一个简化了的窗体，用法跟 flex 的同名组件类似，示例代码：

```
Alert.show("my name is alert","love \nlove love \n love love love \n oh shit!",Alert.OK|Alert.CANCEL,true,true,onclickFun,22);

function onclickFun(btn:int,xx):void
{
    if(btn == Alert.OK)
    {
        trace(xx);
    }
}
```

运行截图：



细节用法，请参阅 api。

LMenu，菜单组件，需要提供菜单数据集，它会使用此数据集生成相应的菜单项。示例

代码：

```
var datas:Vector.<MenuItemVO>=new Vector.<MenuItemVO>();
for (var i2:int = 0; i2 < 10; i2++)
{
    var mvo:MenuItemVO=new MenuItemVO("menuItem"+i2);
    if(i2%3==0)
    {
        mvo.subMenuItemVos=new Vector.<MenuItemVO>;
        mvo.subMenuItemVos.push(new MenuItemVO("subItem"));
        mvo.subMenuItemVos.push(new MenuItemVO("subItem"));
        mvo.subMenuItemVos.push(new MenuItemVO("subItem"));
        mvo.subMenuItemVos.push(new MenuItemVO("subItem"));
    }
    datas.push(mvo);
}

var btn:LButton = new LButton("click me");
btn.setXY(30,40);
addChild(btn);

var menu:LMenu=LMenu.createMenu(btn,datas,true,menuCallFun);

function menuCallFun(vo:MenuItemVO):void
{
    trace("just click a menuItem>>" + vo.text);
}
```


代码解释:

实例化一个菜单数据集 datas;

For 循环向数据集中添加若干 MenuItemVO;

(for 循环索引逢 3 的倍数, 就为它设置子级菜单数据集);

实例化一个 LButton 作为菜单触发器 (触发器为 InteractiveObject 类型, 即可交互对象);

通过 LMenu 静态函数 createMenu() 并提供触发器、数据集、回调函数等, 生成一个 LMenu 实例;

注 1: 也可以自己先 new 一个 LMenu, 然后设置触发器、数据集...。即, 下面这行代码:

```
var menu:LMenu=LMenu.createMenu(btn,datas,true,menuCallFun);
```

可以替换为:

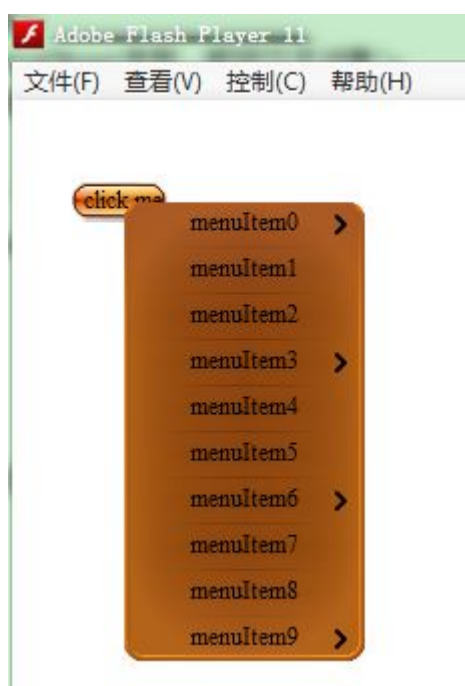
```
var menu:LMenu=new LMenu();  
menu.setInvoker(btn);  
menu.setMenuData(datas,true,menuCallFun);
```

注 2: 菜单点击的回调函数, 必需以 MenuItemVO 类型为参数, 此参数表示被点击的菜单项的 MenuItemVO。

运行截图:



点击这个按钮:



鼠标滑过一个带小箭头的菜单项 (表示此项有子级菜单):



点击一个菜单项（有子级菜单的项，自身不响应点击），会发现回调函数打印出了该菜单项的文本。

除了上边两个辅件外，还有一个隐性辅件：LTrace。

LTrace 用于打印和记录系统信息，方法有 `Log()`、`warning()`。这两个方法会将信息打印到控制台，并记录到内存中，它监听一个热键（键值由 `UiConst.LTRACE_HOTKEY` 定义，可以在运行时修改），按下 `ctrl+热键`，弹出一个 `log` 窗口，两次按下 `ctrl+热键` 则关闭 `log` 窗口，方便游戏发布后，在玩家机器上调出日志信息。`Log` 窗口中有“`copy`”和“`clear`”按钮，用于复制或清除 `Log` 信息，也可以在项目中，调用 `LTrace.clear()` 来在合适的时机清除 `Log` 信息。

Log 窗口截图：



扩展 LeUI

LeUI 提供了笔者认为在页游开发中几乎所有常用的 UI 组件，但用户自己的项目总会有这样那样的新需求，这些新的需求我相信通过使用 LeUI 现有组件也可以实现，不过基于 OOP 的理念，总希望将这些新需求封装成一个新组件，此时，你可以扩展 LeUI。下面给出两个例子。

Image，多数 UI 框架中都有这个组件，用于显示加载的图片。LeUI 之所以没有集成这个组件，是基于两个原因，一是因为这个组件在 LeUI 框架下实现起来太过简单，可以留给用户自己扩展；二是因为（这个是最主要的原因）在游戏开发中，**Image** 组件通常要跟项目自身的加载系统或资源管理系统对接，即，笔者推荐用户基于自己项目的加载\资源系统，来扩展 LeUI，完成自己的 **Image** 组件。

比如，我的项目已经写好了资源系统，基于这个资源系统，**Image** 代码如下：


```

package
{
    import core.App;

    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.DisplayObject;

    import org.leui.components.LComponent;

    public class Image extends LComponent
    {
        private var _source:*;
        public function Image(src:* = null)
        {
            if(src) this.source = src;
        }

        public function get source():*
        {
            return _source;
        }

        public function set source(value:*)void
        {
            _source = value;
            if(value==null)//清除缓存
            {
                clearBg();
            }
            else if(value is DisplayObject)//displayobj
            {
                setBg(value);
            }
            else if(value is String)//url
            {
                App.sysRes.getBitmapData(value,onLoaded);
            }
        }

        private function onLoaded(bmd:BitmapData):void
        {
            var bmp:Bitmap = new Bitmap(bmd);
            setBg(bmp);
        }
    }
}

```

这个 Image 通过 source 来设置内容，可以直接提供内容资源，也可以是 url，如果设置 source=null，则将内容清空。

Image 太简单？下面给出一个复杂的扩展示例：

Slider，滑轨组件，这是一个复合组件，由一个滑块按钮 和 一个滑轨背景条 组成，其中滑块可以在滑轨所在的线上进行拖动，拖动导致 Slider 值的变化，值的变化通过事件或回调通知外界。代码雏形如下：

```
package
{
    import flash.events.MouseEvent;

    import org.leui.components.LButton;
    import org.leui.components.LCombine;
    import org.leui.components.LComponent;
    import org.leui.utils.UiConst;
    import org.leui.vos.ChildStyleHashVO;

    public class Slider extends LCombine
    {
        /**
         * 滑块按钮
         */
        public var ele_dot_btn:LButton;
        /**
         * 背景条
         */
        public var ele_bg_bar:LComponent;
        public function Slider()
        {
            super();
        }
        //扩展LCombine, 需重写此函数, 以初始化 此新组件 中的 子元素
        override protected function initElements():void
        {
            ele_bg_bar = new LComponent();
            ele_dot_btn = new LButton();
            //一般情况下, 通过UiConst的静态常量, 来给予元素设置默认尺寸
            ele_bg_bar.setWH(UiConst.XXXXXX,UiConst.XXXXXX);
            ele_dot_btn.setWH(UiConst.XXXXXX,UiConst.XXXXXX);
            //将子元素添加到显示
            appendAll(ele_bg_bar,ele_dot_btn);
        }
    }
}
```

```

//重写此函数，以便将此类的子元素名称，添加到样式映射
override protected function initElementStyleHash():void
{
    super.initElementStyleHash();
    elementStyleHash.push(new ChildStyleHashVO("ele_bg_bar"));
    elementStyleHash.push(new ChildStyleHashVO("ele_dot_btn"));
}

//重写此函数，以便给某个子元素添加事件
override protected function addEvents():void
{
    super.addEvents();
    ele_dot_btn.addEventListener(MouseEvent.CLICK,onDotBtnMsDown);
}

//重写此函数，以便在销毁时，将之前添加的事件监听移除
override protected function removeEvents():void
{
    super.removeEvents();
    ele_dot_btn.removeEventListener(MouseEvent.CLICK,onDotBtnMsDown);
}

protected function onDotBtnMsDown(event:MouseEvent):void
{
    //在这个函数里，添加对滑块拖动的监听和控制
}

//重写此函数，以设置默认的布局管理器
override public function getLayoutManager():Class
{
    return _layoutManager ||= SliderLayout;
}
}
}

```

然后再写一个 Slider 的布局管理器（需实现 ILayoutManager 接口）SliderLayout，代码雏形如下：

```

package
{
    import org.leui.core.ILayoutContainer;
    import org.leui.core.ILayoutManager;
    import org.leui.utils.UiConst;

    public class SliderLayout implements ILayoutManager
    {
        public function SliderLayout()
        {
        }

        public function doLayout(container:ILayoutContainer):void
        {
            var slidr:Slider = container as Slider;
            if(!slidr)return; //限定类型

            //开始对Slider的子元素进行布局
            slidr.ele_bg_bar.setWH(UiConst.XXXXX,slidr.height);
        }
    }
}

```

此时，把自己要细化的功能添加进去，这个组件就可以使用了。但一般情况下，建议到样式表中为它配置默认样式：

```
var stp:StyleVO=new StyleVO;
stp.styleName="Slider";
stp.decoratorClass=LCombineDecorator;
stp.ele_dot_btn="milkBtn";
stp.ele_bg_bar="bg0";
putStyleVO(stp);
```

注 1：因为 Slider 扩展自 LCombine，所以样式信息的 decoratorClass 要配成 LCombineDecorator；

注 2：Slider 有两个子元素，要通过 StyleVO 的动态属性配它们的样式。

LeUI 组件在基类中使用了模板模式规范了子类的初始化，扩展 LeUI 组件时，请遵照模板，重写相关的 protected 初始化函数，不然可能会出错。其次，尽量使用组合的方式而非继承的方式来写新的组件。

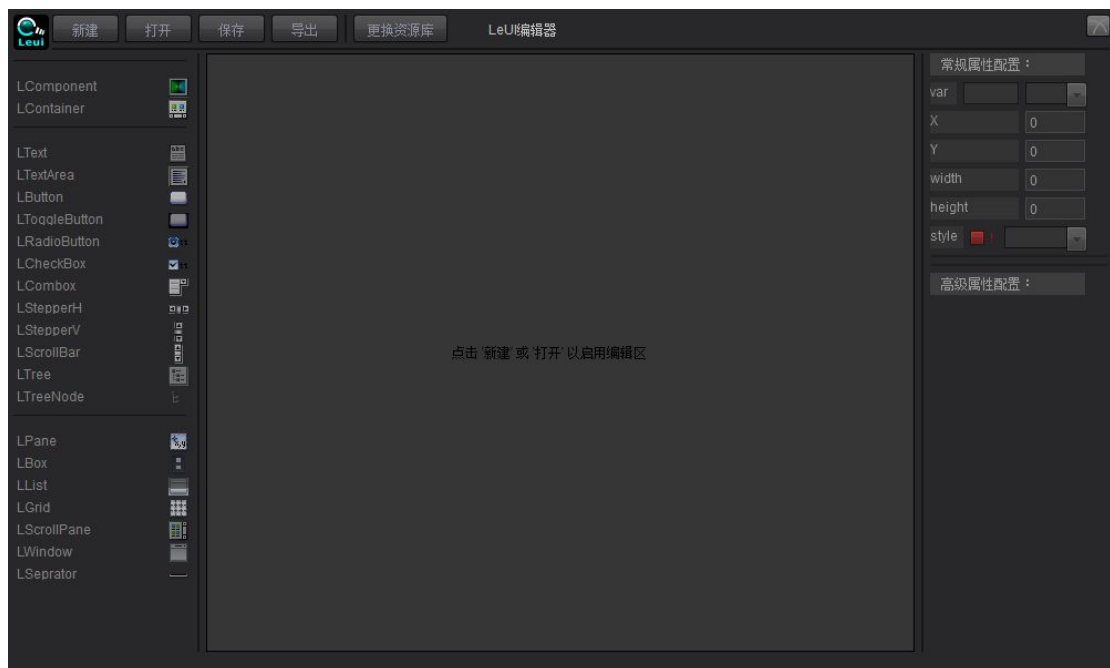
五、UI 编辑器的使用

LeUI 的御用编辑器，经过这段时间下班后的赶工，Beta 版已发布，下载地址与本文档位于同一 git 项目下：<https://github.com/swellee/LeDoc> 下面对 UI 编辑器的工作流程作简单介绍。

UI 编辑器（下称“编辑器”）是一个 air 安装包，安装包内嵌了一套简单的示例样式资源库。当前版本的编辑器提供了对 UI 文件的编辑、保存/打开、导出为 as 代码、更换资源库等功能。依次说明。

新建 UI

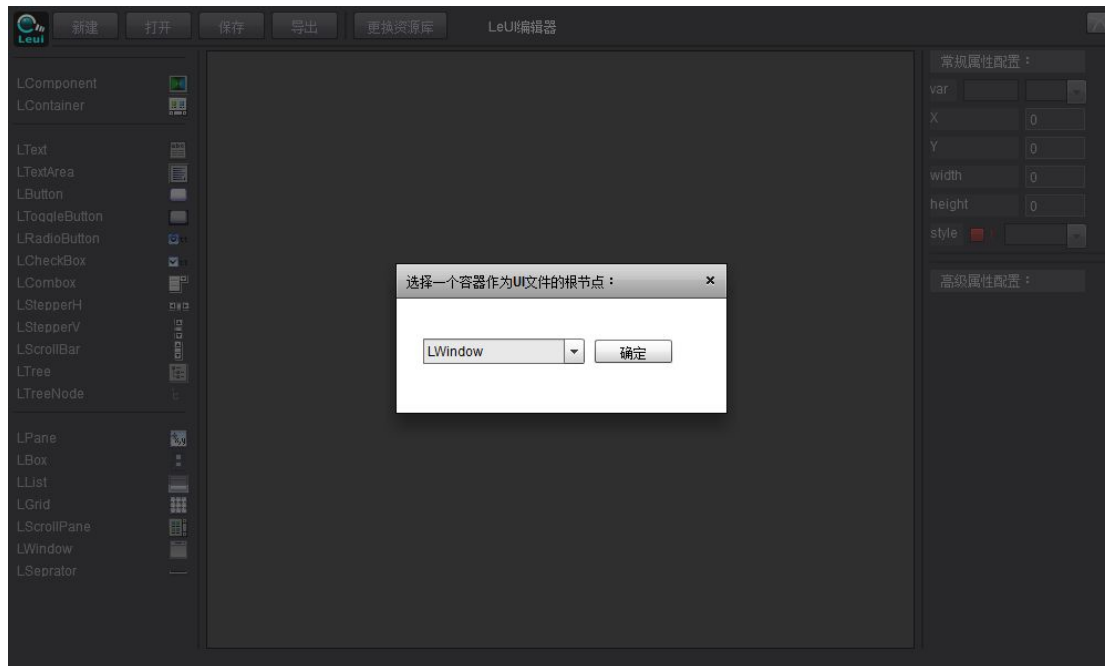
启动编辑器，界面如下：



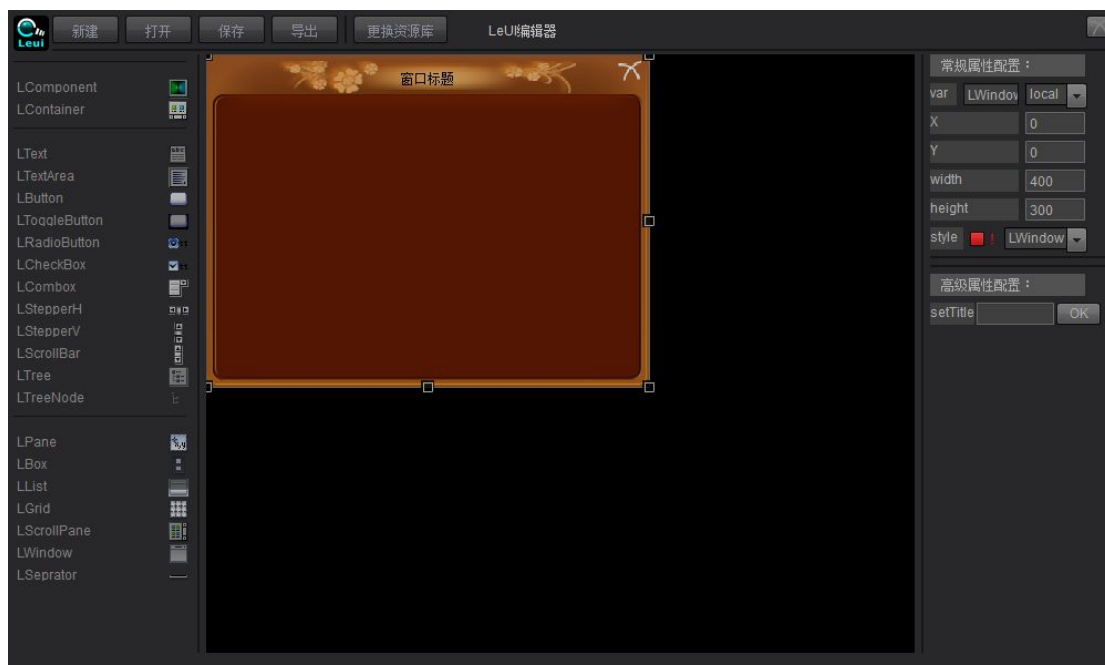
编辑器上方是菜单栏，菜单栏下为功能区。功能区又分左、中、右三个部分。其中左边是目前编辑所支持的 LeUI 组件列表（一些辅助类组件，如 Alert\LMenu 等由于使用情形的特殊性，未添加进来），中部是编辑区（下称“舞台”），右侧是属性设置区。

可以看到，刚启动时，舞台中央有一行提示：点击“新建”或“打开”以启用编辑区。即，当前舞台未激活，需要新建或打开 UI 来激活。

首先，点击菜单栏的“新建”按钮，会弹出一个对话框：

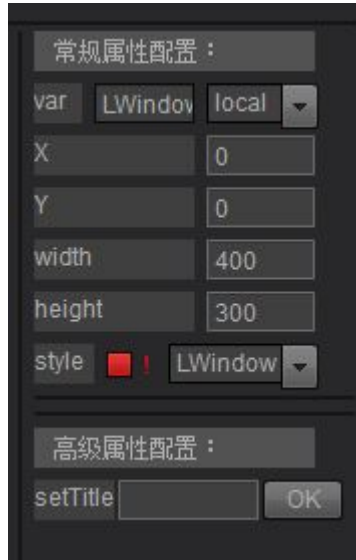


UI 文件的根节点必须是 LeUI 容器组件，最常见的情形是使用窗口组件 LWindow，因为游戏中的 UI 通常都是以窗口形式呈现的，当然你也可以选择其他 LeUI 容器作为 UI 文件的根节点以便可以更自由地控制要编辑的 UI。示例中，我们选择默认的 LWindow 作为根节点，点“确定”后，舞台被激活，并生成了一个 LWindow 的实例：



设置属性

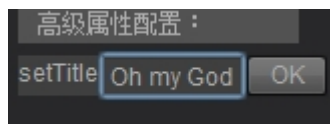
当一个新组件的实例被添加到舞台，它默认处于选中状态，处于选中状态的组件，会激活编辑器右侧的属性设置区：



属性设置区，分为[常规属性](#)设置和[高级属性](#)设置。

[上部分是常规属性](#)，即每个 LeUI 组件都有的属性；在对应的属性输入框中，输入相应的值，点击别处，或者按下键盘上的 Enter 键或 Tab 键，该属性值就会作用到舞台上选中的组件实例。


[下方是高级属性](#)，会根据组件的自身特点，添加该组件特有的属性配置入口，如上图中，LWindow 的高级属性区，有一个 setTitle 的属性栏，是用来设置窗体组件的标题的，输入几个字符：Oh my God



之后点“ok”按钮，或者敲击键盘上的 Enter 键，舞台上选中的 Lwindow 实例就更新了它的标题：



对于常规属性中的 X、Y、width、height，除了可以通过属性设置区来调节外，更方便的是可以直接在舞台上对选中的组件实例进行拖拽、拉伸来调节。

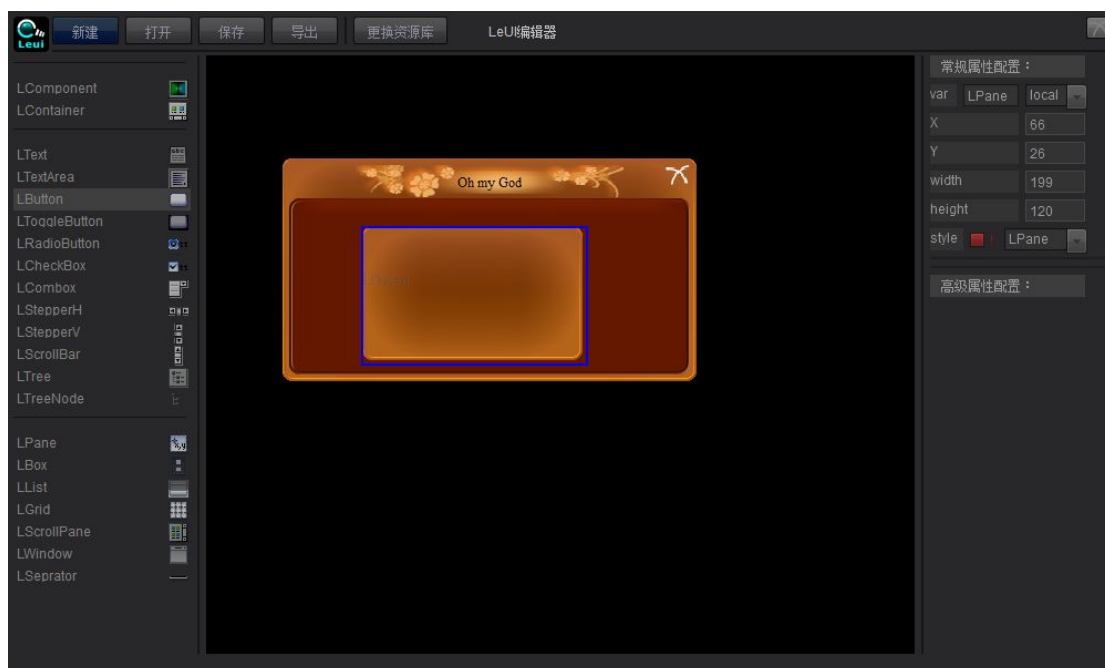
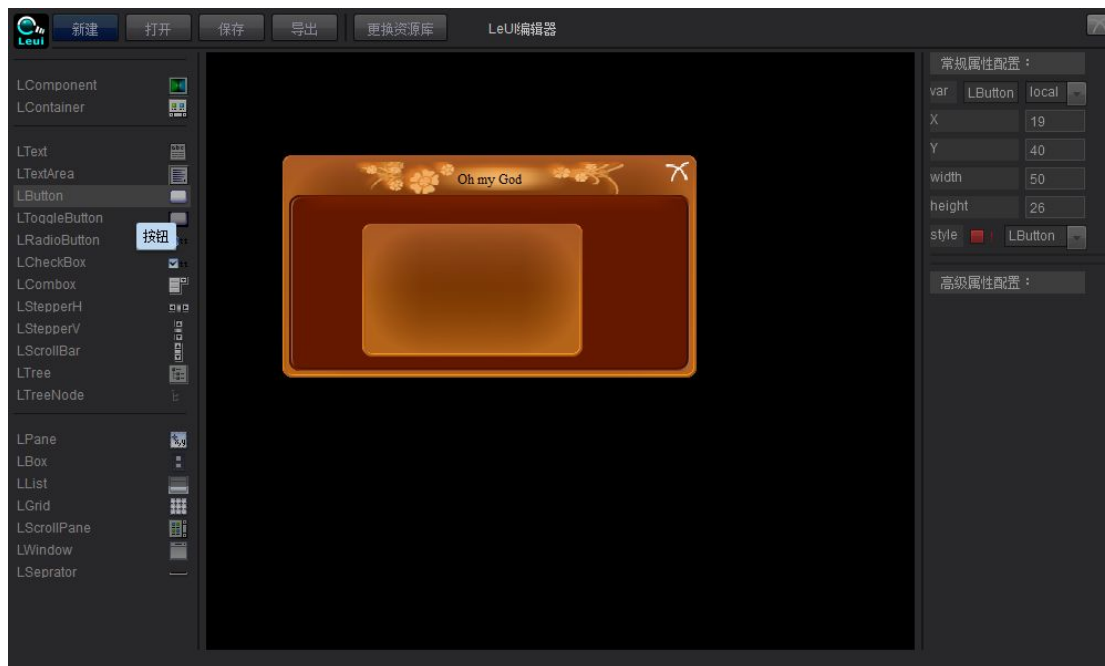
组件实例被选中时，会显示调节点：，其中只有右下方的三个调节点会响应拖动。

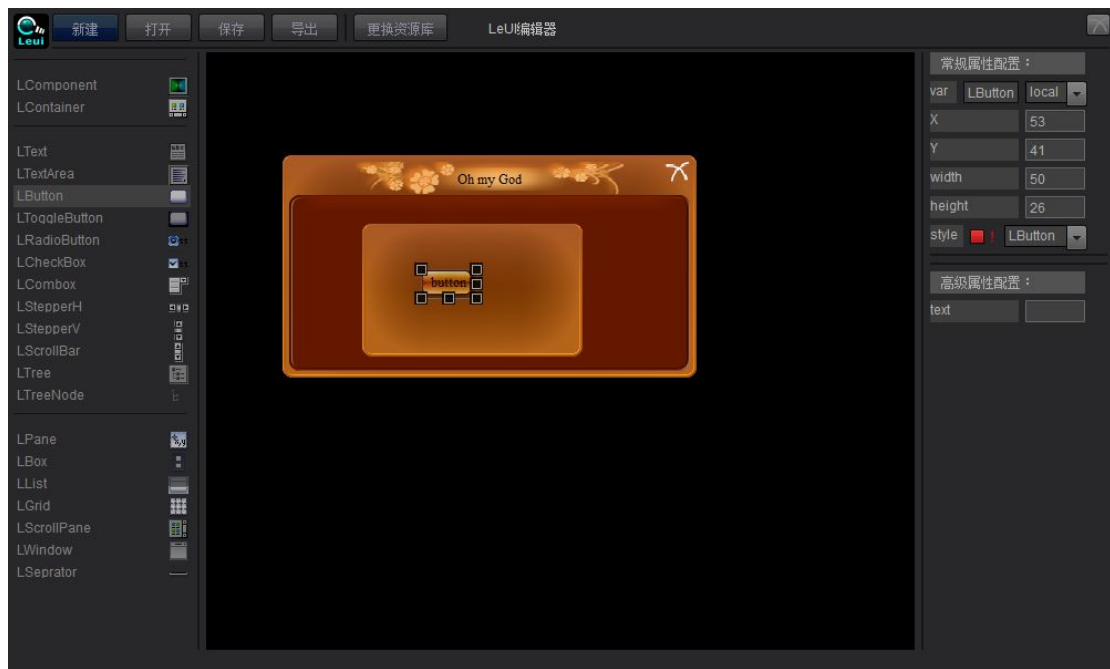


此外，还可以使用键盘上的方向键来调节舞台上选中的组件实例的坐标。

添加新组件

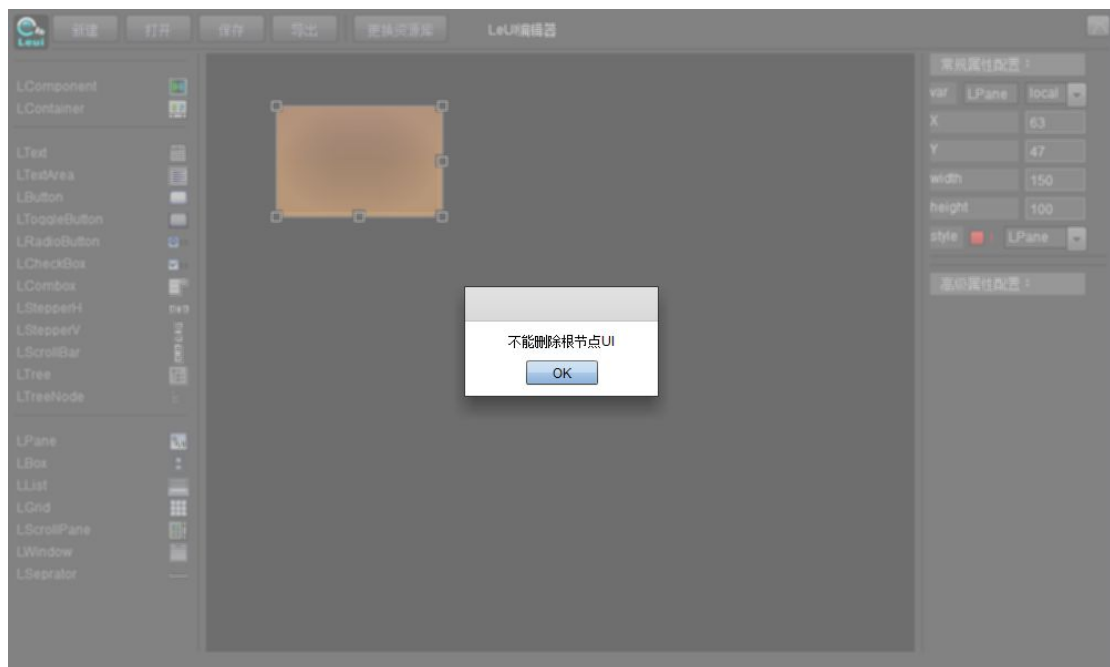
有了 UI 的根节点之后，要添加新的组件实例，方法是：在编辑器左侧的组件列表中，选中你要添加的组件，按住鼠标拖拽到舞台区，注意，当拖拽鼠标经过舞台上的某个 LeUI 容器实例时，该实例会显示一个蓝色描边，以示它准备接收新组件，此时松开鼠标，新的组件会被添加到该容器中。





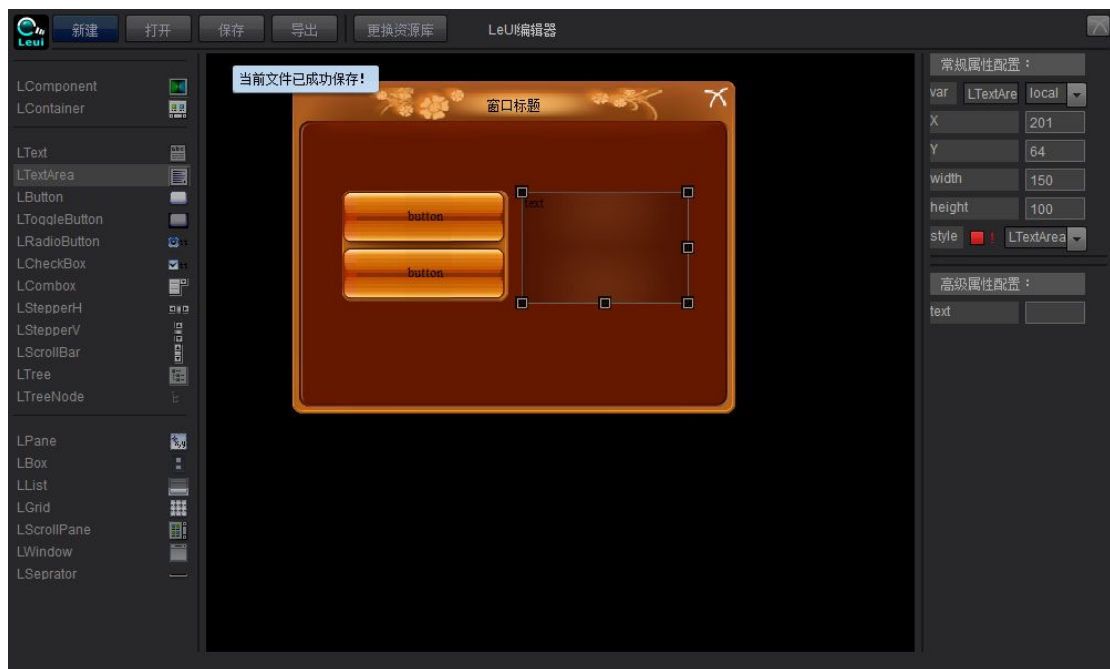
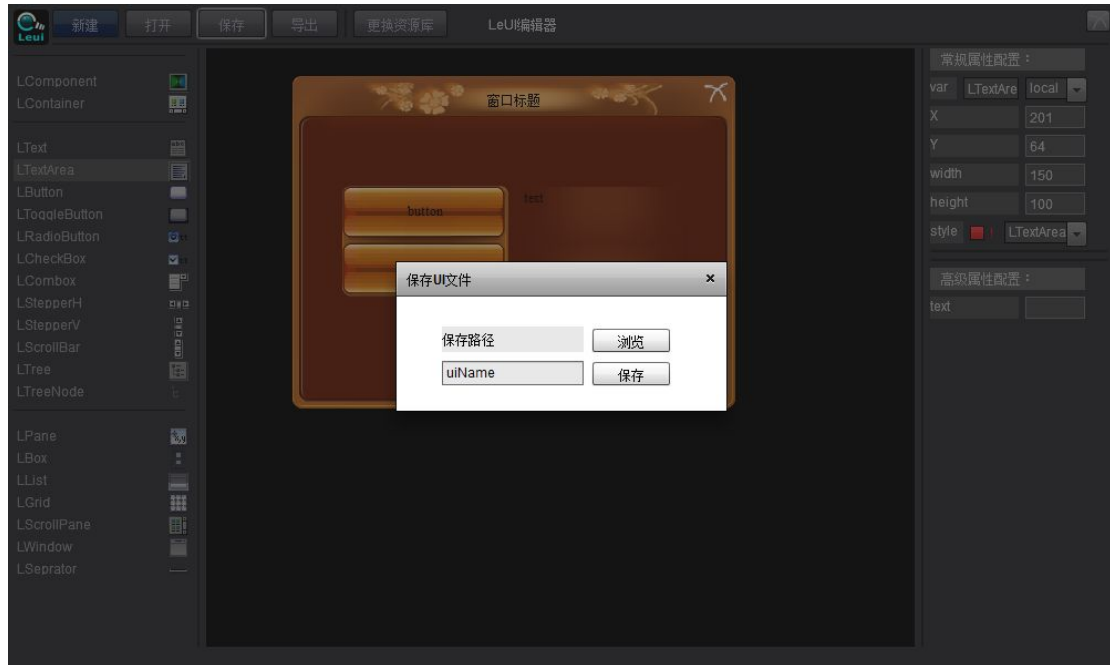
删除组件

如果想删除舞台上的组件实例，选中它，按键盘上的 [Delete](#) 就可以了。
根节点除外：

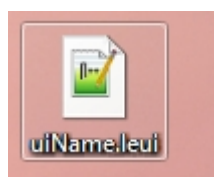


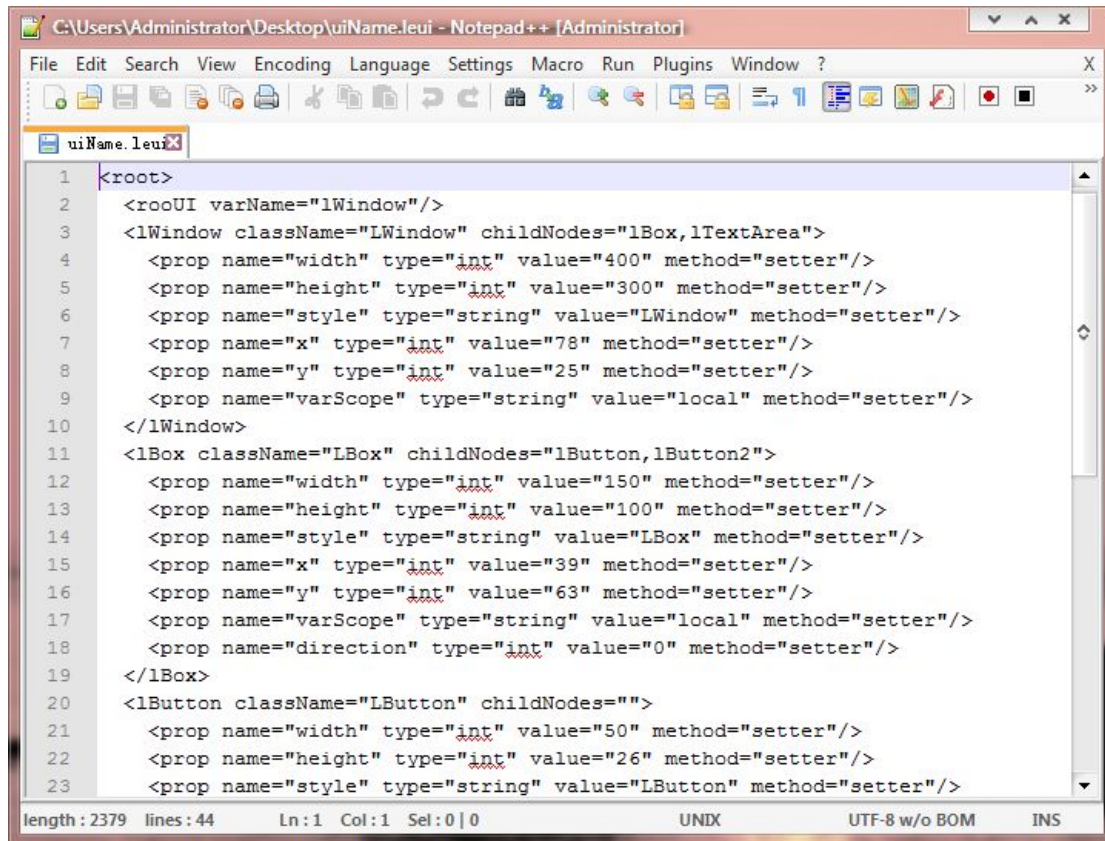
保存 UI

等到编辑完成之后，可以把 UI 文件保存，以便留个纪念（-_-）或下次打开继续编辑。点击菜单栏“保存”按钮，弹出一个对话框，设置路径，点保存就 OK 了：



保存的文件以.leui 为扩展名，内容以 XML 格式记录信息：

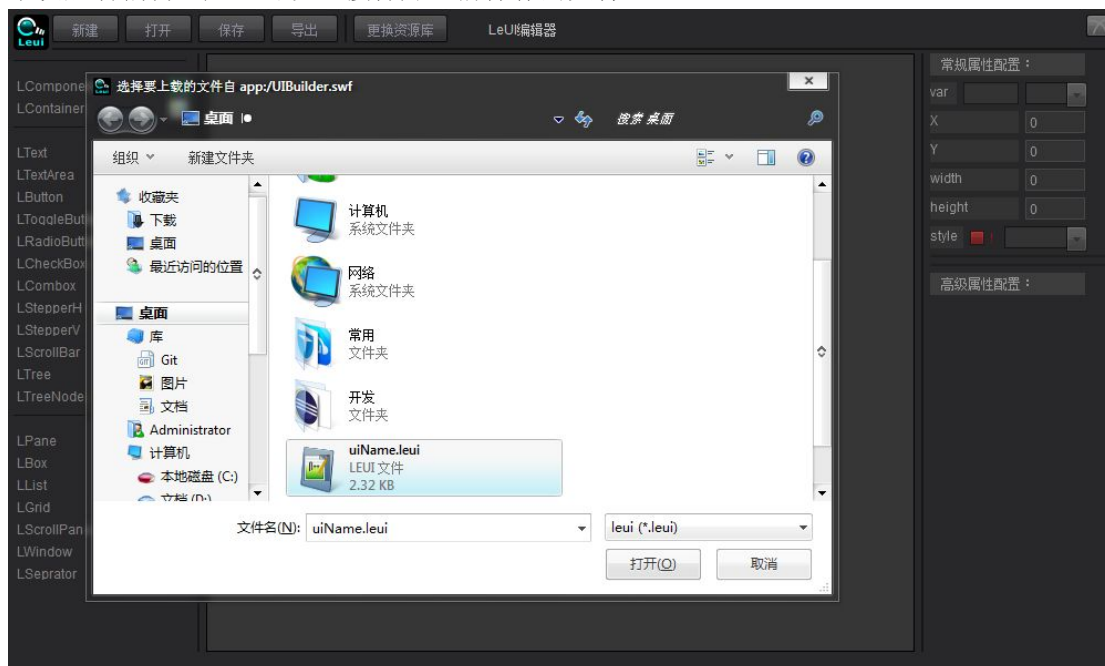




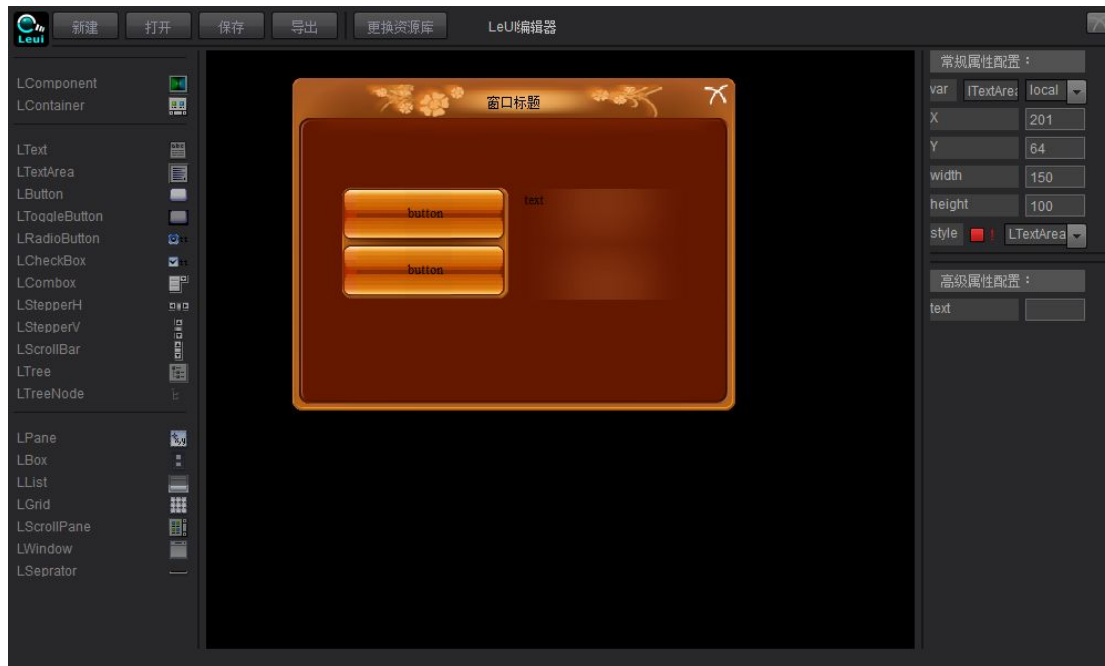
```
1 <root>
2   <rooUI varName="lWindow"/>
3   <lWindow className="LWindow" childNodes="lBox,lTextArea">
4     <prop name="width" type="int" value="400" method="setter"/>
5     <prop name="height" type="int" value="300" method="setter"/>
6     <prop name="style" type="string" value="LWindow" method="setter"/>
7     <prop name="x" type="int" value="78" method="setter"/>
8     <prop name="y" type="int" value="25" method="setter"/>
9     <prop name="varScope" type="string" value="local" method="setter"/>
10  </lWindow>
11  <lBox className="LBox" childNodes="lButton,lButton2">
12    <prop name="width" type="int" value="150" method="setter"/>
13    <prop name="height" type="int" value="100" method="setter"/>
14    <prop name="style" type="string" value="LBox" method="setter"/>
15    <prop name="x" type="int" value="39" method="setter"/>
16    <prop name="y" type="int" value="63" method="setter"/>
17    <prop name="varScope" type="string" value="local" method="setter"/>
18    <prop name="direction" type="int" value="0" method="setter"/>
19  </lBox>
20  <lButton className="LButton" childNodes="">
21    <prop name="width" type="int" value="50" method="setter"/>
22    <prop name="height" type="int" value="26" method="setter"/>
23    <prop name="style" type="string" value="LButton" method="setter"/>
```

打开 UI

下次启动编辑器后，可以直接打开之前保存的文件：

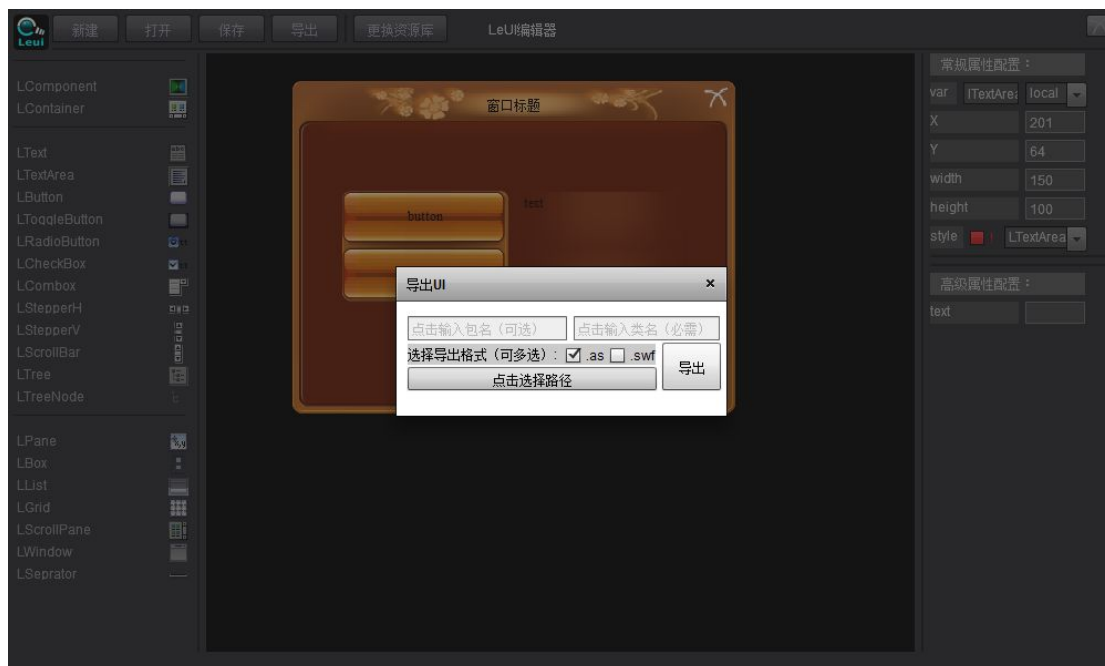


打开后，继续编辑吧：



导出文件

编辑器存在的最重要意义就是导出数据，UI 编辑器自然要能够导出 AS 文件。UI 文件编辑好后，点击菜单栏上的“导出”按钮，会弹出对话框，进行导出前的一些设置：

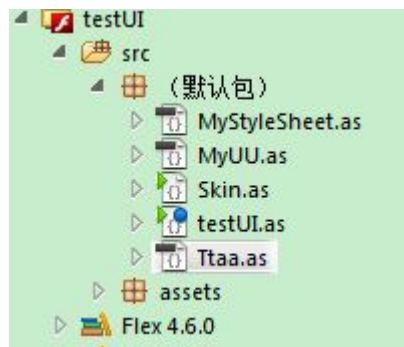


输入包名（可选），如 `org.leui.MyUi`；**输入类名（必需）**，如 `MyXX`；然后选择导出格式，默认导出为 `as` 文件（对于单个 UI 来说，导出为 `swf` 意义不大，所以功能先不提供。请关注后续版本），然后选择导出文件存放的路径，点击“导出”就可以了。

下面示例中，我随意将类名设置为 Ttaa，路径设置为一个测试项目文件夹下。

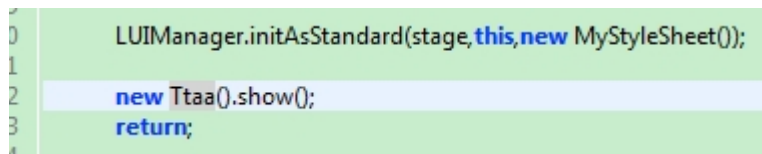


导出后，打开该测试项目：



可以看到，Ttaa.as 已在项目中。

接着，检验一个这个 Ttaa 类是否可用：



运行：



发现这个类，实例化后，跟它在编辑器中的效果是一致的！（注，窗体的标题、按钮的标签，在编辑器中是由编辑器默认提供的，只有当用户为其设置新值后，才会被记录并相应导出到生成的 as 文件中）

至此，编辑器的工作流基本完成，你可以放心地与它一起玩耍了。

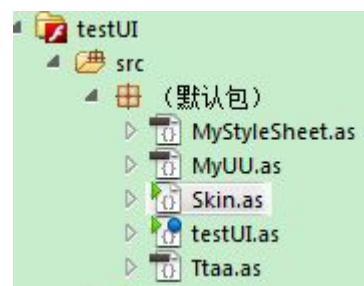
更换资源库

在编辑器中，资源库是指一个包含 LStyleSheet（或其派生类）类声明的 swf 文件。

例如上文第四章 [配置样式库](#) 中提及的示例项目 testUI 中，有一个 MyStyleSheet 类，它里面配置了 LeUI 组件的常用样式信息。

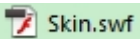
```
19 public class MyStyleSheet extends LStyleSheet
20 {
```

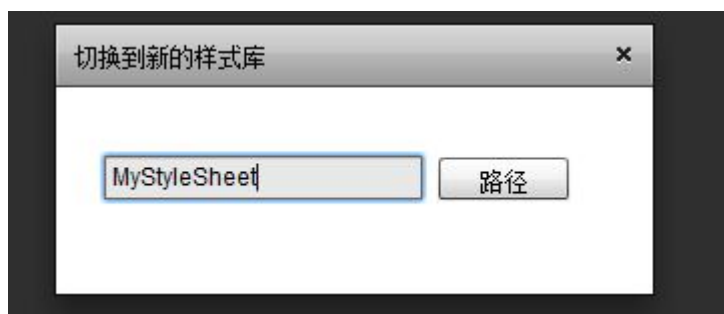
我在 testUI 项目中，添加了一个应用程序类 Skin:




Skin 里，把 MyStyleSheet 的类声明导入进来：

```
1 package
2 {
3     import flash.display.Sprite;
4
5     /**
6      *
7      * @author swellee
8      */
9     public class Skin extends Sprite
10    {
11        //引入样式类
12        private var styleLib:MyStyleSheet;
13        public function Skin()
14        {
15            super();
16        }
17    }
18 }
```

然后把 Skin 编译成  Skin.swf，它里面就含有 MyStyleSheet 的类声明，打开 UI 编辑器，点击菜单栏上的“更换资源库”，在弹出的对话框中输入 MyStyleSheet



然后点击“路径”，找到  Skin.swf，UI 编辑器就会加载 skin.swf，然后查找里面的 MyStyleSheet 类。如果成功找到，就实例化它作为新的样式库，否则，还使用原先可用的那个老样式库。

关于样式库 MyStyleSheet 的讲解，请回到上面的相关章节。

六、小结

此文档涵盖了 LeUI 的架构设计、样式库管理、常用组件的代码示例等内容，能够帮助 LeUI 的使用者快速了解该 UI 框架的思想和用例。未尽之处必然还有不少。更多细节，请阅读源码，或下载 api 文档（<https://github.com/swellee/LeDoc>）

Bug 及建议，欢迎反馈至 svtt@163.com。由于本人纯业余时间开发和维护 LeUI，所以不一定能及时回复，如果愿意，推荐你看源码。

因代码维护更加及时，当文档与源码有出入时，请以代码为准。

另：本文档示例项目所用图片资源，是自己设计的，一个码农的设计肯定谈不上美观，欢迎有兴趣有能力的设计人士提供新的资源，以便更好地展示。设计师的大名将被记录在本文档中以示感谢。