

开始学习

Fairy官网：[官网网址](#)

GitHub源码：[Unity源码网址](#)

本文目录：

开始学习

FairyGUI编辑器的功能

- 编辑器的使用
- fairy项目
- fairy项目所记录的内容
- 导出内容

提供给Unity的SDK

- 文件夹概览
- 主要类图
 - 可交互型组件
 - 控制器的属性控制
 - 控制器的动作
 - 字体
 - 碰撞检测

主要功能

- 启用FairyGUI
 - FairyGUI的启用方式汇总
 - 使用unity编辑器启用FairyGUI
 - 编写代码动态启用FairyGUI
 - StageCamera
 - Stage
 - StageEngine
 - 每帧更新
 - UIContentScaler
- fairy包
 - UIPackage概览
 - 加载fairy包
 - 加载主数据文件
 - 使用UnityEngine.AssetBundle包
 - 直接使用fairy包名
 - 直接使用二进制数据
 - 载入所涉及的资源文件
 - ByteBuffer
 - 概览
 - 解析字符串
 - ZipReader
 - Packageltem
 - type和objectType
 - 包内项目不等同于fairy组件
- 创建fairy元件
 - 同步创建
 - 异步创建
- fairy元件与unity物件的关联
 - 创建UnityEngine.GameObject
 - FairyGUI.DisplayObject创建UnityEngine.GameObject

- FairyGUI.GObject创建UnityEngine.GameObject
 - 在Unity3D编辑器中查看FairyGUI创建的物件的信息
- fairy创建物件汇总
 - 显示物件的分类
- NGraphics
 - MeshFilter和MeshRenderer
 - Material
 - 更改着色器和贴图
 - 1.NGraphics.UpdateManager()
 - 2.创建MaterialManager
 - 3.获取着色器
 - 每帧更新
- Image
 - 纹理贴图的数据流
 - 1.在构建GImage时，设置Image内容
 - 2.加载图片LoadImage()
 - 3.加载图集LoadAtlas()
 - 4.获取UnityEngine.Texture数据
 - 5.创建FairyGUI.NTexture，存储到PackageItem.texture
 - 6.刷新纹理
 - 7.将数据传递给Image
 - 8.将数据传递给NGraphics
- GoWrapper
- 坐标系统
 - 定位点与锚点
 - 坐标转换
 - DisplayObject.LocalToGlobal
 - 源码
 - 解读
 - 坐标转换方法汇总
- 事件机制
 - EventDispatcher 事件分发器
 - EventListener 事件接收器
 - EventBridge 事件桥
 - 通过事件接收器注册事件
 - 通过事件分发器注册事件
 - 抛出事件
 - 冒泡机制
 - 源码
 - 解读规则
 - EventContext 事件内容
- 事件流概览
- 缓动
 - GTween
 - 概览
 - 创建缓动的方法
 - 创建缓动的逻辑
 - TweenManager
 - 概览
 - 主要属性字段
 - 每帧执行
 - GTweener
- 其他类与接口
 - GTweener
 - TweenValue
 - ITweenListener
- 缓动的逻辑算法
 - 算法出处

- 缓动类型
- fairy自带的缓动功能汇总
 - 给组件创建动效
 - 对元件的属性控制启用缓动
 - 代码调用进度条的动态变化
- UE相关
 - 组件
 - 控制器
 - Controller
 - 属性控制
 - 控制器的动作
 - 动效
 - Transition
 - TransitionItem
 - 关联

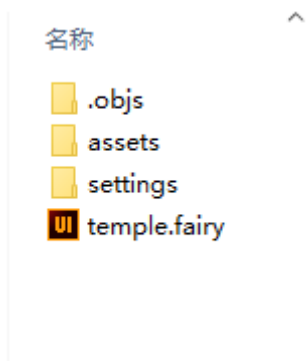
FairyGUI编辑器的功能

编辑器的使用

Fairy编辑器的使用在官网【教程】中已有详细说明。

fairy项目

用户在Fairy编辑器中创作的内容，可以独立记录在Unity项目之外。这个fairy项目存储的文件资源如下：

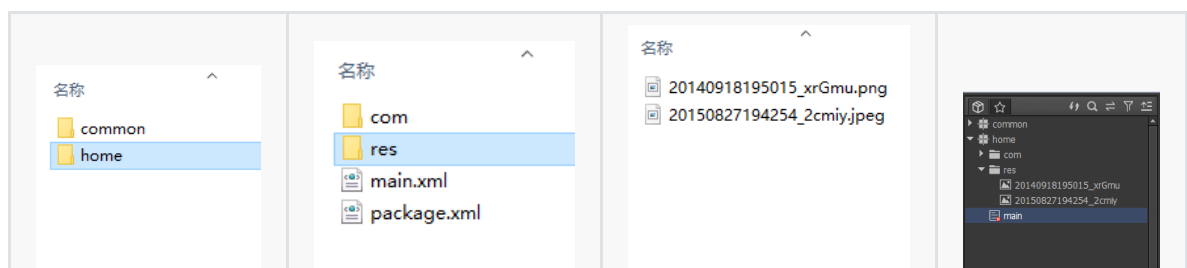


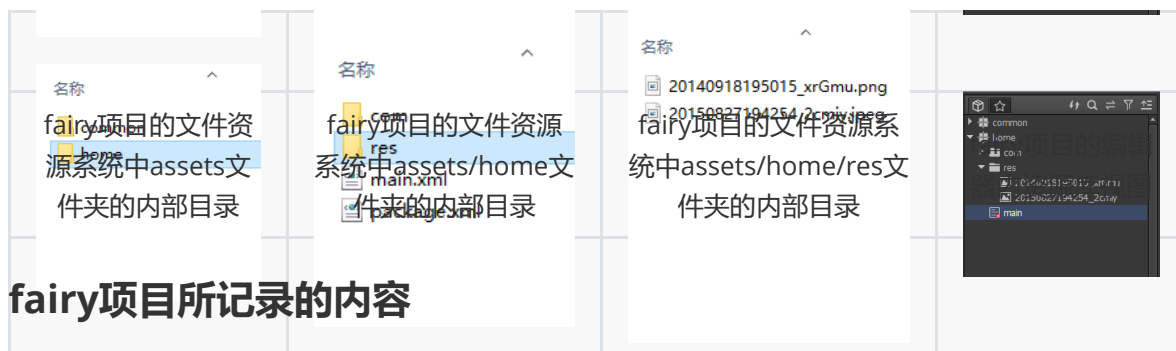
*.fairy文件为编辑器识别的文件，打开项目的入口。

.objs文件夹内为此项目在编辑器中的缓存文件。

settings文件夹内为此项目的偏好设置，这些设置让编辑器的使用更为便捷。

项目内容被放置在assets文件夹中，其内部文件夹对应到fairy项目内部的包。





fairy项目所记录的内容

package.xml文件标识着这个文件夹不是一个普通文件夹，而是fairy项目中的包。

它记录了此包中，包含的所有组件，包括fairy组件和资源组件。

例：

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <packageDescription id="f0o1sc4z" jpegQuality="80" compressPNG="true">
3   <resources>
4     <component id="sooi0" name="main.xml" path="/" exported="true"/>
5     <image id="h1cp2" name="20150827194254_2cmiy.jpeg" path="/res/" />
6     <image id="h1cp3" name="20140918195015_xrGmu.png" path="/res/"
7       scale="9grid" scale9grid="128,192,256,384"/>
8   </resources>
9   <publish name="home"/>
10 </packageDescription>
```

除此之外，其他文件全部对应到fairy编辑器中资源库所显示的组件。

导入到fairy项目内的资源组件，如图片/音频/视频，不做处理而直接保存。

在fairy编辑器中创建的包含资源组件的fairy组件，被记录为xml文件。

这种标识fairy组件的xml文件，记录着，此组件中所有元件的信息。

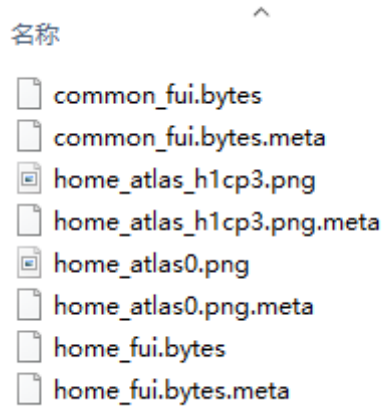
例：

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <component size="640,1136">
3   <displayList>
4     <graph id="n0_sooi" name="n0" xy="0,0" size="640,1136" type="rect"
5       fillColor="#ffccccc">
6       <relation target="" sidePair="center-center,middle-middle"/>
7       <relation target="" sidePair="width-width,height-height"/>
8     </graph>
9     <image id="n5_h1cp" name="n5" src="h1cp3"
10       fileName="res/20140918195015_xrGmu.png" xy="0,175" size="640,960"
11       aspect="true">
12       <relation target="" sidePair="center-center,bottom-bottom"/>
13     </image>
14     <component id="n2_sooi" name="n2" src="sooi1" fileName="btn1.xml"
15       pkg="fvm90flm" xy="528,21">
16       <relation target="" sidePair="right-right,top-top"/>
17     </component>
18     <component id="n3_h1cp" name="n3" src="h1cp1" fileName="btn2.xml"
19       pkg="fvm90flm" xy="190,848">
20       <relation target="" sidePair="center-center,middle-middle"/>
21       <Button title="Start"/>
22     </component>
```

```
18 <text id="n6_jtlz" name="n6" xy="161,266" size="152,43" font="Microsoft
YaHei" fontSize="30" text="第一行文字"/>
19 <text id="n7_jtlz" name="n7" xy="161,334" size="308,82" font="Microsoft
YaHei" fontSize="60" text="第二行文字"/>
20 <text id="n8_jtlz" name="n8" xy="164,451" size="195,98" font="Microsoft
YaHei" fontSize="30" ubb="true" text="第[size=72]三[/size]行文字"/>
21 </displayList>
22 </component>
```

导出内容

fairy项目导出的内容是unity项目可用的资源。

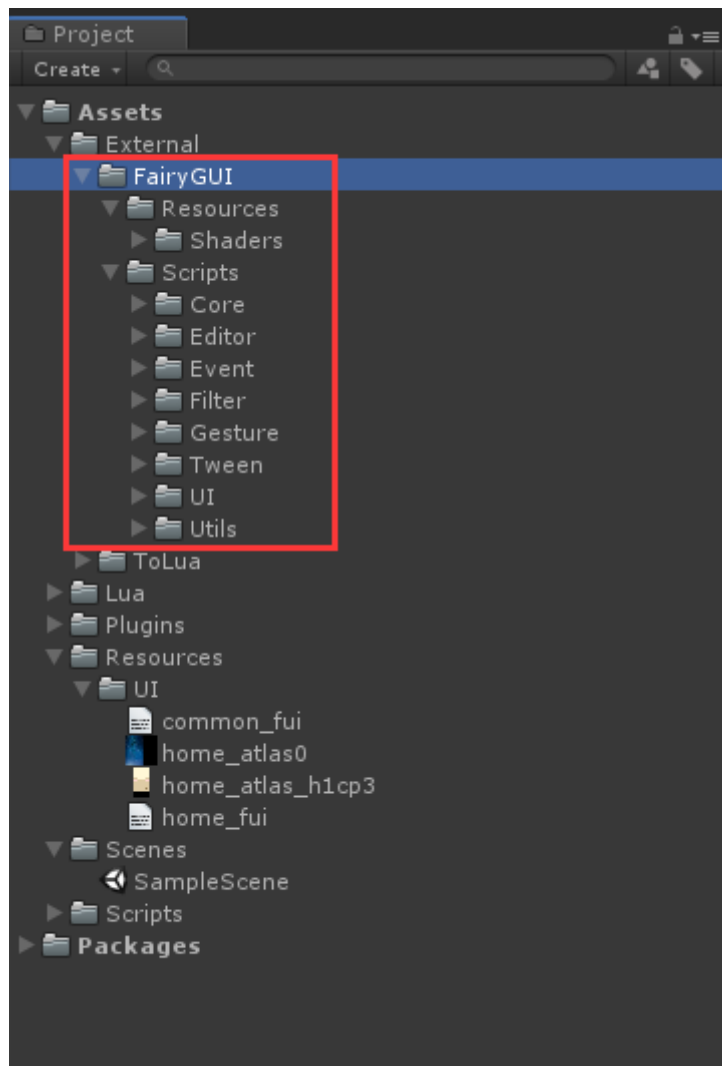


fairy项目中的每个包，会被导出为1个*.bytes文件，和多个图集文件。

提供给Unity的SDK

要在Unity中使用FairyGUI，需要置入fairy相关的两个文件夹。

fairy为多种游戏引擎提供了sdk。在Unity中使用FairyGUI，需要置入的内容：

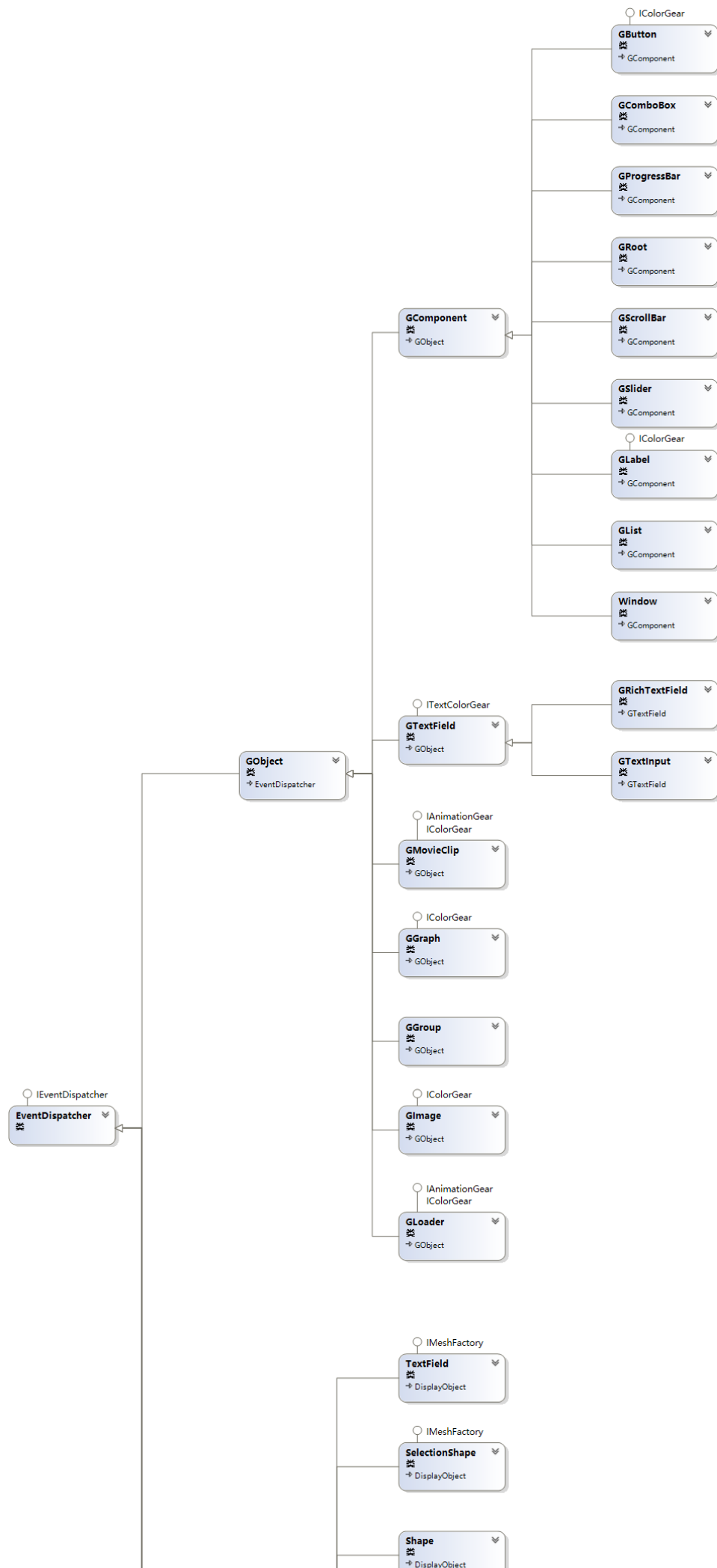


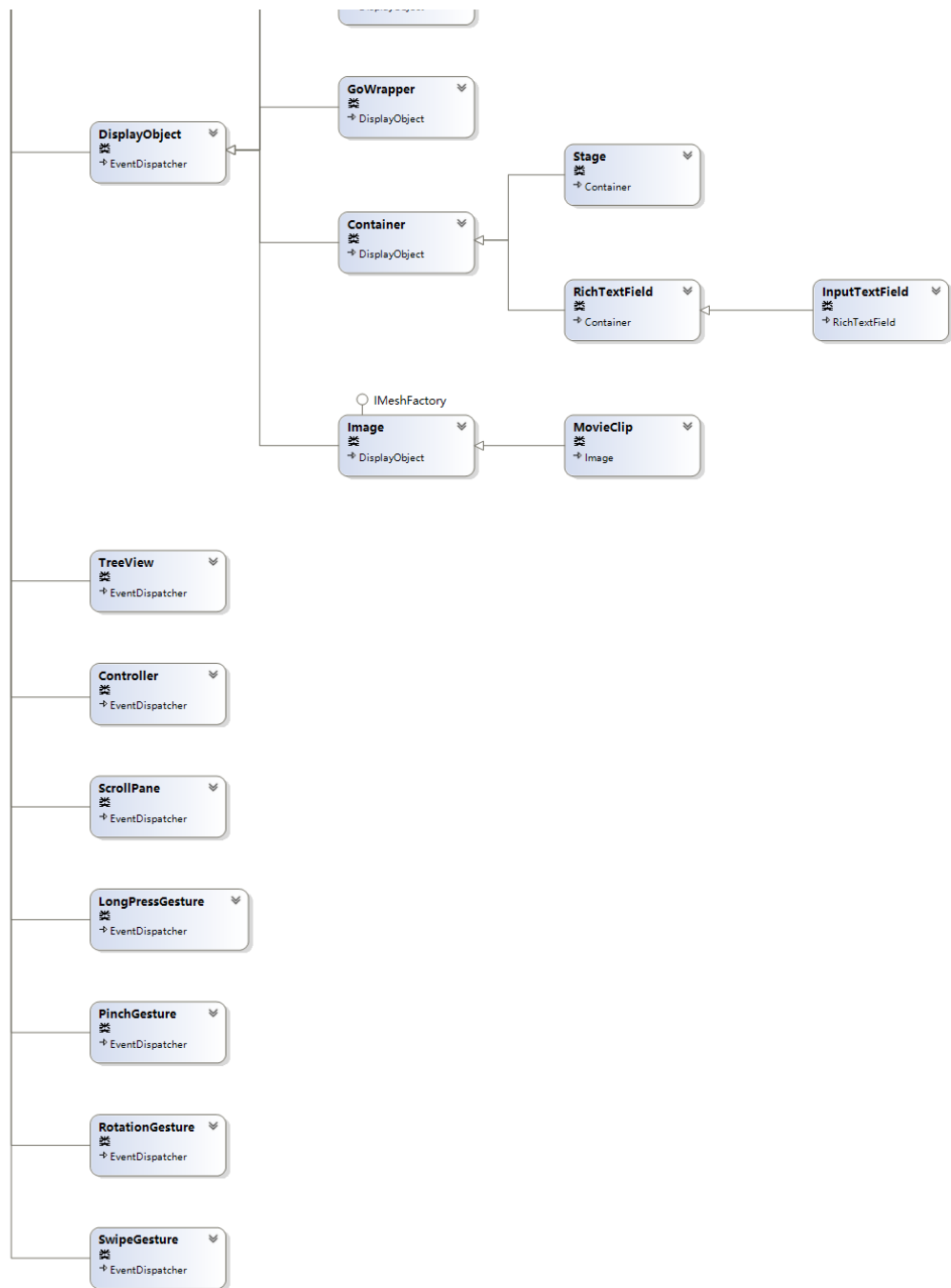
文件夹概览

- **Resources** Unity资源文件夹
 - **Shaders** 着色器
- **Scripts** 代码
 - **UI** Fairy编辑器中用到的组件直接相关
 - **Event** 事件相关
 - **Filter** 滤镜相关
 - **Gesture** 手势相关
 - **Tween** 缓动相关
 - **Core** 其他fairy特色功能
 - **Utils** 为以上代码提供通用静态接口
 - **Editor** Unity编辑器

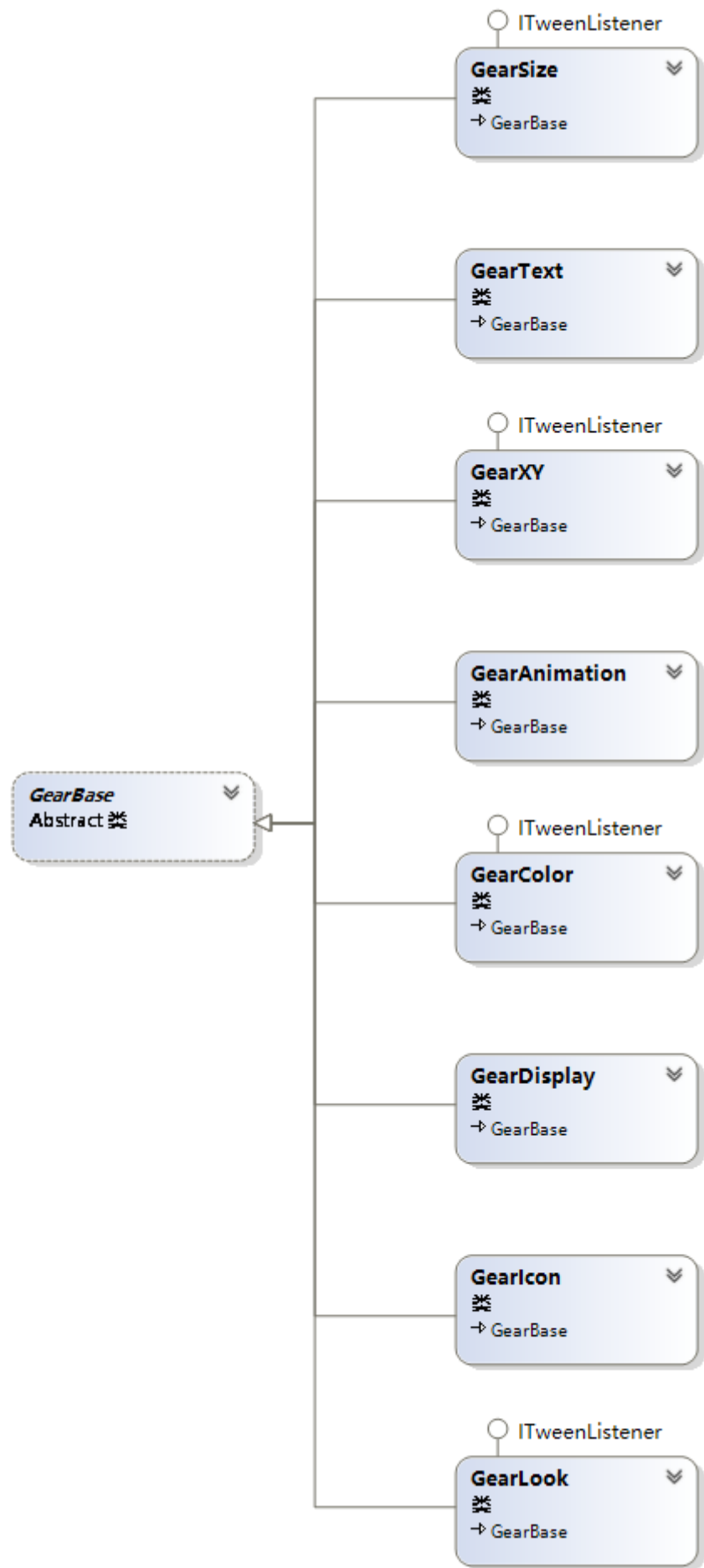
主要类图

可交互型组件

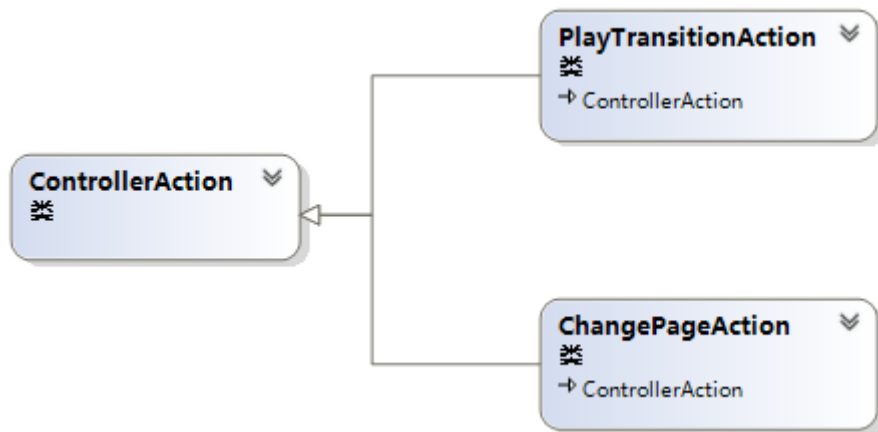




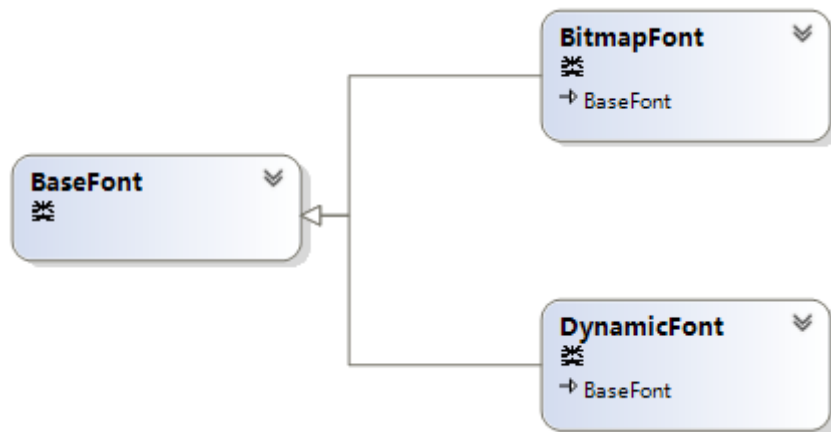
控制器的属性控制



控制器的动作



字体



碰撞检测



主要功能

启用FairyGUI

FairyGUI的启用方式汇总

有两种方法加载fairy包。

使用unity编辑器启用FairyGUI

将fairy提供给unity的sdk源码放入项目之后，unity识别到了其中命名为Editor的文件夹，因此扩展了自身编辑器功能。

在unity编辑器中，新增了目录 **GameObject > FairyGUI > UI Panel / UI Camera** 。

由此，可以在场景中创建fairy内容。

扩展编辑器的相关源码，都在FairyGUIEditor.EditorToolSet类中。

```
1      [MenuItem("GameObject/FairyGUI/UI Panel", false, 0)]
2      static void CreatePanel()
3      {
4  #if !(UNITY_5 || UNITY_5_3_OR_NEWER)
5          EditorApplication.update -= EditorApplication_Update;
6          EditorApplication.update += EditorApplication_Update;
7  #endif
8          StageCamera.CheckMainCamera();
9
10         GameObject panelObject = new GameObject("UIPanel");
11         if (Selection.activeGameObject != null)
12         {
13             panelObject.transform.parent =
14 Selection.activeGameObject.transform;
15             panelObject.layer = Selection.activeGameObject.layer;
16         }
17         else
18         {
19             int layer = LayerMask.NameToLayer(StageCamera.LayerName);
20             panelObject.layer = layer;
21         }
22         panelObject.AddComponent<FairyGUI.UIPanel>();
23         Selection.objects = new Object[] { panelObject };
24     }
```

```
1      [MenuItem("GameObject/FairyGUI/UI Camera", false, 0)]
2      static void CreateCamera()
3      {
4          StageCamera.CheckMainCamera();
5          Selection.objects = new Object[] { StageCamera.main.gameObject };
6      }
```

编写代码动态启用FairyGUI

编写代码动态加载fairy包，它将提供包内的组件和资源，之后再根据需求，创建出其中包含的组件，加入到fairy在unity中创建的根节点。

```
1      UIPackage.AddPackage("UI/common");
2      UIPackage.AddPackage("UI/home");
3      GComponent winHome = UIPackage.CreateObject("home", "main").asCom;
4      GRoot.inst.AddChild(winHome);
```

StageCamera

编写代码动态启用fairy包，在操作根节点FairyGUI.GRoot时，创建了此根节点的单例。

```

1      public static GRoot inst
2      {
3          get
4          {
5              if (_inst == null)
6                  Stage.Instantiate();
7
8              return _inst;
9          }
10     }

```

这里，创建FairyGUI.GRoot单例时，也创建了FairyGUI.Stage的单例。

```

1      public static void Instantiate()
2      {
3          if (_inst == null)
4          {
5              _inst = new Stage();
6              GRoot._inst = new GRoot();
7              GRoot._inst.ApplyContentScaleFactor();
8              _inst.AddChild(GRoot._inst.displayObject);
9
10             StageCamera.CheckMainCamera();
11         }
12     }

```

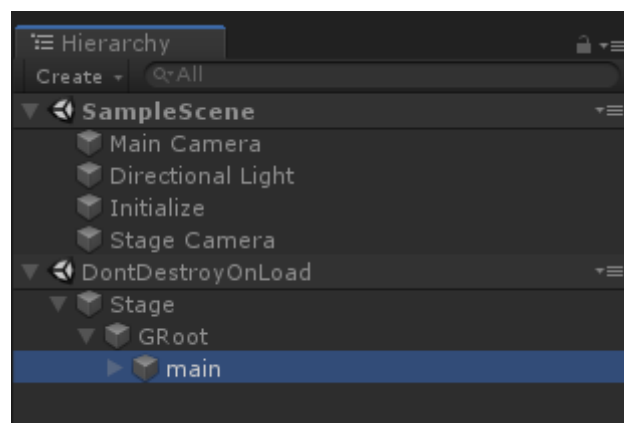
上面几种启用FairyGUI的方式，最终殊途同归，调用了StageCamera.CheckMainCamera方法。

```

1      public static void CheckMainCamera()
2      {
3          if (GameObject.Find(Name) == null)
4          {
5              int layer = LayerMask.NameToLayer(LayerName);
6              CreateCamera(Name, 1 << layer);
7          }
8
9          HitTestContext.cachedMainCamera = Camera.main;
10     }

```

由此，创建了fairy自己的UI相机：Stage Camera。



Stage

使用编写代码动态启用FairyGUI的方式，会创建一个Stage物件，在Unity3D编辑器的Hierarchy视窗中，放在DontDestroyOnLoad场景中。

```
1      public Stage()
2          : base()
3      {
4          _inst = this;
5          soundVolume = 1;
6
7          _updateContext = new UpdateContext();
8          stageWidth = Screen.width;
9          stageHeight = Screen.height;
10         _frameGotHitTarget = -1;
11
12         _touches = new TouchInfo[5];
13         for (int i = 0; i < _touches.Length; i++)
14             _touches[i] = new TouchInfo();
15
16         if (Application.platform == RuntimePlatform.WindowsPlayer
17             || Application.platform == RuntimePlatform.WindowsEditor
18             || Application.platform == RuntimePlatform.OSXPlayer
19             || Application.platform == RuntimePlatform.OSXEditor)
20             touchScreen = false;
21         else
22             touchScreen = Input.touchSupported && SystemInfo.deviceType
23 != DeviceType.Desktop;
24
25         _rollOutChain = new List<DisplayObject>();
26         _rolloverChain = new List<DisplayObject>();
27
28         StageEngine engine = GameObject.FindObjectOfType<StageEngine>
29 ();
30         if (engine != null)
31             Object.Destroy(engine.gameObject);
32
33         this.gameObject.name = "Stage";
34         this.gameObject.layer =
35 LayerMask.NameToLayer(StageCamera.LayerName);
36         this.gameObject.AddComponent<StageEngine>();
37         this.gameObject.AddComponent<UIContentScaler>();
38         this.gameObject.SetActive(true);
39         Object.DontDestroyOnLoad(this.gameObject);
40
41         this.cachedTransform.localScale = new
42 Vector3(StageCamera.UnitsPerPixel, StageCamera.UnitsPerPixel,
43 StageCamera.UnitsPerPixel);
44
45         EnableSound();
46
47         Timers.inst.Add(5, 0, RunTextureCollector);
48
49 #if UNITY_5_4_OR_NEWER
50     SceneManager.sceneLoaded += SceneManager_sceneLoaded;
51 #endif
52
53     _focusRemovedDelegate = OnFocusRemoved;
54 }
```

通过UnityEngine提供的Object.DontDestroyOnLoad方法，将Stage物件设置为不在加载新场景时销毁。

除此之外，使用unity提供了UnityEngine.SceneManagement命名空间下SceneManager.sceneLoaded这个事件，在任一UnityEngine.SceneManagement.Scene场景加载后，执行StageCamera.CheckMainCamera()方法，来确认fairy主相机的存在。

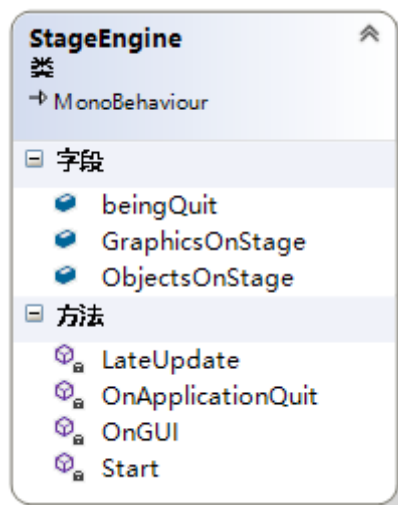
StageEngine

在Stage构造时，给它挂上了一个StageEngine组件。

StageEngine组件可以展示，所有FairyGUI物件的活动状态信息，即FairyGUI.Stats类中记录的静态值。

每帧更新

StageEngine继承自UnityEngine.MonoBehaviour，因而可执行unity开放的多种接口。



所有的更新方法，都放在unity提供的LateUpdate中执行。

```
1      void LateUpdate()
2      {
3          stage.inst.InternalUpdate();
4
5          ObjectsOnStage = Stats.ObjectCount;
6          GraphicsOnStage = Stats.GraphicsCount;
7      }
```

因而执行了Stage.InternalUpdate()方法。

```
1      internal void InternalUpdate()
2      {
3          HandleEvents();
4
5          _updateContext.Begin();
6          Update(_updateContext);
7          _updateContext.End();
8
9          if (DynamicFont.textRebuildFlag)
10         {
11             //字体贴图更改了，重新渲染一遍，防止本帧文字显示错误
12             _updateContext.Begin();
13             Update(_updateContext);
14         }
```

```

14         _updateContext.End();
15
16         DynamicFont.textRebuildFlag = false;
17     }
18 }

```

因而执行了Container.Update()方法，执行了Container的基类DisplayObject的Update()方法，以及此Container的所有子节点DisplayObject的Update()方法。

```

1     override public void Update(UpdateContext context)
2     {
3         if (_isPanel && !gameObject.activeInHierarchy)
4             return;
5
6         base.Update(context);
7
8         if (_cacheAsBitmap && _paintingMode != 0 && _paintingFlag == 2)
9         {
10             if (onUpdate != null)
11                 onUpdate();
12             return;
13         }
14
15         if (_mask != null)
16         {
17             context.EnterClipping(this.id, reversedMask);
18             _mask.graphics._PreUpdateMask(context);
19         }
20         else if (_clipRect != null)
21             context.EnterClipping(this.id,
22 this.TransformRect((Rect)_clipRect, null), clipSoftness);
23
24         float savedAlpha = context.alpha;
25         context.alpha *= this.alpha;
26         bool savedGrayed = context.grayed;
27         context.grayed = context.grayed || this.grayed;
28
29         if (_fBatching)
30             context.batchingDepth++;
31
32         if (context.batchingDepth > 0)
33         {
34             int cnt = _children.Count;
35             for (int i = 0; i < cnt; i++)
36             {
37                 DisplayObject child = _children[i];
38                 if (child.visible)
39                     child.Update(context);
40             }
41         }
42         else
43         {
44             if (_mask != null)
45                 _mask.renderingOrder = context.renderingOrder++;
46
47             int cnt = _children.Count;
48             for (int i = 0; i < cnt; i++)

```

```

48         {
49             DisplayObject child = _children[i];
50             if (child.visible)
51             {
52                 if (child != _mask)
53                     child.renderingOrder =
context.renderingOrder++;
54
55                 child.Update(context);
56             }
57         }
58
59         if (_mask != null)
60
_mask.graphics._SetStencilEraserOrder(context.renderingOrder++);
61     }
62
63     if (_fBatching)
64     {
65         if (context.batchingDepth == 1)
66             SetRenderingOrder(context);
67         context.batchingDepth--;
68     }
69
70     context.alpha = savedAlpha;
71     context.grayed = savedGrayed;
72
73     if (_clipRect != null || _mask != null)
74         context.LeaveClipping();
75
76     if (_paintingMode > 0 && paintingGraphics.texture != null)
77         UpdateContext.OnEnd += _captureDelegate;
78
79     if (onUpdate != null)
80         onUpdate();
81 }

```

UIContentScaler

一般都需要将这个组件，手动挂到unity场景中，或者在业务层代码中自行添加，来设置适配模式。



相关编辑器扩展的源码，在FairyGUIEditor.UIContentScalerEditor。

fairy包

UIPackage概览

UIPackage的全部内容:

1. **管理所有已加载的fairy包**

字段/属性/方法	用途
static List<UIPackage> _packageList static Dictionary<string, UIPackage> _packageInstById static Dictionary<string, UIPackage> _packageInstByName	运行时所有已加载fairy包
internal static int _constructing	记录正在构造中的fairy元件的数量
public static UIPackage GetById(string id) public static UIPackage GetByName(string name) public static List<UIPackage> GetPackages()	获取已加载的fairy包
public static UIPackage AddPackage(AssetBundle bundle) public static UIPackage AddPackage(AssetBundle desc, AssetBundle res) public static UIPackage AddPackage(AssetBundle desc, AssetBundle res, string mainAssetName) public static UIPackage AddPackage(string descFilePath) public static UIPackage AddPackage(string assetPath, LoadResource loadFunc) public static UIPackage AddPackage(byte[] descData, string assetNamePrefix, LoadResource loadFunc)	加载fairy包
public static void RemovePackage(string packageIdOrName) public static void RemoveAllPackages()	卸载fairy包
public static GObject CreateObject(string pkgName, string resName) public static GObject CreateObject(string pkgName, string resName, System.Type userClass) public static GObject CreateObjectFromURL(string url) public static GObject CreateObjectFromURL(string url, System.Type userClass) public static void CreateObjectAsync(string pkgName, string resName, CreateObjectCallback callback) public static void CreateObjectFromURL(string url, CreateObjectCallback callback)	创建指定包内的指定项目
public static object GetItemAsset(string pkgName, string resName) public static object GetItemAssetByURL(string url) public static PackageItem GetItemByURL(string url) static int ComparePackageItem(PackageItem p1, PackageItem p2)	获取指定包内的指定项目
public static string GetItemURL(string pkgName, string resName) public static string NormalizeURL(string url)	包内资源的路径字符串形式转换
public static void SetStringsSource(XML source)	加载XML格式的动效文件 (无引用)

2. 对应一个fairy包

字段/属性/方法	用途
public string id	此包的唯一ID
public string name	包名
public string assetPath	此包的资源路径
List<PackageItem> _items Dictionary<string, PackageItem> _itemsByName Dictionary<string, PackageItem> _itemsByName	此包的所有包内项目
Dictionary<string, string>[] _dependencies public Dictionary<string, string>[] dependencies	此包的所有依赖项
AssetBundle _resBundle public AssetBundle resBundle bool _fromBundle	此包的ab资源
string _customId public string customId	自定义替换id值
LoadResource _loadFun	加载此包后的回调委托
Dictionary<string, AtlasSprite> _sprites	此包所涉及的所有图集资源
public UIPackage()	此包的构造函数
bool LoadPackage(ByteBuffer buffer, string packageSource, string assetNamePrefix)	加载此包
public void LoadAllAssets() public void ReloadAssets() public void ReloadAssets(AssetBundle resBundle) void LoadAtlas(PackageItem item) void LoadImage(PackageItem item) void LoadSound(PackageItem item) byte[] LoadBinary(PackageItem item) void LoadMovieClip(PackageItem item) void LoadFont(PackageItem item)	加载此包内的项目数据和资源
public void UnloadAssets()	卸载此包的数据
void Dispose()	销毁此包
public GObject CreateObject(string resName) public GObject CreateObject(string resName, System.Type userClass) public void CreateObjectAsync(string resName, CreateObjectCallback callback) GObject CreateObject(PackageItem item, System.Type userClass)	创建此包内的项目

字段/属性/方法	用途
<pre>public object GetItemAsset(string resName) public List<PackageItem> GetItems() public PackageItem GetItem(string itemId) public PackageItem GetItemByName(string itemName) public object GetItemAsset(PackageItem item)</pre>	获取此包内的项目

3. 内嵌内容

类型	内嵌内容	描述
公开静态常量	<code>public const string URL_PREFIX</code>	包内资源路径字符串的前缀
委托	<code>public delegate object LoadResource(string name, string extension, System.Type type, out DestroyMethod destroyMethod);</code>	fairy包加载完成后的回调
委托	<code>public delegate void CreateObjectCallback(GObject result);</code>	fairy元件创建完成后的回调
内嵌私有类	<code>class AtlasSprite</code>	图集

加载fairy包

使用全局的UIPackage.AddPackage()方法，可以加载整个fairy包内的所有项目。

在全局的UIPackage.AddPackage()的多种重载方法中，加载了此fairy包的主数据文件，并解析；

然后，创建了一个UIPackage类的对象，调用此对象私有的LoadPackage()方法，遍历主数据文件的内容，加载其中所有涉及到的资源。

加载主数据文件

对于全局的UIPackage.AddPackage()方法，fairy提供了多种传参类型，根据传参类型可以将这些方法分为以下3类。

1. 使用UnityEngine.AssetBundle包
2. 直接使用fairy包名
3. 直接使用二进制数据

使用UnityEngine.AssetBundle包

fairy所导出的所有文件，被打成一个AssetBundle包，作为desc参数，传进来。

此fairy包所使用的外部资源文件，被导出一个AssetBundle包，作为res参数，传进来。如果这个参数被缺省，则使用desc；如果这个参数不同于desc，则只保留desc的主数据内容，而弃用desc。

mainAssetName为主数据的文件名。如果缺省，则遍历所有desc内容，去查找到它。

```
1 public static UIPackage AddPackage(AssetBundle desc, AssetBundle
   res, string mainAssetName)
```

```

2      {
3          byte[] source = null;
4      #if (UNITY_5 || UNITY_5_3_OR_NEWER)
5          if (mainAssetName != null)
6          {
7              TextAsset ta = desc.LoadAsset<TextAsset>(mainAssetName);
8              if (ta != null)
9                  source = ta.bytes;
10         }
11         else
12         {
13             string[] names = desc.GetAllAssetNames();
14             string searchPattern = "_fui";
15             foreach (string n in names)
16             {
17                 if (n.IndexOf(searchPattern) != -1)
18                 {
19                     TextAsset ta = desc.LoadAsset<TextAsset>(n);
20                     if (ta != null)
21                     {
22                         source = ta.bytes;
23                         mainAssetName =
24 Path.GetFileNameWithoutExtension(n);
25                         break;
26                     }
27                 }
28             }
29         #else
30             if (mainAssetName != null)
31             {
32                 TextAsset ta = (TextAsset)desc.Load(mainAssetName,
33 typeof(TextAsset));
34                 if (ta != null)
35                     source = ta.bytes;
36             }
37             else
38             {
39                 source = ((TextAsset)desc.mainAsset).bytes;
40                 mainAssetName = desc.mainAsset.name;
41             }
42         #endif
43         if (source == null)
44             throw new Exception("FairyGUI: no package found in this
45 bundle.");
46
47         if (desc != res)
48             desc.Unload(true);
49
50         ByteBuffer buffer = new ByteBuffer(source);
51
52         UIPackage pkg = new UIPackage();
53         pkg._resBundle = res;
54         pkg._fromBundle = true;
55         int pos = mainAssetName.IndexOf("_fui");
56         string assetNamePrefix;
57         if (pos != -1)
58             assetNamePrefix = mainAssetName.Substring(0, pos);

```

```

57         else
58             assetNamePrefix = mainAssetName;
59         if (!pkg.LoadPackage(buffer, res.name, assetNamePrefix))
60             return null;
61
62         _packageInstById[pkg.id] = pkg;
63         _packageInstByName[pkg.name] = pkg;
64         _packageList.Add(pkg);
65
66         return pkg;
67     }

```

将desc这个AssetBundle文件解压后，就得到了fairy的所有导出文件。

fairy的导出文件，包含一个名为 `包名_fui.bytes` 的二进制文件（主数据），和多个资源文件（某些资源文件也可能为bytes类型）。

这个主数据文件，为解析fairy包的入口，作为mainAssetName参数传入；如果没有传入此参数，将遍历所有的AssetBundle解压文件（即fairy导出文件），直到找到包含 `_fui` 字符串的那个文件，记录下它的名称。

使用UnityEngine的AssetBundle.LoadAsset<TextAsset>()方法，来解析这个主数据bytes文件，得到了它的UnityEngine.TextAsset数据。

根据它的数据内容 ([byte[]] TextAsset.bytes)，能创建一个ByteBuffer类的对象。

如果desc这个AssetBundle不同于res，则把desc整个AssetBundle包全部卸载掉。（但能留下desc的主数据内容，被记录在了ByteBuffer类的对象中。）

根据res这个AssetBundle，创建一个新的UIPackage类的对象。

然后使用这个UIPackage对象的LoadPackage()方法加载此包。这里传入的参数依次为：以desc主数据文件为数据创建的ByteBuffer类的对象，res的文件名，desc中的包名。

最后将此包记录在此管理类的已加载包列表字段中。

直接使用fairy包名

可以直接传入此fairy包的包名（包含了相对路径），来加载fairy包。

```

1     public static UIPackage AddPackage(string descFilePath)
2     {
3         if (descFilePath.StartsWith("Assets/"))
4         {
5             #if UNITY_EDITOR
6                 return AddPackage(descFilePath, (string name, string
extension, System.Type type, out DestroyMethod destroyMethod) =>
7                 {
8                     destroyMethod = DestroyMethod.Unload;
9                     return AssetDatabase.LoadAssetAtPath(name + extension,
type);
10                });
11            #else
12
13                Debug.LogWarning("FairyGUI: failed to load package in '" +
descFilePath + "'");
14                return null;
15            #endif
16        }

```

```

17         return AddPackage(descFilePath, (string name, string extension,
System.Type type, out DestroyMethod destroyMethod) =>
18         {
19             destroyMethod = DestroyMethod.Unload;
20             return Resources.Load(name, type);
21         });
22     }
23
24     public static UIPackage AddPackage(string assetPath, LoadResource
loadFunc)
25     {
26         if (_packageInstById.ContainsKey(assetPath))
27             return _packageInstById[assetPath];
28
29         DestroyMethod dm;
30         TextAsset asset = (TextAsset)loadFunc(assetPath + "_fui",
".bytes", typeof(TextAsset), out dm);
31         if (asset == null)
32         {
33             if (Application.isPlaying)
34                 throw new Exception("FairyGUI: Cannot load ui package
in '" + assetPath + "'");
35             else
36                 Debug.LogWarning("FairyGUI: Cannot load ui package in
'" + assetPath + "'");
37         }
38
39         ByteBuffer buffer = new ByteBuffer(asset.bytes);
40
41         UIPackage pkg = new UIPackage();
42         pkg._loadFunc = loadFunc;
43         pkg.assetPath = assetPath;
44         if (!pkg.LoadPackage(buffer, assetPath, assetPath))
45             return null;
46
47         _packageInstById[pkg.id] = pkg;
48         _packageInstByName[pkg.name] = pkg;
49         _packageInstById[assetPath] = pkg;
50         _packageList.Add(pkg);
51         return pkg;
52     }

```

根据fairy包名，可找到此fairy包的主数据文件。

使用传入的LoadResource委托loadFunc参数，来解析这个主数据文件；如果此参数被缺省，则默认使用UnityEditor的AssetDatabase.LoadAssetAtPath()静态方法，来解析此bytes文件。

将解析结果（object类型），强制转换为UnityEngine.TextAsset类型。

一如既往。根据此TextAsset数据的内容，创建一个ByteBuffer类的对象。

根据包名，创建一个新的UIPackage类的对象。

然后使用这个UIPackage对象的LoadPackage()方法加载此包。这里传入的参数依次为：以此包的主数据文件为数据创建的ByteBuffer类的对象，包名，包名。

最后将此包记录在此管理类的已加载包列表字段中。

直接使用二进制数据

可以直接传入二进制数据，将其加载为fairy包的形式。

```
1      public static UIPackage AddPackage(byte[] descData, string
    assetNamePrefix, LoadResource loadFunc)
2      {
3          ByteBuffer buffer = new ByteBuffer(descData);
4
5          UIPackage pkg = new UIPackage();
6          pkg._loadFunc = loadFunc;
7          if (!pkg.LoadPackage(buffer, "raw data", assetNamePrefix))
8              return null;
9
10         _packageInstById[pkg.id] = pkg;
11         _packageInstByName[pkg.name] = pkg;
12         _packageList.Add(pkg);
13
14         return pkg;
15     }
```

根据传入的二进制数据，创建一个ByteBuffer类的对象。

创建一个新的UIPackage类的对象。

使用这个UIPackage对象的LoadPackage()方法加载此包。这里传入的参数依次为：以传入二进制数据为数据创建的ByteBuffer类的对象，统一字符 raw data，传入的前缀名。

将此包记录在此管理类的已加载包列表字段中。

载入所涉及的资源文件

```
1      bool LoadPackage(ByteBuffer buffer, string packageSource, string
    assetNamePrefix)
2      {
3          if (buffer.ReadUInt() != 0x46475549)
4          {
5              if (Application.isPlaying)
6                  throw new Exception("FairyGUI: old package format
    found in '" + packageSource + "'");
7              else
8              {
9                  Debug.LogWarning("FairyGUI: old package format found
    in '" + packageSource + "'");
10                 return false;
11             }
12         }
13
14         buffer.version = buffer.ReadInt();
15         buffer.ReadBool(); //compressed
16         id = buffer.ReadString();
17         name = buffer.ReadString();
18         if (_packageInstById.ContainsKey(id) && name !=
    _packageInstById[id].name)
19         {
20             Debug.LogWarning("FairyGUI: Package id conflicts, '" +
    name + "' and '" + _packageInstById[id].name + "'");
21             return false;
22         }
```



```

23     buffer.Skip(20);
24     int indexTablePos = buffer.position;
25     int cnt;
26
27     buffer.Seek(indexTablePos, 4);
28
29     cnt = buffer.ReadInt();
30     string[] stringTable = new string[cnt];
31     for (int i = 0; i < cnt; i++)
32         stringTable[i] = buffer.ReadString();
33     buffer.stringTable = stringTable;
34
35     if (buffer.Seek(indexTablePos, 5))
36     {
37         cnt = buffer.ReadInt();
38         for (int i = 0; i < cnt; i++)
39         {
40             int index = buffer.ReadUshort();
41             int len = buffer.ReadInt();
42             stringTable[index] = buffer.ReadString(len);
43         }
44     }
45
46     buffer.Seek(indexTablePos, 1);
47
48     PackageItem pi;
49
50     if (assetNamePrefix == null)
51         assetNamePrefix = string.Empty;
52     else if (assetNamePrefix.Length > 0)
53         assetNamePrefix = assetNamePrefix + "_";
54
55     cnt = buffer.ReadShort();
56     for (int i = 0; i < cnt; i++)
57     {
58         int nextPos = buffer.ReadInt();
59         nextPos += buffer.position;
60
61         pi = new PackageItem();
62         pi.owner = this;
63         pi.type = (PackageItemType)buffer.ReadByte();
64         pi.id = buffer.ReadS();
65         pi.name = buffer.ReadS();
66         buffer.ReadS(); //path
67         pi.file = buffer.ReadS();
68         pi.exported = buffer.ReadBool();
69         pi.width = buffer.ReadInt();
70         pi.height = buffer.ReadInt();
71
72         switch (pi.type)
73         {
74             case PackageItemType.Image:
75             {
76                 pi.objectType = ObjectType.Image;
77                 int scaleOption = buffer.ReadByte();
78                 if (scaleOption == 1)
79                 {
80                     Rect rect = new Rect();

```

```

81         rect.x = buffer.ReadInt();
82         rect.y = buffer.ReadInt();
83         rect.width = buffer.ReadInt();
84         rect.height = buffer.ReadInt();
85         pi.scale9Grid = rect;
86
87         pi.tileGridIndice = buffer.ReadInt();
88     }
89     else if (scaleOption == 2)
90         pi.scaleByTile = true;
91
92     buffer.ReadBool(); //smoothing
93     break;
94 }
95
96 case PackageItemType.MovieClip:
97 {
98     buffer.ReadBool(); //smoothing
99     pi.objectType = ObjectType.MovieClip;
100    pi.rawData = buffer.ReadBuffer();
101    break;
102 }
103
104 case PackageItemType.Font:
105 {
106     pi.rawData = buffer.ReadBuffer();
107     break;
108 }
109
110 case PackageItemType.Component:
111 {
112     int extension = buffer.ReadByte();
113     if (extension > 0)
114         pi.objectType = (ObjectType)extension;
115     else
116         pi.objectType = ObjectType.Component;
117     pi.rawData = buffer.ReadBuffer();
118
119     UIObjectFactory.ResolvePackageItemExtension(pi);
120     break;
121 }
122
123 case PackageItemType.Atlas:
124 case PackageItemType.Sound:
125 case PackageItemType.Misc:
126 {
127     pi.file = assetNamePrefix + pi.file;
128     break;
129 }
130 }
131 _items.Add(pi);
132 _itemsById[pi.id] = pi;
133 if (pi.name != null)
134     _itemsByName[pi.name] = pi;
135
136 buffer.position = nextPos;
137 }

```

```

138
139         buffer.Seek(indexTablePos, 2);
140
141         cnt = buffer.ReadShort();
142         for (int i = 0; i < cnt; i++)
143         {
144             int nextPos = buffer.ReadShort();
145             nextPos += buffer.position;
146
147             string itemId = buffer.ReadS();
148             pi = _itemsById[buffer.ReadS()];
149
150             AtlasSprite sprite = new AtlasSprite();
151             sprite.atlas = pi;
152             sprite.rect.x = buffer.ReadInt();
153             sprite.rect.y = buffer.ReadInt();
154             sprite.rect.width = buffer.ReadInt();
155             sprite.rect.height = buffer.ReadInt();
156             sprite.rotated = buffer.ReadBool();
157             _sprites[itemId] = sprite;
158
159             buffer.position = nextPos;
160         }
161
162         if (buffer.Seek(indexTablePos, 3))
163         {
164             cnt = buffer.ReadShort();
165             for (int i = 0; i < cnt; i++)
166             {
167                 int nextPos = buffer.ReadInt();
168                 nextPos += buffer.position;
169
170                 if (_itemsById.TryGetValue(buffer.ReadS(), out pi))
171                 {
172                     if (pi.type == PackageItemType.Image)
173                     {
174                         pi.pixelHitTestData = new PixelHitTestData();
175                         pi.pixelHitTestData.Load(buffer);
176                     }
177                 }
178
179                 buffer.position = nextPos;
180             }
181         }
182
183         if (!Application.isPlaying)
184             _items.Sort(ComparePackageItem);
185
186         buffer.Seek(indexTablePos, 0);
187         cnt = buffer.ReadShort();
188         _dependencies = new Dictionary<string, string>[cnt];
189         for (int i = 0; i < cnt; i++)
190         {
191             Dictionary<string, string> kv = new Dictionary<string,
string>();
192             kv.Add("id", buffer.ReadS());
193             kv.Add("name", buffer.ReadS());
194             _dependencies[i] = kv;

```

```

195         }
196
197         return true;
198     }

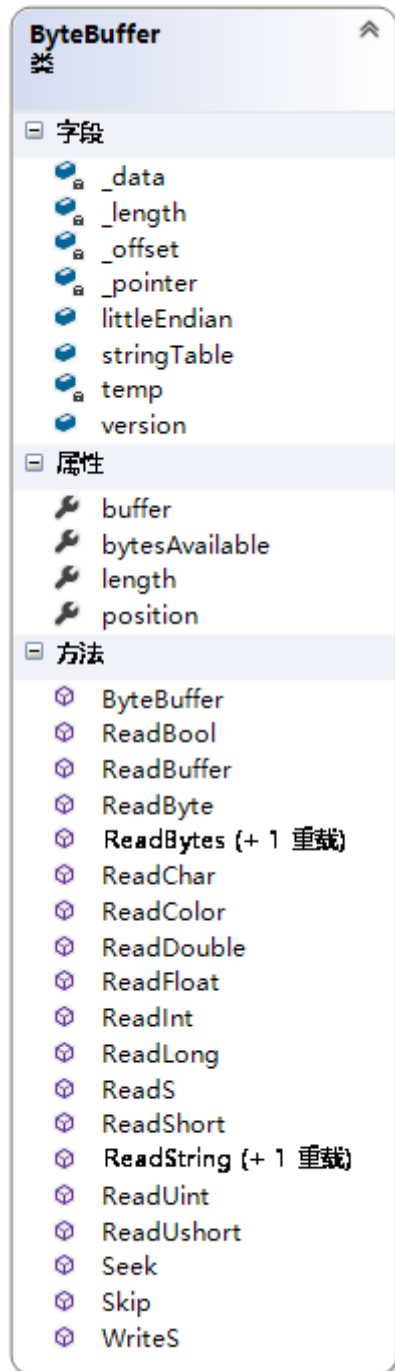
```

解析传入的ByteBuffer数据，按类型创建所有包内项目。

ByteBuffer

概览

FairyGUI.Utls.ByteBuffer用于解析fairy包的主数据二进制文件。



公开的position属性和私有的_pointer字段一致，为记录当前解析位置的int值。

每解析几个byte字节，就将当前解析位置移动几个字节，下次再从当前位置开始解析。

使用多种不同的Read()方法，从当前解析字节位置开始，在要求类型的字节长度内，将这些byte字节解析为所要求类型的数据。

使用Skip()方法，可以跳过几个byte字节，不解析，而只移动当前解析位置。

使用Seek()方法，可以根据指定索引的byte值，跳过几个字节。

解析字符串

解析字符串有ReadString()和ReadS()两种方法。

加载fairy包时，第一次解析，找出其中所有的字符串，使用ReadString()，存入了[string[]]stringTable字段中；

之后需要解析字符串时，可以使用ReadS()方法，直接从此数组中取值，读取当前解析字节位置开始的字符串。

需要修改字符串内容时，使用WriteS()方法，将新的数据直接修改存入stringTable字段中，之后即可使用ReadS()方法读取到新数据。

ZipReader

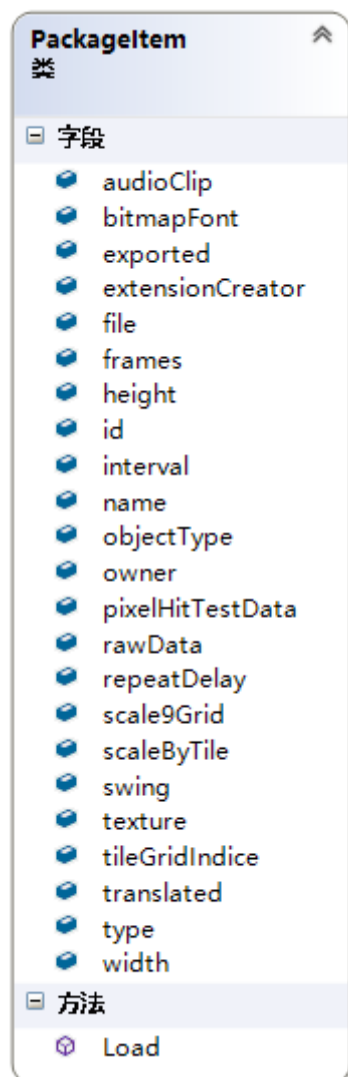
ByteBuffer.littleEndian字段记录了_data数据是否来自zip文件。

如果不是来自zip文件，则解析字节时，需将数据倒序处理；

如果来自未解压的zip文件，则解析字节时无需倒序处理（目前无引用）。




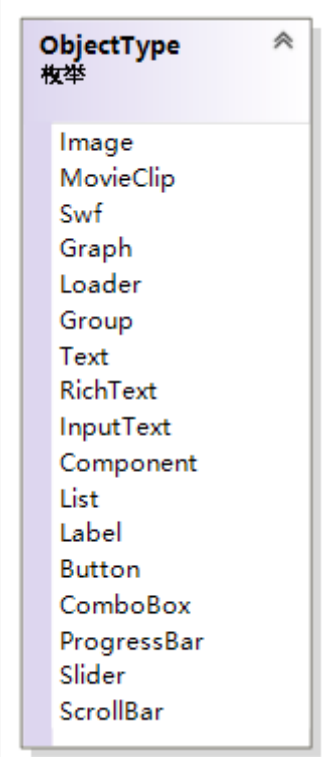
Packageltem

一个Packageltem对象就是一个包内项目。



type和objectType

Packageltem的type字段和objectType字段的比较。

type	objectType
<div> (字段) PackageltemType Packageltem.type</div> <div></div>	<div> (字段) ObjectType Packageltem.objectType</div> <div></div>

这两个字段都是在UIPackage类的对象的LoadPackage()方法执行时被赋值。

type字段为解析ByteBuffer数据直接得到的值。

objectType字段是在type字段被赋值后，根据其值的不同，而在其分支中被赋值；

它可能为空值，所以不能作为分类依据，而只能作为一个标识。

包内项目不等同于fairy组件

一个包内项目Packageltem，并不等同于一个元件。

它跟fairy数据存储规则相关。它的数据来自于ByteBuffer的解析结果。

一个元件可能会被解析为多个包内项目。比如，一个图片元件，它会被解析为，一个type为Image的Packageltem，和一个type为Atlas（不重复）的Packageltem。

如果一个fairy组件，在fairy编辑器中没有被设置为导出，且没有设置为导出的组件引用它，那么解析结果中，将不包含此组件的相关信息。

创建fairy元件

在包加载完成后，可以创建fairy元件。

如果fairy包没有加载，则会报错提示。

创建fairy元件的方法很多，按它们最终执行的方法，分为两类：

1. 同步创建

2. 异步创建

同步创建

所有同步创建fairy元件的方法，最终都需调用下面这个UIPackage类的对象的CreateObject()方法。

```
1      GObject CreateObject(PackageItem item, System.Type userClass)
2      {
3          Stats.LatestObjectCreation = 0;
4          Stats.LatestGraphicsCreation = 0;
5
6          GetItemAsset(item);
7
8          GObject g = null;
9          if (item.type == PackageItemType.Component)
10         {
11             if (userClass != null)
12                 g = (GComponent)Activator.CreateInstance(userClass);
13             else
14                 g = UIObjectFactory.NewObject(item);
15         }
16         else
17             g = UIObjectFactory.NewObject(item);
18
19         if (g == null)
20             return null;
21
22         _constructing++;
23         g.packageItem = item;
24         g.ConstructFromResource();
25         _constructing--;
26         return g;
27     }
```

根据包数据中记录的此项目的不同类型，把它创建出来，然后执行它的构造函数。

创建方法有两大类：

1. 自定义类型

这里，允许扩展fairy内置的PackageItemType枚举类型；

但此扩展类型必须继承自FairyGUI.GComponent；

将真实的类型，作为userClass参数传入；

调用System.Activator.CreateInstance()静态方法；

并将得到的object对象，强制转换为GComponent类型。

2. fairy自带的类型

对于fairy自带类型，调用FairyGUI.UIObjectFactory.NewObject()静态方法；

如果有自定义的PackageItem.extensionCreator()方法，则执行它；

默认则根据此PackageItem对象objectType的不同，而执行各组件类的构造函数。

异步创建

所有异步创建fairy元件的方法，最终都调用了AsyncCreationHelper.CreateObject()方法。

```

1      public static void CreateObject(PackageItem item,
UIPackage.CreateObjectCallback callback)
2      {
3          Timers.inst.StartCoroutine(_CreateObject(item, callback));
4      }
5
6      static IEnumerator _CreateObject(PackageItem item,
UIPackage.CreateObjectCallback callback)
7      {
8          Stats.LatestObjectCreation = 0;
9          Stats.LatestGraphicsCreation = 0;
10
11         float frameTime = UIConfig.frameTimeForAsyncUIConstruction;
12
13         List<DisplayListItem> itemList = new List<DisplayListItem>();
14         DisplayListItem di = new DisplayListItem(item,
ObjectType.Component);
15         di.childCount = CollectComponentChildren(item, itemList);
16         itemList.Add(di);
17
18         int cnt = itemList.Count;
19         List<GObject> objectPool = new List<GObject>(cnt);
20         GObject obj;
21         float t = Time.realtimeSinceStartup;
22         bool alreadyNextFrame = false;
23
24         for (int i = 0; i < cnt; i++)
25         {
26             di = itemList[i];
27             if (di.packageItem != null)
28             {
29                 obj = UIObjectFactory.NewObject(di.packageItem);
30                 obj.packageItem = di.packageItem;
31                 objectPool.Add(obj);
32
33                 UIPackage._constructing++;
34                 if (di.packageItem.type == PackageItemType.Component)
35                 {
36                     int poolStart = objectPool.Count - di.childCount -
1;
37
38                     ((GComponent)obj).ConstructFromResource(objectPool,
poolStart);
39
40                     objectPool.RemoveRange(poolStart, di.childCount);
41                 }
42                 else
43                 {
44                     obj.ConstructFromResource();
45                 }
46                 UIPackage._constructing--;
47             }
48             else
49             {
50                 obj = UIObjectFactory.NewObject(di.type);
51                 objectPool.Add(obj);
52
53                 if (di.type == ObjectType.List && di.listItemCount > 0)

```



```

54         {
55             int poolStart = objectPool.Count - di.listItemCount
- 1;
56             for (int k = 0; k < di.listItemCount; k++)
57                 ((GList)obj).itemPool.ReturnObject(objectPool[k
+ poolStart]);
58             objectPool.RemoveRange(poolStart,
di.listItemCount);
59         }
60     }
61
62     if ((i % 5 == 0) && Time.realtimeSinceStartup - t >=
frameTime)
63     {
64         yield return null;
65         t = Time.realtimeSinceStartup;
66         alreadyNextFrame = true;
67     }
68 }
69
70 if (!alreadyNextFrame)
71     yield return null;
72
73 callback(objectPool[0]);
74 }

```

创建一个列表，用来记录这个组件的所有显示内容(DisplayListItem)。

首先，把这个PackageItem本身，作为一个objectType为ObjectType.Component的显示内容放入列表。

然后，把其各层子节点全部放入这个列表。

对应这个DisplayListItem列表，创建一个等大的GObject的对象池。

创建新的GObject，并放入对象池中。

执行此GObject对象的构造函数。

创建5个之后，等待下一帧再继续执行。

在下帧执行的时候，才执行创建后的回调。

fairy元件与unity物件的关联

创建UnityEngine.GameObject

FairyGUI.DisplayObject创建UnityEngine.GameObject

FairyGUI.DisplayObject通过其CreateGameObject等方法来创建UnityEngine.GameObject，并将这个物件放置到unity的DontDestroyOnLoad场景。

```

1     protected void CreateGameObject(string gameObjectName)
2     {
3         gameObject = new GameObject(gameObjectName);
4         cachedTransform = gameObject.transform;
5         if (Application.isPlaying)

```

```

6         Object.DontDestroyOnLoad(gameObject);
7         gameObject.hideFlags = DisplayOptions.hideFlags;
8         gameObject.SetActive(false);
9
10    #if FAIRYGUI_TEST
11        DisplayObjectInfo info =
12        gameObject.AddComponent<DisplayObjectInfo>();
13        info.displayObject = this;
14    #endif
15
16        _ownsGameObject = true;
17    }

```

之后，对于FairyGUI.DisplayObject，可以通过其gameObject属性来访问到UnityEngine.GameObject。

```
🔑 GameObject DisplayObject.gameObject { get; protected set; }
```

FairyGUI.GObject创建UnityEngine.GameObject

FairyGUI.GObject中有个CreateDisplayObject虚方法，由它的子类去复写这个虚方法，根据子类不同的需求去创建FairyGUI.DisplayObject，从而创建UnityEngine.GameObject。

```

🔍 [client] override void GComponent.CreateDisplayObject() (0 个引用)
🔍 [client] override void GGraph.CreateDisplayObject() (0 个引用)
🔍 [client] override void GImage.CreateDisplayObject() (0 个引用)
🔍 [client] override void GLoader.CreateDisplayObject() (0 个引用)
🔍 [client] override void GMovieClip.CreateDisplayObject() (0 个引用)
🔍 [client] override void GRichTextField.CreateDisplayObject() (0 个引用)
🔍 [client] override void GTextField.CreateDisplayObject() (0 个引用)
🔍 [client] override void GTextInput.CreateDisplayObject() (0 个引用)
🔍 [client] virtual void GObject.CreateDisplayObject() (1 个引用)
📄 client\Assets\External\FairyGUI\Scripts\UI\GObject.cs - (163, 4) : CreateDisplayObject();

```

以FairyGUI.GComponent为例，它创建了一个FairyGUI.Container（继承自FairyGUI.DisplayObject）。

```

1    override protected void CreateDisplayObject()
2    {
3        rootContainer = new Container("GComponent");
4        rootContainer.gOwner = this;
5        rootContainer.onUpdate = OnUpdate;
6        container = rootContainer;
7
8        displayObject = rootContainer;
9    }

```

之后，对于FairyGUI.GObject，可以通过其displayObject属性来访问FairyGUI.DisplayObject，从而访问到UnityEngine.GameObject。

```
🔑 DisplayObject GObject.displayObject { get; protected set; }
```

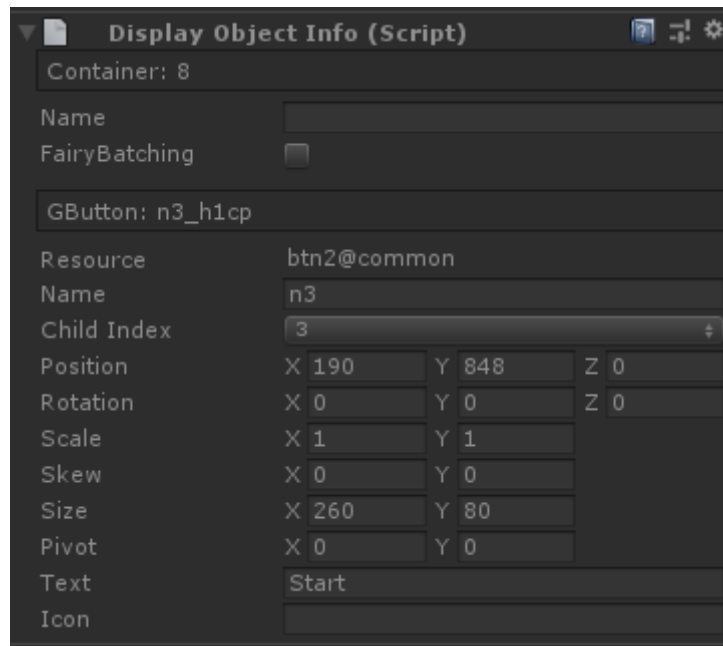
在Unity3D编辑器中查看FairyGUI创建的物件的信息

当**FAIRYGUI_TEST**这个fairy自定义给unity的宏开启后，FairyGUI.DisplayObject在创建UnityEngine.GameObject时，为其添加了**DisplayObjectInfo**组件。

fairy在其Editor目录下，**DisplayObjectEditor**为DisplayObjectInfo组件定制了其unity编辑器扩展内容。

```
1 [CustomEditor(typeof(DisplayObjectInfo))]  
2 public class DisplayObjectEditor : Editor
```

在Unity3D编辑器运行时，可以在Inspector视窗中查看此UnityEngine.GameObject所对应的FairyGUI.DisplayObject相关信息。



fairy创建物件汇总

显示物件的分类

以上得出，所有的UnityEngine.GameObject都由FairyGUI.DisplayObject来创建。

在类图中发现，DisplayObject的子类，分为3大类：

- 空物件Container
- 平面UI物件，包括：
 - 图片Image
 - 文本TextField
 - 图形Shape
 - 选项图形SelectionShape
- 3D物件GoWrapper

NGraphics

MeshFilter和MeshRenderer

这些平面物件类型，在创建时，均创建了一个NGraphics类的对象。

```
1 public NGraphics(GameObject gameObject)  
2 {  
3     this.gameObject = gameObject;
```

```

4
5         _alpha = 1f;
6         _shader = ShaderConfig.imageShader;
7         _color = Color.white;
8         _meshFactory = this;
9
10        meshFilter = gameObject.AddComponent<MeshFilter>();
11        meshRenderer = gameObject.AddComponent<MeshRenderer>();
12    #if (UNITY_5 || UNITY_5_3_OR_NEWER)
13        meshRenderer.shadowCastingMode =
14        UnityEngine.Rendering.ShadowCastingMode.Off;
15        meshRenderer.reflectionProbeUsage =
16        UnityEngine.Rendering.ReflectionProbeUsage.Off;
17    #else
18        meshRenderer.castShadows = false;
19    #endif
20        meshRenderer.receiveShadows = false;
21
22        mesh = new Mesh();
23        mesh.name = gameObject.name;
24        mesh.MarkDynamic();
25
26        meshFilter.mesh = mesh;
27
28        meshFilter.hideFlags = DisplayOptions.hideFlags;
29        meshRenderer.hideFlags = DisplayOptions.hideFlags;
30        mesh.hideFlags = DisplayOptions.hideFlags;
31
32        Stats.LatestGraphicsCreation++;
33    }

```

FairyGUI.NGraphics在构建时,

为这个UnityEngine.GameObject创建了,

一个UnityEngine.MeshFilter, 和, 一个UnityEngine.MeshRenderer组件,

并将MeshFilter组件的mesh作为其mesh属性存储下来。

Material

NGraphics的, material属性, 和, _material字段, 为UnityEngine.Material类型,

存储了于这个NGraphics相关的MeshRenderer.sharedMaterial的材质球信息。

```

1    public Material material
2    {
3        get { return _material; }
4        set
5        {
6            _material = value;
7            if (_material != null)
8            {
9                _customMaterial = true;
10               meshRenderer.sharedMaterial = _material;
11               if (_texture != null)
12                   _material.mainTexture = _texture.nativeTexture;
13            }
14        }
15    }

```

```

14         else
15         {
16             _customMaterial = false;
17             meshRenderer.sharedMaterial = null;
18         }
19     }
20 }

```

更改着色器和贴图

1.NGraphics.UpdateManager()

NGraphics中的UpdateManager()方法，用来对着色器和纹理贴图，做统一更新处理。

```

1     void UpdateManager()
2     {
3         MaterialManager mm;
4         if (_texture != null)
5             mm = _texture.GetMaterialManager(_shader,
6             _materialKeywords);
7         else
8             mm = null;
9         if (_manager != null && _manager != mm)
10            _manager.Release();
11            _manager = mm;
12    }

```

这里，创建了一个FairyGUI.MaterialManager类的对象，并将它存储在了NGraphics._manager字段中。

2.创建MaterialManager

通过NTexture.GetMaterialManager()方法来创建MaterialManager。

```

1     public MaterialManager GetMaterialManager(string shaderName,
2     string[] keywords)
3     {
4         if (_root != this)
5         {
6             if (_root == null)
7                 return null;
8             else
9                 return _root.GetMaterialManager(shaderName, keywords);
10        }
11
12        if (_materialManagers == null)
13            _materialManagers = new Dictionary<string, MaterialManager>
14            ();
15
16        string key = shaderName;
17        if (keywords != null)
18        {
19            //对于带指定关键字的，目前的设计是不参加共享材质了，因为逻辑会变得更复
20            杂
21
22            key = shaderName + "_" + _gCounter++;
23        }
24    }

```

```

21         MaterialManager mm;
22         if (!_materialManagers.TryGetValue(key, out mm))
23         {
24             mm = new MaterialManager(this,
ShaderConfig.GetShader(shaderName), keywords);
25             mm._managerKey = key;
26             _materialManagers.Add(key, mm);
27         }
28
29         return mm;
30     }

```

3.获取着色器

直接使用UnityEngine提供的Shader.Find()方法来获取着色器UnityEngine.Shader。

```

1         public static Shader GetShader(string name)
2         {
3             Shader shader = Get(name);
4             if (shader == null)
5             {
6                 Debug.LogWarning("FairyGUI: shader not found: " + name);
7                 shader = Shader.Find("UI/Default");
8             }
9             shader.hideFlags = DisplayOptions.hideFlags;
10
11             if (_propertyIDs == null)
12                 InitPropertyIDs();
13
14             return shader;
15         }

```

每帧更新

上文StageEngine部分提到，在StageEngine这个继承自UnityEngine.MonoBehaviour的类中，有LateUpdate()方法，调用了根目录下所有DisplayObject子节点的Update方法。

在DisplayObject.Update()方法中，调用了NGraphics.Update()方法。

(待定：渲染逻辑)

Image

纹理贴图的数据流

1.在构建GImage时，设置Image内容

Image在GImage中，作为其DisplayObject.displayObject属性和GImage._content字段而被存储。

在此GImage类的对象，被构建时，执行了其ConstructFromResource()方法。

```

1         override public void ConstructFromResource()
2         {
3             packageItem.Load();
4
5             sourcewidth = packageItem.width;
6             sourceheight = packageItem.height;
7             initwidth = sourcewidth;

```

```

8         initHeight = sourceHeight;
9         _content.scale9Grid = packageItem.scale9Grid;
10        _content.scaleByTile = packageItem.scaleByTile;
11        _content.tileGridIndice = packageItem.tileGridIndice;
12        _content.texture = packageItem.texture;
13
14        SetSize(sourceWidth, sourceHeight);
15    }

```

2.加载图片LoadImage()

此方法中，首先调用了PackageItem的Load()方法，而最终调用了UIPackage类的对象的GetItemAsset()方法。

由于其type值为PackageItemType.Image，所以，执行了LoadImage()方法。

```

1    void LoadImage(PackageItem item)
2    {
3        AtlasSprite sprite;
4        if (_sprites.TryGetValue(item.id, out sprite))
5            item.texture = new
NTexture((NTexture)GetItemAsset(sprite.atlas), sprite.rect, sprite.rotated);
6        else
7            item.texture = NTexture.Empty;
8    }

```

3.加载图集LoadAtlas()

根据此图片PackageItem，在UIPackage记录的_sprites词典中，找到对应的AtlasSprite图集数据。

也就找到了此图片PackageItem，所对应的图集PackageItem。

再次调用UIPackage类的对象的GetItemAsset()方法，执行了图集的加载方法LoadAtlas()。

```

1    void LoadAtlas(PackageItem item)
2    {
3        string ext = Path.GetExtension(item.file);
4        string fileName = item.file.Substring(0, item.file.Length -
ext.Length);
5
6        Texture tex = null;
7        Texture alphaTex = null;
8        DestroyMethod dm;
9
10       if (_fromBundle)
11       {
12           if (_resBundle != null)
13           {
14               #if (UNITY_5 || UNITY_5_3_OR_NEWER)
15                   tex = _resBundle.LoadAsset<Texture>(fileName);
16               #else
17                   tex = (Texture2D)_resBundle.Load(fileName,
typeof(Texture2D));
18               #endif
19           }
20           else
21               Debug.LogWarning("FairyGUI: bundle already unloaded.");
22       }

```

```

23         dm = DestroyMethod.None;
24     }
25     else
26         tex = (Texture)_loadFunc(fileName, ext, typeof(Texture),
out dm);
27
28     if (tex == null)
29         Debug.LogWarning("FairyGUI: texture '" + item.file + "' not
found in " + this.name);
30     else if (!(tex is Texture2D))
31     {
32         Debug.LogWarning("FairyGUI: settings for '" + item.file +
"' is wrong! Correct values are: (Texture Type=Default, Texture
Shape=2D)");
33         tex = null;
34     }
35     else
36     {
37         if (((Texture2D)tex).mipmapCount > 1)
38             Debug.LogWarning("FairyGUI: settings for '" + item.file
+ "' is wrong! Correct values are: (Generate Mip Maps=unchecked)");
39     }
40
41     if (tex != null)
42     {
43         fileName = fileName + ".a";
44         if (_fromBundle)
45         {
46             if (_resBundle != null)
47             {
48                 #if (UNITY_5 || UNITY_5_3_OR_NEWER)
49                     alphaTex = _resBundle.LoadAsset<Texture2D>
(fileName);
50                 #else
51                     alphaTex = (Texture2D)_resBundle.Load(fileName,
typeof(Texture2D));
52                 #endif
53             }
54         }
55         else
56             alphaTex = (Texture2D)_loadFunc(fileName, ext,
typeof(Texture2D), out dm);
57     }
58
59     if (tex == null)
60     {
61         tex = NTexture.CreateEmptyTexture();
62         dm = DestroyMethod.Destroy;
63     }
64
65     if (item.texture == null)
66     {
67         item.texture = new NTexture(tex, alphaTex, (float)tex.width
/ item.width, (float)tex.height / item.height);
68         item.texture.destroyMethod = dm;
69     }
70     else
71     {

```



```

72         item.texture.Reload(tex, alphaTex);
73         item.texture.destroyMethod = dm;
74     }
75 }

```

4. 获取UnityEngine.Texture数据

使用UnityEngine提供的Resources.Load()方法，或是AssetBundle.LoadAsset()方法，

可以获得这个图集的纹理UnityEngine.Texture。

另外，对于在fairy编辑器中设置了透明通道分离的图集，需多获得一个纹理UnityEngine.Texture类的对象。

如果没有获取到纹理集，则使用NTexture.CreateEmptyTexture()方法创建一个空白的纹理（继承自UnityEngine.Texture的UnityEngine.Texture2D）。

5. 创建FairyGUI.NTexture，存储到PackageItem.texture

根据这两个UnityEngine.Texture图集，图片的宽高信息，创建一个FairyGUI.NTexture，存储这些信息。

并将这个FairyGUI.NTexture类的对象，存储到PackageItem.texture字段中。

6. 刷新纹理

如果在存储这个FairyGUI.NTexture类的对象时，PackageItem.texture字段已经有值，

则调用NTexture.Reload()方法，来刷新纹理。

（待定，没有被调用过。）

7. 将数据传递给Image

在纹理和图片都加载完成之后，回到GImage.ConstructFromResource()方法，继续执行。

将PackageItem.texture的值，这个NTexture数据，传递给，GImage._content这个Image，作为其texture属性存储下来。

8. 将数据传递给NGraphics

在设置Image.texture这个属性值时，会调用Image.UpdateTexture()方法，

将这个NTexture值，传递给Image.graphics这个属性（Image.graphics为NGraphics类型），

设置为NGraphics.texture属性值。

GoWrapper

（待定）

坐标系统

定位点与锚点

GObject.pivot表示此GObject的定位点，这个Vector2的x和y的取值范围为0~1。

GObject.pivotAsAnchor可将定位点设置为锚点，而不能脱离定位点去设置锚点。

坐标转换

fairy系的坐标系，原点为左上角，2D坐标系；

而unity系的坐标系，为左手3D坐标系。

DisplayObject.LocalToGlobal

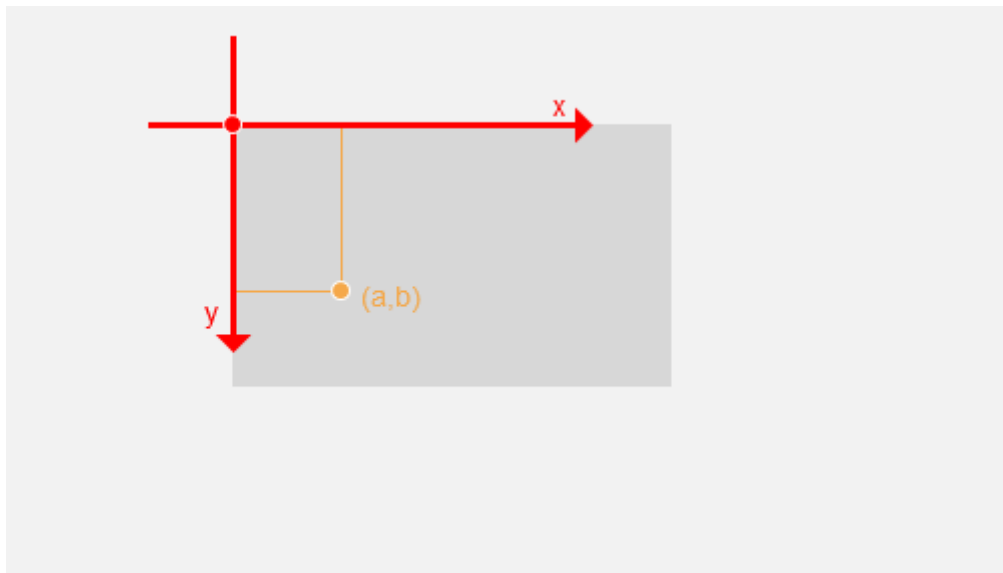
将fairy的本地坐标（即相对于FairyGUI.DisplayObject坐标），转换为fairy系的屏幕坐标（如FairyGUI.InputEvent所记录的坐标），可以直接使用DisplayObject.LocalToGlobal方法。

源码

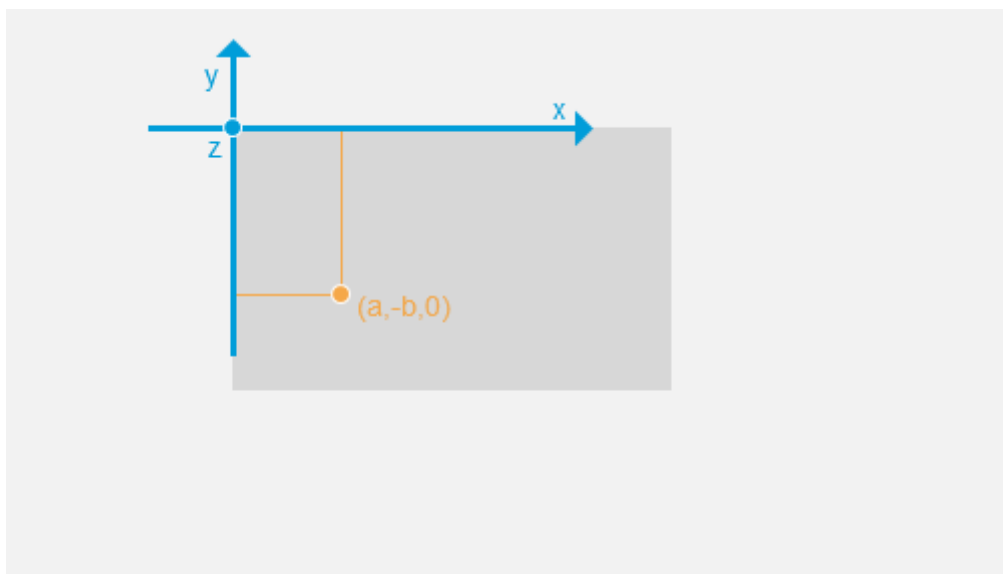
```
1      public Vector2 LocalToGlobal(Vector2 point)
2      {
3          Container wsc = this.worldSpaceContainer;
4
5          Vector3 worldPoint =
6      this.cachedTransform.TransformPoint(point.x, -point.y, 0);
7          if (wsc != null)
8          {
9              if (wsc.hitArea is MeshColliderHitTest) //Not supported for
10      UIPainter, use TransformPoint instead.
11                  return new Vector2(float.NaN, float.NaN);
12
13          Vector3 screePoint =
14      wsc.GetRenderCamera().WorldToScreenPoint(worldPoint);
15          return new Vector2(screePoint.x, Stage.inst.stageHeight -
16      screePoint.y);
17      }
18      else
19      {
20          point =
21      Stage.inst.cachedTransform.InverseTransformPoint(worldPoint);
22          point.y = -point.y;
23          return point;
24      }
25      }
```

解读

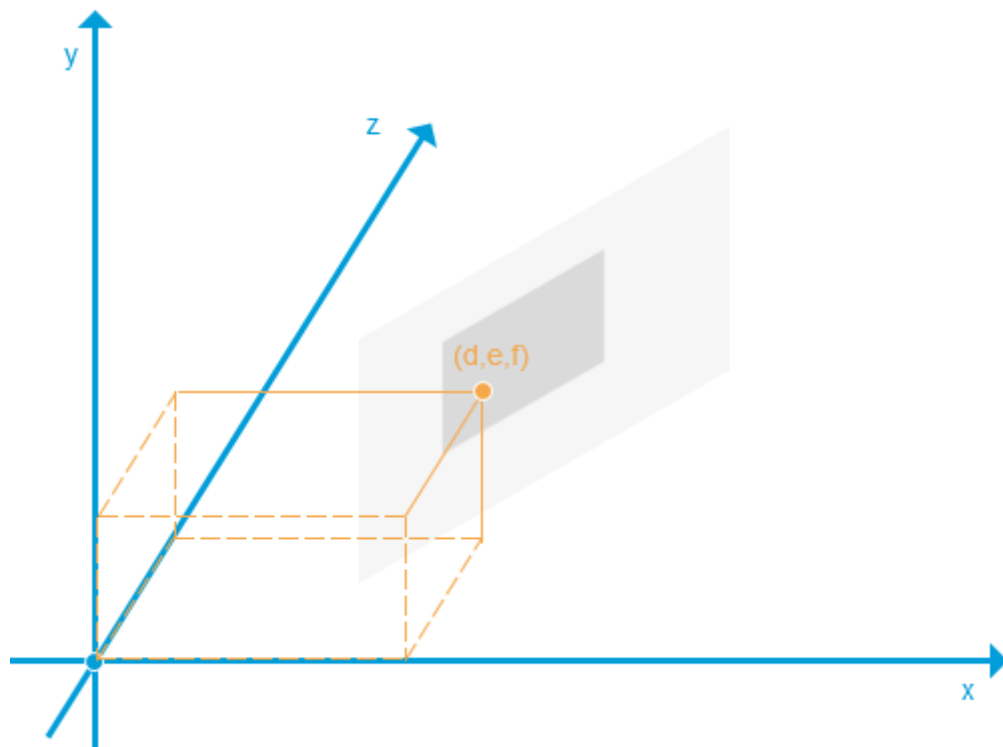
1. 获取到FairyGUI.DisplayObject所记录的坐标(a,b)，它是此DisplayObject相对于其父节点的坐标。



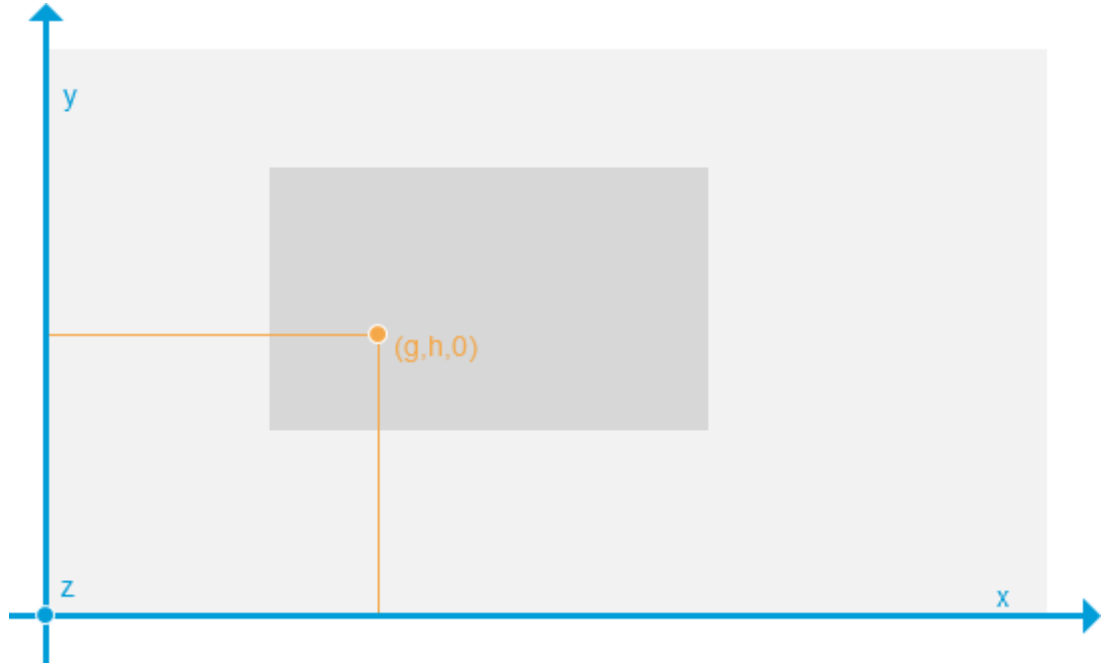
2. 调整坐标值，得到其unity系本地坐标(a,-b,0)，便于后续利用UnityEngine提供的方法去操纵坐标。



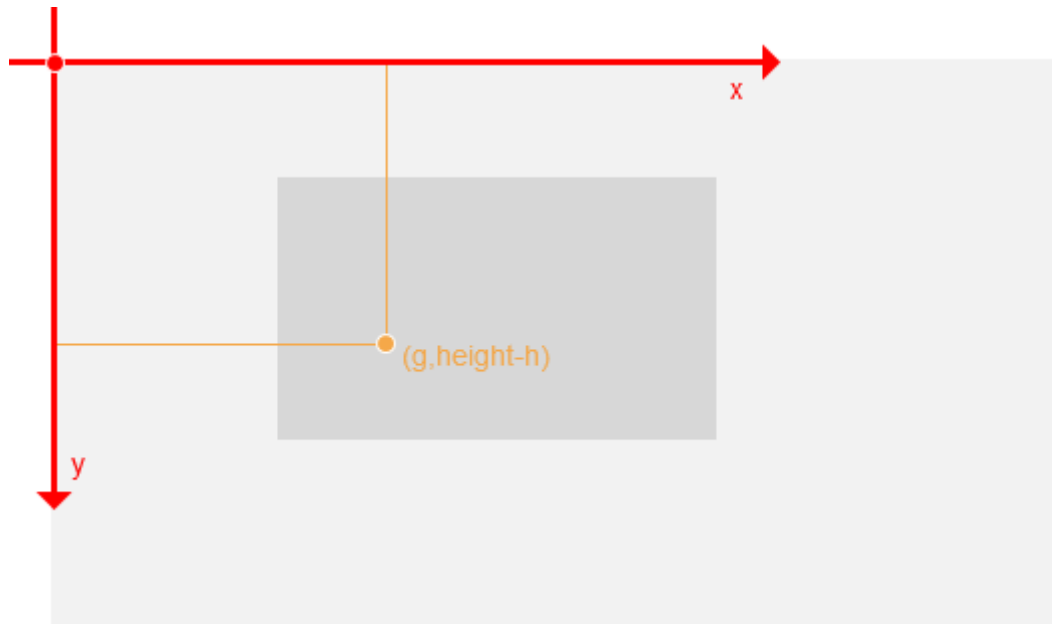
3. 根据此DisplayObject的cachedTransform属性（UnityEngine.Transform类型），使用UnityEngine.Transform.TransformPoint()方法，计算出其unity系世界坐标(d,e,f)。



4. 此DisplayObject的worldSpaceContainer属性，记录了此DisplayObject的根节点容器Container，使用Container.GetRenderCamera()方法，获取到它的渲染相机，一个UnityEngine.Camera。
5. 根据此UnityEngine.Camera，使用Camera.WorldToScreenPoint()方法，将之前得到的unity系世界坐标(d,e,f)，转化为此相机内的屏幕坐标(g,h,0)。



6. 此屏幕坐标为unity系坐标，若要将其应用于fairy系坐标的计算，还需将其转化为fairy系坐标。
[FairyGUI.Stage]Stage.inst这个全局静态变量，记录了屏幕信息。



7. 最终得到的这个fairy系屏幕坐标，将可以与从FairyGUI.InputEvent中取得的坐标，做计算。

坐标转换方法汇总

DisplayObject.LocalToGlobal

DisplayObject.GlobalToLocal

local指局部坐标，fairy的本地坐标，即相对于FairyGUI.DisplayObject坐标。

global指全局坐标，fairy系的屏幕坐标，如FairyGUI.InputEvent所记录的坐标。

GObject.LocalToRoot

GObject.RootToLocal

如果有UI适配导致的全局缩放，那么逻辑屏幕大小和物理屏幕大小将会不一致。

这里的local还是指局部坐标。

而root将不同于global（物理屏幕坐标），而是指逻辑屏幕坐标。

DisplayObject.WorldToLocal

转换世界坐标点到等效的本地xy平面的点。等效的意思是他们在屏幕方向看到的位置一样。

这种需要主要出现在碰撞检测。

DisplayObject.TransformPoint

DisplayObject.TransformRect

两个DisplayObject之间的坐标转换。

事件机制


EventDispatcher 事件分发器

从类图可以看出，所有fairy可交互型组件都继承自EventDispatcher。


这样，每种fairy组件，便都能使用事件机制。

EventListener 事件接收器

创建事件接收器时，需要一个事件分发器和一个事件名作为构造参数。

```
 EventListener.EventListener(EventDispatcher owner, string type)
```


这使得每个事件接收器都对应一个事件分发器，一个事件桥。

```
 (字段) EventBridge EventListener._bridge
```


而一个事件分发器，如GObject，往往，根据需求，会有多个事件接收器作为属性字段，如GObject.onClick，GObject.onRightClick，之类。

EventBridge 事件桥


创建事件桥时，需要一个事件分发器作为构造参数。

```
 EventBridge.EventBridge(EventDispatcher owner)
```

这使得每个事件桥都对应一个事件分发器。


```
 (字段) EventDispatcher EventBridge.owner
```

而一个事件分发器，应允许它使用多个事件桥，因此事件分发器中用一个词典记录了事件名-事件桥。

```
 (字段) Dictionary<string, EventBridge> EventDispatcher._dic
```


通过事件接收器注册事件

在业务层，比如给按钮注册点击事件，fairy使用的是**EventListener.Add**之类的方法。

```
 void EventListener.Add(EventCallback1 callback)
```


```
1 public void Add(EventCallback1 callback)
2 {
3     _bridge.Add(callback);
4 }
```

事件接收器将这个回调函数（代理）作为参数，传递给了自身宿主的事件桥。

```
 void EventBridge.Add(EventCallback1 callback)
```


```
1 public void Add(EventCallback1 callback)
2 {
3     _callback1 -= callback;
4     _callback1 += callback;
5 }
```

事件桥再将这个回调函数存储在自己的字段中，以备使用。

```
 (字段) EventCallback1 EventBridge._callback1
```

通过事件分发器注册事件

在业务层，有时需要使用事件机制来传递参数，我们一般使用的是**EventDispatcher.AddEventListener**之类的方法。


```
 void EventDispatcher.AddEventListener(string strType, EventCallback1 callback)
```

```
1 public void AddEventListener(string strType, EventCallback1
callback)
2 {
3     if (strType == null)
4         throw new Exception("event type cant be null");
5
6     if (_dic == null)
7         _dic = new Dictionary<string, EventBridge>();
8
9     EventBridge bridge = null;
10    if (!_dic.TryGetValue(strType, out bridge))
11    {
12        bridge = new EventBridge(this);
13        _dic[strType] = bridge;
14    }
15    bridge.Add(callback);
16 }
```


事件分发器注册事件的逻辑，和事件接收器逻辑基本一致：将回调函数传递给自身记录的事件桥词典，然后由事件桥去存储回调函数。

抛出事件

在业务层，我们抛出自定义事件，一般是使用**EventDispatcher.DispatchEvent**方法。

```
 bool EventDispatcher.DispatchEvent(string strType, object data)
```

而事件分发器，会在自身记录的事件桥词典中，索引出这个事件桥，让事件桥去做相关逻辑操作。


```
 void EventBridge.CallInternal(EventContext context)
```

```
1      public void CallInternal(EventContext context)
2      {
3          _dispatching = true;
4          context.sender = owner;
5          try
6          {
7              if (_callback1 != null)
8                  _callback1(context);
9              if (_callback0 != null)
10                 _callback0();
11            }
12            finally
13            {
14                _dispatching = false;
15            }
16        }
```

冒泡机制

源码

而用户的输入操作，fairy使用**EventDispatcher.BubbleEvent**抛出事件。

```
 bool EventDispatcher.BubbleEvent(string strType, object data, List<EventBridge> addChain)
```

```
1      internal bool BubbleEvent(string strType, object data,
2      List<EventBridge> addChain)
3      {
4          EventContext context = EventContext.Get();
5          context.initiator = this;
6
7          context.type = strType;
8          context.data = data;
9          if (data is InputEvent)
10             sCurrentInputEvent = (InputEvent)data;
11             context.inputEvent = sCurrentInputEvent;
12             List<EventBridge> bubbleChain = context.callChain;
13             bubbleChain.Clear();
14
15             GetChainBridges(strType, bubbleChain, true);
16
17             int length = bubbleChain.Count;
18             for (int i = length - 1; i >= 0; i--)
19             {
20                 bubbleChain[i].CallCaptureInternal(context);
21                 if (context._touchCapture)
22                 {
23                     }
```

```

22         context._touchCapture = false;
23         if (strType == "onTouchBegin")
24
25             Stage.inst.AddTouchMonitor(context.inputEvent.touchId,
26             bubbleChain[i].owner);
27
28         if (!context._stopsPropagation)
29         {
30             for (int i = 0; i < length; ++i)
31             {
32                 bubbleChain[i].CallInternal(context);
33
34                 if (context._touchCapture)
35                 {
36                     context._touchCapture = false;
37                     if (strType == "onTouchBegin")
38
39                         Stage.inst.AddTouchMonitor(context.inputEvent.touchId,
40                         bubbleChain[i].owner);
41
42                     if (context._stopsPropagation)
43                         break;
44                 }
45
46                 if (addChain != null)
47                 {
48                     length = addChain.Count;
49                     for (int i = 0; i < length; ++i)
50                     {
51                         EventBridge bridge = addChain[i];
52                         if (bubbleChain.IndexOf(bridge) == -1)
53                         {
54                             bridge.CallCaptureInternal(context);
55                             bridge.CallInternal(context);
56                         }
57                     }
58                 }
59
60                 EventContext.Return(context);
61                 context.initiator = null;
62                 context.sender = null;
63                 context.data = null;
64                 return context._defaultPrevented;
65             }

```

其中，调用了EventDispatcher.GetChainBridges这个方法，用来获取这个事件分发器及其所有父节点事件分发器的事件桥。

```

void EventDispatcher.GetChainBridges(string strType, List<EventBridge> chain, bool bubble)

```

```

1         internal void GetChainBridges(string strType, List<EventBridge>
chain, bool bubble)

```



```

2      {
3          EventBridge bridge = TryGetEventBridge(strType);
4          if (bridge != null && !bridge.isEmpty)
5              chain.Add(bridge);
6
7          if ((this is DisplayObject) && ((DisplayObject)this).gOwner !=
null)
8          {
9              bridge =
((DisplayObject)this).gOwner.TryGetEventBridge(strType);
10             if (bridge != null && !bridge.isEmpty)
11                 chain.Add(bridge);
12         }
13
14         if (!bubble)
15             return;
16
17         if (this is DisplayObject)
18         {
19             DisplayObject element = (DisplayObject)this;
20             while ((element = element.parent) != null)
21             {
22                 bridge = element.TryGetEventBridge(strType);
23                 if (bridge != null && !bridge.isEmpty)
24                     chain.Add(bridge);
25
26                 if (element.gOwner != null)
27                 {
28                     bridge = element.gOwner.TryGetEventBridge(strType);
29                     if (bridge != null && !bridge.isEmpty)
30                         chain.Add(bridge);
31                 }
32             }
33         }
34         else if (this is GObject)
35         {
36             GObject element = (GObject)this;
37             while ((element = element.parent) != null)
38             {
39                 bridge = element.TryGetEventBridge(strType);
40                 if (bridge != null && !bridge.isEmpty)
41                     chain.Add(bridge);
42             }
43         }
44     }

```

然后去处理这个冒泡链列表内容。

解读规则

1. 从最上层父节点事件分发器开始，处理捕获型事件，依次往下，最后处理这个事件分发器本身所注册的捕获型事件。
2. 最先处理这个事件分发器本身所注册的非捕获型事件，依次往上，处理其父节点事件分发器所注册的非捕获型事件。直到遇到了禁用冒泡的父节点事件分发器，便中断，不再往上冒泡，不再处理上层父节点事件分发器所注册的非捕获型事件。
3. 对于额外加入的冒泡链，只要此事件分发器未禁用冒泡，则依次处理此冒泡链中所有事件分发器的注册事件。

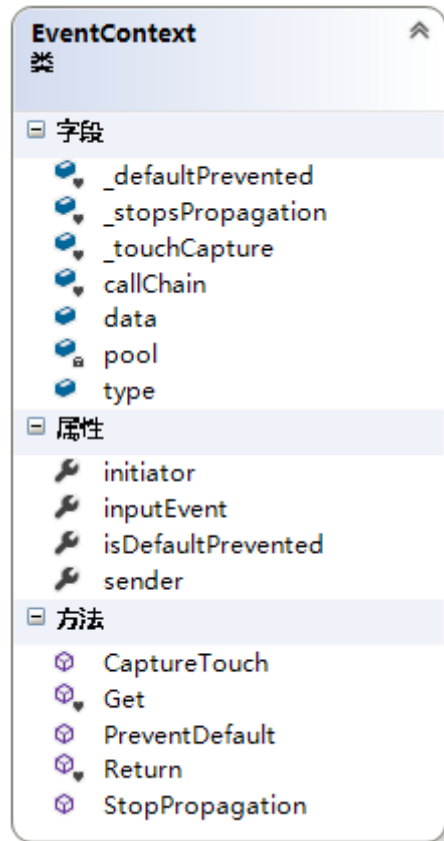
EventContext 事件内容

fairy源码提供2种类型的事件回调代理：EventCallback0和EventCallback1。

```
delegate void FairyGUI.EventCallback0()
```

```
delegate void FairyGUI.EventCallback1(FairyGUI.EventContext context)
```

EventCallback0不带参数，而EventCallback1带EventContext类型参数。



EventDispatcher EventContext.sender 此事件内容的事件分发器

object EventContext.initiator 此事件内容的创始者（而不可能是父节点事件分发器）

string EventContext.type 事件名

object EventContext.data 自定义数据

InputEvent EventContext.inputEvent 输入信息数据

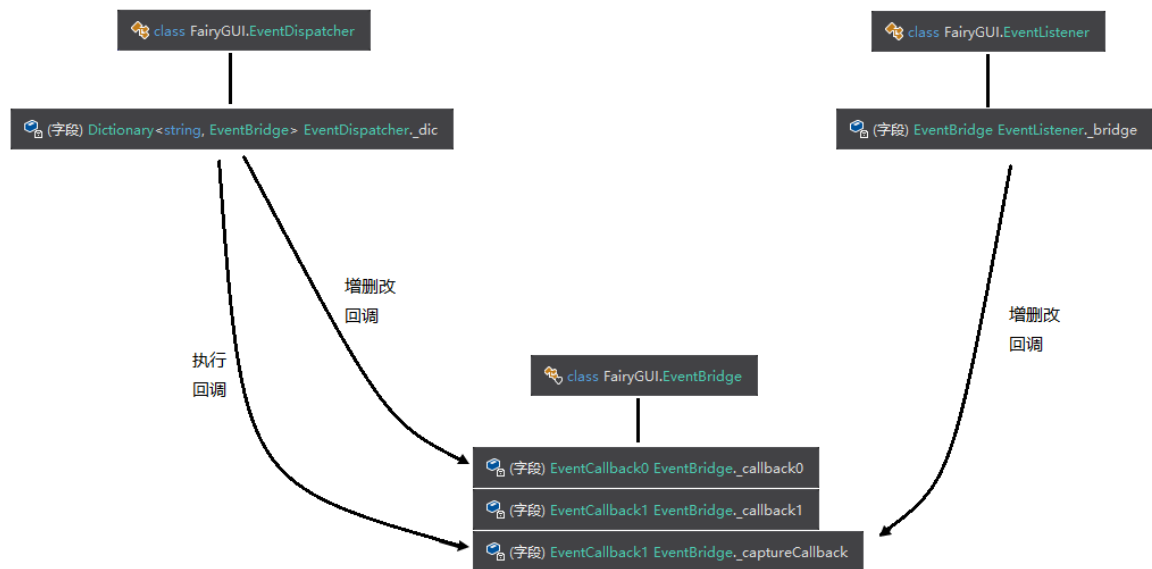
void EventContext.StopPropagation() 不再往上层冒泡

void EventContext.PreventDefault() 对事件不做响应

void EventContext.CaptureTouch() 触摸事件类型设置为捕获型事件

EventContext EventContext.Get() 取出事件内容池中第一个元素

事件流概览

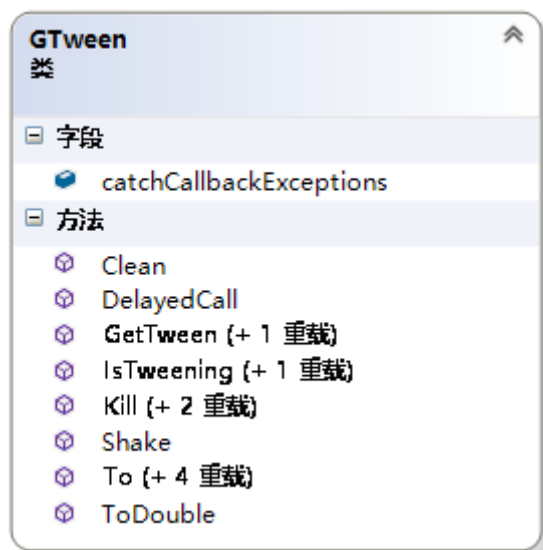


缓动

GTween

概览

GTween类是一个工具类，它提供了与缓动相关的一切静态方法、记录了静态字段，供全局调用。



创建缓动的方法

在业务层，创建缓动的方式，是使用GTween.To的多种重载方法和其他方法。

public class GTween
public static GTweener To(float startValue, float endValue, float duration)
public static GTweener To(Vector2 startValue, Vector2 endValue, float duration)
public static GTweener To(Vector3 startValue, Vector3 endValue, float duration)
public static GTweener To(Vector4 startValue, Vector4 endValue, float duration)
public static GTweener To(Color startValue, Color endValue, float duration)
public static GTweener ToDouble(double startValue, double endValue, float duration)
public static GTweener Shake(Vector3 startValue, float amplitude, float duration)

创建缓动的逻辑

使用GTween.To方法创建缓动时，它调用了TweenManager.CreateTween()方法，并将这个缓动GTweener记录在TweenManager中，便于统一管理。如：

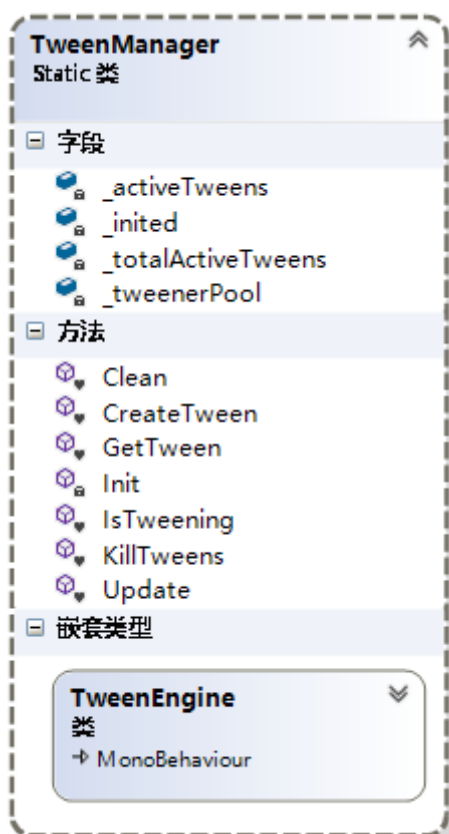
```

1      public static GTweener To(float startValue, float endValue, float
duration)
2      {
3          return TweenManager.CreateTween()._To(startValue, endValue,
duration);
4      }
```

TweenManager


概览

TweenManager用来管理fairy中的所有缓动。




主要属性字段

TweenManager._activeTweens 记录了正在活跃的缓动

 (字段) GTweener[] TweenManager._activeTweens

TweenManager._tweenerPool 为缓动的回收池

 (字段) List<GTweener> TweenManager._tweenerPool

每帧执行

TweenManager这个静态类内部嵌套一个TweenEngine类。

FairyGUI.TweenManager.TweenEngine类继承自UnityEngine.MonoBehaviour，因而包含了unity的Update方法。

```
1      class TweenEngine : MonoBehaviour
2      {
3          void Update()
4          {
5              TweenManager.Update();
6          }
7      }
```

再由TweenManager在其Init方法中，加载这个TweenEngine这个组件。

```
1      static void Init()
2      {
3          _inited = true;
4          if (Application.isPlaying)
5          {
6              GameObject gameObject = new GameObject("
[FairyGUI.TweenManager]");
7              gameObject.hideFlags = HideFlags.HideInHierarchy;
8              gameObject.SetActive(true);
9              Object.DontDestroyOnLoad(gameObject);
10
11              gameObject.AddComponent<TweenEngine>();
12          }
13      }
```

这样，TweenManager.Update方法就会被unity每帧执行。

GTweener

当使用TweenManager.CreateTween方法，来创建一个缓动创建了一个GTweener实例。

```
1      internal static GTweener CreateTween()
2      {
3          if (!_inited)
4              Init();
5
6          GTweener tweener;
7          int cnt = _tweenerPool.Count;
8          if (cnt > 0)
```

```

9         {
10             tweener = _tweenerPool[cnt - 1];
11             _tweenerPool.RemoveAt(cnt - 1);
12         }
13         else
14             tweener = new GTweener();
15         tweener._Init();
16         _activeTweens[_totalActiveTweens++] = tweener;
17
18         if (_totalActiveTweens == _activeTweens.Length)
19         {
20             GTweener[] newArray = new GTweener[_activeTweens.Length +
21             Mathf.CeilToInt(_activeTweens.Length * 0.5f)];
22             _activeTweens.CopyTo(newArray, 0);
23             _activeTweens = newArray;
24         }
25         return tweener;
26     }

```

在Update时，去执行_activeTweens字段中记录的所有GTweener实例的Update方法。

```

1     internal static void Update()
2     {
3         int cnt = _totalActiveTweens;
4         int freePosStart = -1;
5         for (int i = 0; i < cnt; i++)
6         {
7             GTweener tweener = _activeTweens[i];
8             if (tweener == null)
9             {
10                 if (freePosStart == -1)
11                     freePosStart = i;
12             }
13             else if (tweener._killed)
14             {
15                 tweener._Reset();
16                 _tweenerPool.Add(tweener);
17                 _activeTweens[i] = null;
18
19                 if (freePosStart == -1)
20                     freePosStart = i;
21             }
22             else
23             {
24                 if ((tweener._target is GObject) &&
25                 ((GObject)tweener._target)._disposed)
26                     tweener._killed = true;
27                 else if (!tweener._paused)
28                     tweener._Update();
29
30                 if (freePosStart != -1)
31                 {
32                     _activeTweens[freePosStart] = tweener;
33                     _activeTweens[i] = null;
34                     freePosStart++;
35                 }
36             }
37         }
38     }

```

```

35         }
36     }
37
38     if (freePosStart >= 0)
39     {
40         if (_totalActiveTweens != cnt) //new tweens added
41         {
42             int j = cnt;
43             cnt = _totalActiveTweens - cnt;
44             for (int i = 0; i < cnt; i++)
45             {
46                 _activeTweens[freePosStart++] = _activeTweens[j];
47                 _activeTweens[j] = null;
48                 j++;
49             }
50         }
51         _totalActiveTweens = freePosStart;
52     }
53 }

```

其他类与接口

GTweener

本文中所说的一个缓动就是一个GTweener类的实例。

[TweenValue] startValue 此缓动的开始帧数据

[TweenValue] endValue 此缓动的结束帧数据

[TweenValue] value 此缓动的当前帧数据

[float] duration 此缓动的过程时长

[object] target 此缓动的作用目标

[TweenPropType] _propType 此缓动的属性类型

[EaseType] _easeType 此缓动的类型

[GTweenCallback] _onUpdate 此缓动的回调委托



































[bool] _paused 当前暂停状态

[GPath] _path 目前没有使用













GTweener

类







字段

-  `_breakpoint`
-  `_delay`
-  `_deltaValue`
-  `_duration`
-  `_easeOvershootOrAmplitude`
-  `_easePeriod`
-  `_easeType`
-  `_elapsedTime`
-  `_ended`
-  `_endValue`
-  `_ignoreEngineTimeScale`
-  `_killed`
-  `_listener`
-  `_normalizedTime`
-  `_onComplete`
-  `_onComplete1`
-  `_onStart`
-  `_onStart1`
-  `_onUpdate`
-  `_onUpdate1`
-  `_path`
-  `_paused`
-  `_propType`
-  `_repeat`
-  `_smoothStart`
-  `_snapping`
-  `_started`
-  `_startValue`
-  `_target`
-  `_timeScale`
-  `_userData`
-  `_value`
-  `_valueSize`
-  `_yoyo`

属性

-  `allCompleted`
-  `completed`
-  `delay`
-  `deltaValue`
-  `duration`
-  `endValue`
-  `normalizedTime`
-  `repeat`
-  `startValue`
-  `target`
-  `userData`
-  `value`

方法

-  `_Init`
-  `_Reset`
-  `_Shake`
-  `_To (+ 5 重载)`
-  `_Update`
-  `CallCompleteCallback`

- CallCompleteCallback
- CallStartCallback
- CallUpdateCallback
- GTweener
- Kill
- OnComplete (+ 1 重载)
- OnStart (+ 1 重载)
- OnUpdate (+ 1 重载)
- Seek
- SetBreakpoint
- SetDelay
- SetDuration
- SetEase
- SetEaseOvershootOrAmplitude
- SetEasePeriod
- SetIgnoreEngineTimeScale
- SetListener
- SetPath
- SetPaused
- SetRepeat
- SetSnapping
- SetTarget (+ 1 重载)
- SetTimeScale
- SetUserData
- Update

TweenValue

TweenValue用于缓动的帧的数据。

TweenValue
 类

字段

- d
- w
- x
- y
- z

属性

- color
- this
- vec2
- vec3
- vec4

方法

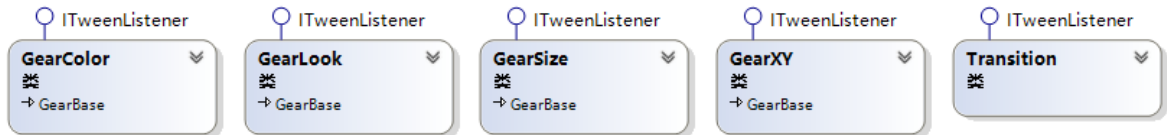
- SetZero
- TweenValue

ITweenListener

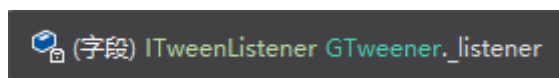
ITweenListener是缓动侦听接口。



一些有缓动需求的类实现了此接口。



在GTweener中记录了一个ITweenListener类型的_listener字段。



在GTweener中合适的时机去调用执行它的各种方法。

缓动的逻辑算法

算法出处

在GTweener.Update()方法中，调用了EaseManager.Evaluate()方法，用数学计算的方式，来获取缓动的当前数值。

EaseManager提供了多种缓动类型的计算。

EaseManager为引进的代码，此C#代码的主要作者为[Daniele Giardini](#)，少部分缓动方程来自[Robert Penner](#)。

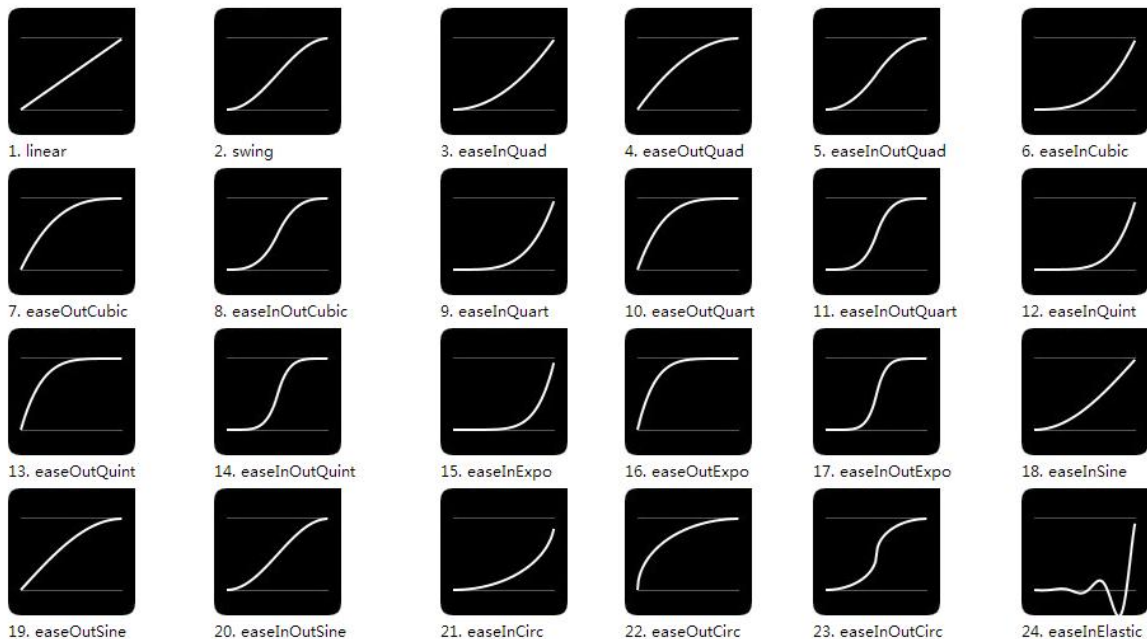
打开Daniele Giardini的个人网站会发现，fairy的缓动计算公式，即DOTween。

缓动类型

缓动类型记录在EaseType这个枚举中。

缓动类型效果示例：[Ease Visualizer](#)

缓动类型图解：



fairy自带的缓动功能汇总

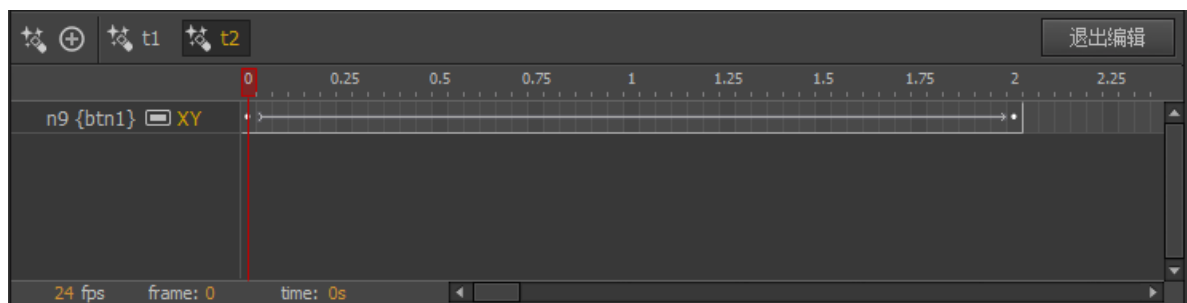
逻辑程序员编写代码调用GTween.To等方法，可以创建一个缓动；

fairy在编辑器中提供了一些便捷操作，让UE可以直接控制缓动，之后再动态创建出来。

在编辑器中有3种创建缓动的方式：

给组件创建动效

给组件创建动效Transition，能创建多个缓动，并灵活地控制它们。



对元件的属性控制启用缓动

给一个组件创建了控制器之后，可以对其元件设置属性控制。



勾选缓动之后，在创建此GearBase时，创建了一个GearTweenConfig对象，赋值给了此GearBase的_tweenConfig属性。

在GearBase这个抽象类中，有个Apply抽象方法。当控制器跳转页签时，会执行此Apply方法。

GearBase的各个子类复写了这个方法。

以GearXY.Apply()方法为例:

```
1      override public void Apply()
2      {
3          GearXYValue gv;
4          if (!_storage.TryGetValue(_controller.selectedPageId, out gv))
5              gv = _default;
6
7          if (_tweenConfig != null && _tweenConfig.tween &&
8              UIPackage._constructing == 0 && !disableAllTweenEffect)
9          {
10             if (_tweenConfig._twener != null)
11             {
12                 if (_tweenConfig._twener.endValue.x != gv.x ||
13                     _tweenConfig._twener.endValue.y != gv.y)
14                 {
15                     _tweenConfig._twener.Kill(true);
16                     _tweenConfig._twener = null;
17                 }
18                 else
19                     return;
20             }
21
22             if (_owner.x != gv.x || _owner.y != gv.y)
23             {
24                 if (_owner.CheckGearController(0, _controller))
25                     _tweenConfig._displayLockToken =
26                     _owner.AddDisplayLock();
27
28                 _tweenConfig._twener = GTween.To(_owner.xy, new
29                     Vector2(gv.x, gv.y), _tweenConfig.duration)
30                     .SetDelay(_tweenConfig.delay)
31                     .SetEase(_tweenConfig.easeType)
32                     .SetTarget(this)
33                     .SetListener(this);
34             }
35             else
36             {
37                 _owner._gearLocked = true;
38                 _owner.SetXY(gv.x, gv.y);
39                 _owner._gearLocked = false;
40             }
41         }
42     }
```

代码调用进度条的动态变化

调用GProgressBar.TweenValue方法, 在进度条的进度变化时播放一个缓动。

```
1      public GTweener TweenValue(double value, float duration)
2      {
3          double oldValule;
4
5          GTweener twener = GTween.GetTween(this,
6              TweenPropType.Progress);
7          if (twener != null)
8          {
```

```
8         oldvalule = twener.value.d;
9         twener.kill(false);
10    }
11    else
12        oldvalule = _value;
13
14    _value = value;
15    return GTween.ToDouble(oldvalule, _value, duration)
16        .SetEase(EaseType.Linear)
17        .SetTarget(this, TweenPropType.Progress);
18 }
```

UE相关

组件

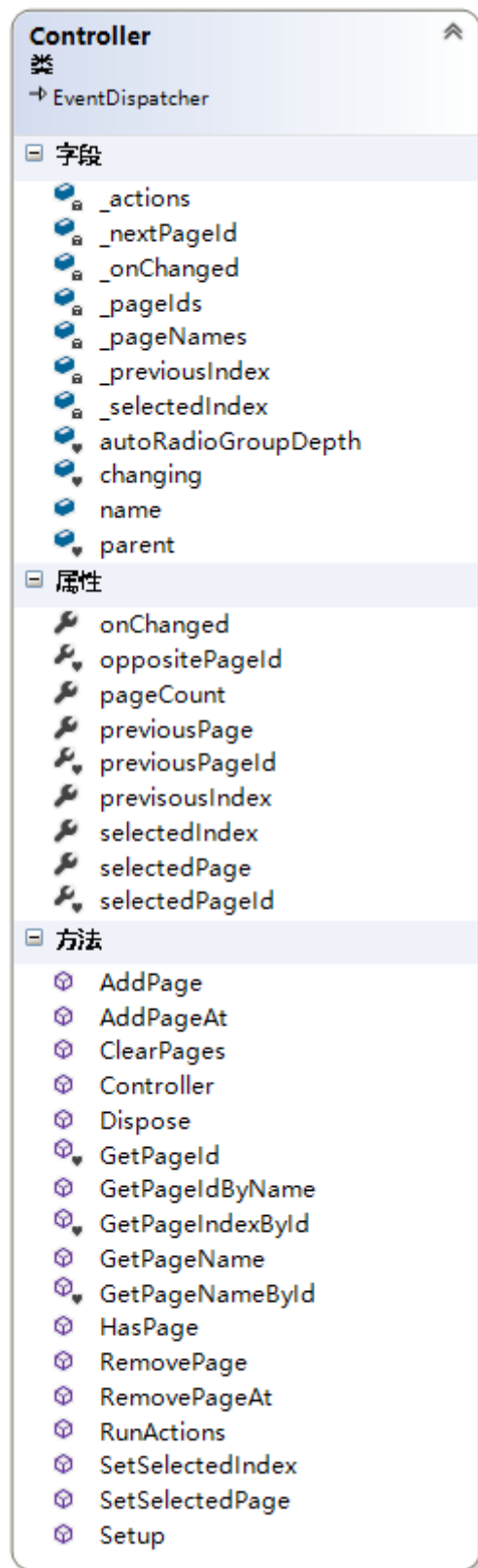
UE所使用的组件，可交互型组件，一般都继承自GObject。

可以顾名思义，在上文可交互类型组件的类图中，找到它所对应的类。

控制器

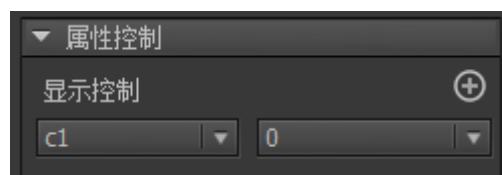
Controller

一个控制器即为一个Controller类的实例。



属性控制

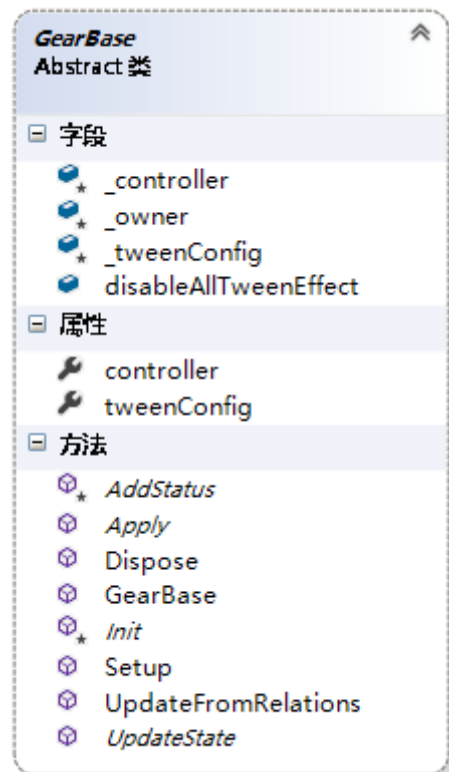
在fairy元件的属性控制栏，可以设置其在此控制器下的显示控制。



除此之外，fairy还提供了多种属性控制类型。



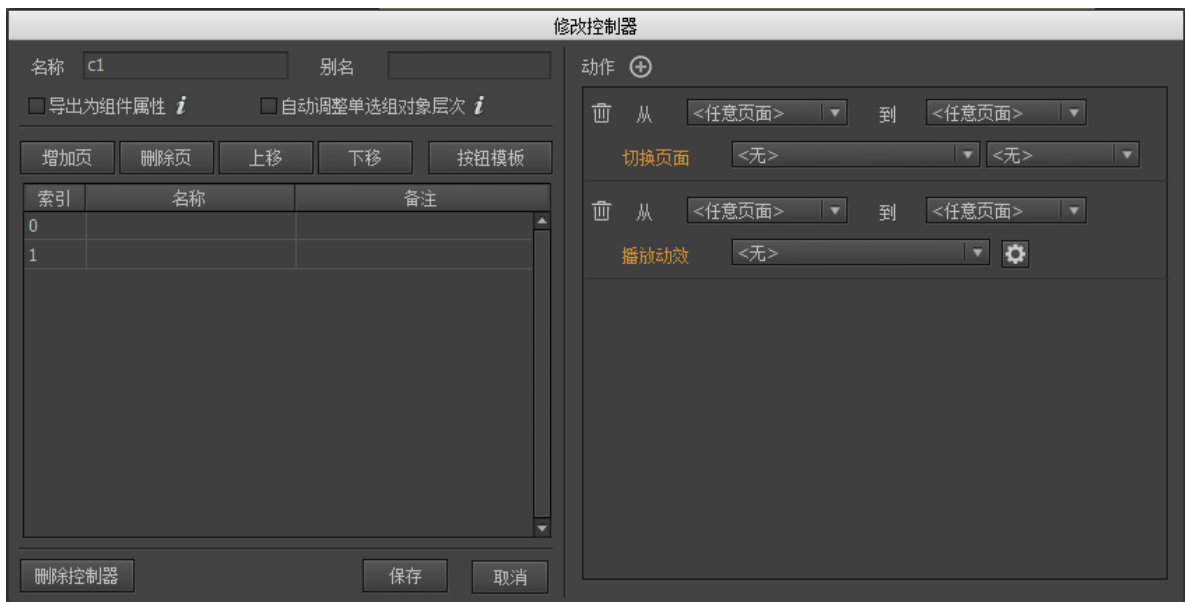
每种属性控制，都是一种GearBase类型的子类。



在上文控制器的缓动类图中，能找到其对应的类。

控制器的动作

在修改控制器面板中，可添加的动作**ControllerAction**，动作类型均来自其子类。



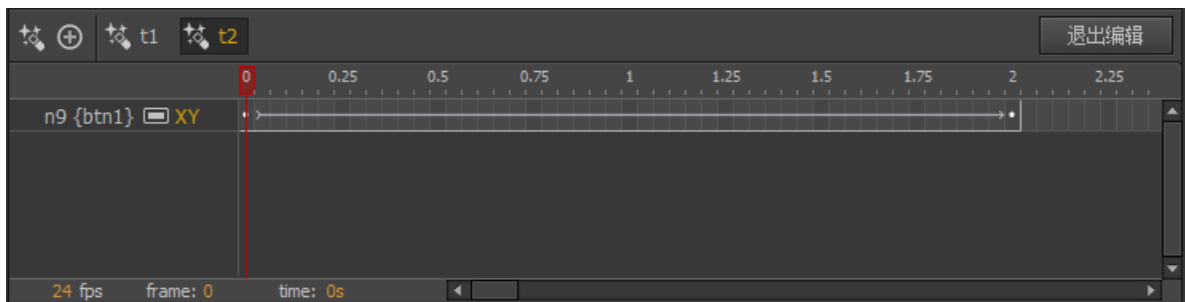
播放动效，相关代码在**PlayTransitionAction**。

改变其他控制器页面，相关代码在**ChangePageAction**。

动效

Transition

一个动效，就是一个Transition类的实例。




























Transition类如下：





Transition

类






















字段

-  _autoPlay
-  _autoPlayDelay
-  _autoPlayTimes
-  _delayedCallDelegate
-  _delayedCallDelegate2
-  _endTime
-  _ignoreEngineTimeScale
-  _items
-  _onComplete
-  _options
-  _owner
-  _ownerBaseX
-  _ownerBaseY
-  _paused
-  _playing
-  _reversed
-  _startTime
-  _timeScale
-  _totalDuration
-  _totalTasks
-  _totalTimes
-  invalidateBatchingEveryFrame
-  OPTION_AUTO_STOP_AT_END
-  OPTION_AUTO_STOP_DISABLED
-  OPTION_IGNORE_DISPLAY_CONTROLLER

属性

-  ignoreEngineTimeScale
-  name
-  playing
-  timeScale

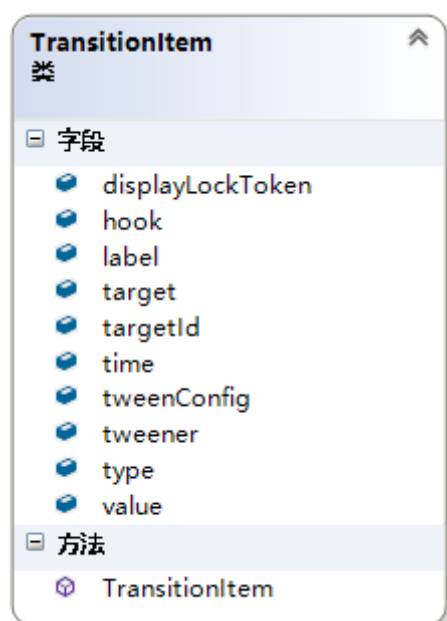
方法

-  _Play
-  ApplyValue
-  CallHook
-  ChangePlayTimes
-  CheckAllComplete
-  ClearHooks
-  DecodeValue
-  Dispose
-  GetLabelTime
-  InternalPlay
-  OnDelayedPlay
-  OnDelayedPlayItem
-  OnOwnerAddedToStage
-  OnOwnerRemovedFromStage
-  OnPlayTransCompleted
-  OnTweenComplete
-  OnTweenStart
-  OnTweenUpdate
-  Play (+ 3 重载)
-  PlayItem
-  PlayReverse (+ 2 重载)

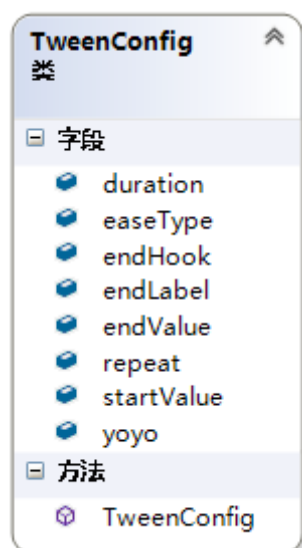
- SetAutoPlay
- SetDuration
- SetHook
- SetPaused
- SetTarget
- Setup
- SetValue
- SkipAnimations
- Stop (+ 1 重载)
- StopItem
- Transition
- UpdateFromRelations

TransitionItem

在fairy编辑器的动效编辑窗口中，每个图层都对应一个TransitionItem类的实例，记录在Transition._items字段中。



TransitionItem的tweenConfig字段为TweenConfig类型，记录了其缓动数据。



TransitionItem的value字段为object类型，在构造时，会根据其[TransitionActionType]type字段，为其创建不同类型的数据。

TValue
类

字段

- b1
- b2
- f1
- f2
- f3
- f4

属性

- color
- vec2
- vec4

方法

- Copy
- TValue

TValue_Animation
类

字段

- flag
- frame
- playing

TValue_Shake
类

字段

- amplitude
- duration
- lastOffset
- offset

TValue_Sound
类

字段

- audioClip
- sound
- volume

TValue_Text
类

字段

- text

TValue_Visible
类

字段

- visible

TValue_Transition
类

字段

- playCompleteDele...
- playTimes
- stopTime
- trans
- transName

关联
