Unity编辑器开发

编辑器相关文件夹介绍

1. `Editor`
    - 该文件夹可以放在项目的任何文件夹下，可以有多个`Editor`文件夹
    - 编辑器扩展相关的脚本都要放在该文件夹内，该文件夹中的脚本只会对Unity编辑器起作用
    - 项目打包的时候，不会被打包到项目中。如果编辑器相关脚本不放在该文件夹中，打包项目可能会出错
    - 如果非要有些编辑器相关脚本不放在该文件夹中，需要在该类的前后加上`UNITY_EDITOR`的宏定义
2. `Editor Default Resources`
    - 该文件夹需要放在`Assets`根目录下，用来存储编辑器所需要的图片等资源，书写的时候需要注意中间有空格隔开。此文件夹也不会被打包，访问方法为：`EditorGUIUtility.Load()`
    - 当然，也可以在`Editor`文件夹内创建一个`Resources`文件夹，将相关资源放在该文件夹内，通过`Resources.Load()`获取资源，也是可以的
3. `Gizmos`
    - 该文件夹也需要放在`Assets`根目录下，可以用来存放`Gizmos.DrawIcon()`的图片资源

编辑器拓展的命名空间

命名空间

- `Using UnityEditor ;`

相关的类

- **Editor**：
    - 这个类用于扩展一些编辑器全局的功能.且这个类可以和目标对象作用`[CustomEditor(MyScript)];`在`Editor`类里,可以重写`OnInspectorGUI(这个用的比较多的)`等方法。
- **EditorWindow**：
    - 这个类，用于自定义一个窗口，你可以为窗口添加一些按钮，选项等。比如，如果你想弄一个任务编辑器，用于配置一些数据。那么直接可以使用该类
    - 它有一个`OnSceneUI`的事件可以监听，可以使你实现像自己的编辑器一样，在`Scene`中点击某个物体后，在鼠标位置显示一个菜单，或者一些操作按钮
    - 还有一个比较适用的事件是`OnHierarchyChange`。当一个对象的父物体被改变，或者被新建的时候，这个事件会被触发
- **EditorApplication**： **主应用程序类**
    - 这个类提供了许多变量的访问，同时提供了`Save`等方法。比如，你可以新建一个编辑器脚本，在它的`update`函数里，进行记事，已做定时保存，从而避免不必要的损失
    - 这个类还提供了新建场景，打开项目等操作
    - 如果你是在界面下使用Unity3D，可能这个类的意义不大。但如果你是基于U3D的命令行来构建一个一键式多平台发布方案，那么这个类的地位就举足轻重了
- **EditorUtility**：

- 提供了很多全局函数，并且多半是静态的。比如，你想弹出一个打开文件的对话框，或者保存文件的对话框。或者你要查看场景树中的某个对象`enable`是与否，都可以通过它来访问

特性

- Unity特性总览
- 常用特性
- `P.S. 多个特性可以用逗号隔开，例如：[SerializeField, Range(0,5)]`

编辑器开发相关特性

MenuItem添加菜单栏按钮

- `[MenuItem]`： 添加菜单栏按钮，`[MenuItem("MyTools/test1",false,priority)]`
  - 第一个参数用来表示菜单的路径
  - 第二个参数用来判断是否是有效函数，是否需要显示
  - 第三个参数priority是优先级，用来表示菜单按钮的先后顺序，默认值为1000。一般菜单中的分栏，数值相差大于10
- 注意需要是静态方法
- 实现点击菜单按钮，删除场景或者Project中选中的多个对象

```
[MenuItem("MyTool/DeleteAllObj", true)]
private static bool DeleteValidate()
{
    if (Selection.objects.Length > 0)
        return true;
    else
        return false;
}

[MenuItem("MyTool/DeleteAllObj",false)]
private static void MyToolDelete()
{
    //Selection.objects 返回场景或者Project中选择的多个对象
    foreach (Object item in Selection.objects)
    {
        //记录删除操作，允许撤销
        Undo.DestroyObjectImmediate(item);
    }
}
```

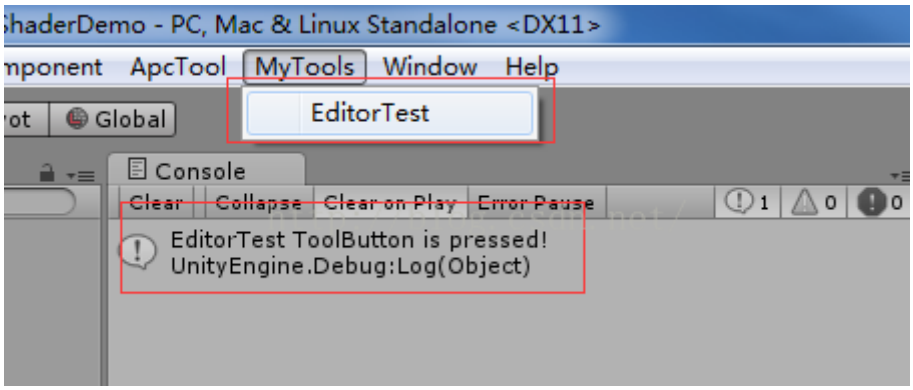
> `DeleteValidate`方法是`MyToolDelete`方法的有效函数，所以第二个参数为`true`。该有效函数用来判断当前是否选择了对象，如果选择了，返回`true`，才可以执行`MyToolDelete`方法。

- 添加快捷键

符号	字符
%	Ctr/Command
#	Shift
&	Alt
LEFT/Right/UP/DOWN	方向键
F1-F2	F功能键
_g	字母g

    - 例如：`[MenuItem("MyTools/test1 %_q")]` 快捷键 `Ctrl+Q`

- ```
  using UnityEngine;
  using UnityEditor;
  using System.Collections;

  public class EditorTest
  {
      [MenuItem("MyTools/EditorTest")]
      public static void ConfigDialog()
      {
          Debug.Log("EditorTest ToolButton is pressed!");
      }
  }
  ```

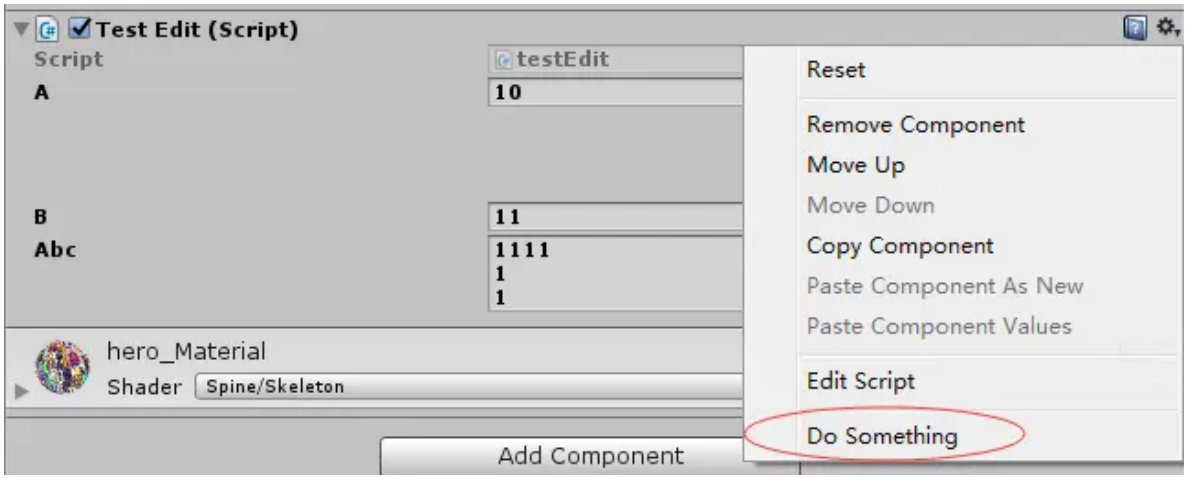


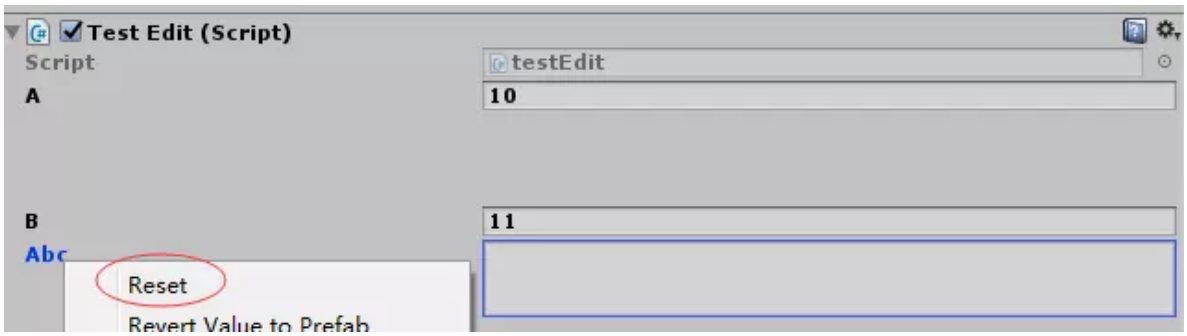
CONTEXT 给某组件添加右键菜单选项

- `[MenuItem("CONTEXT/组件名/按钮名")]`： 给某组件添加右键菜单选项
- 注意 `CONTEXT` 大写

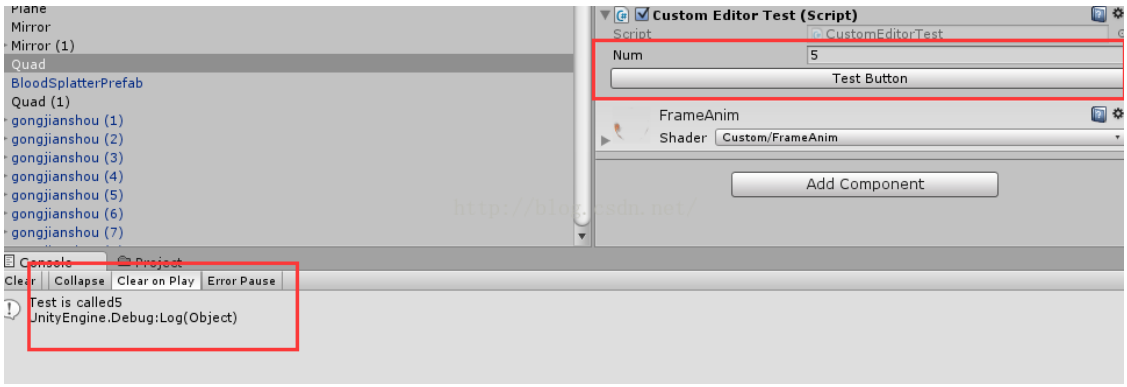
```
[MenuItem("CONTEXT/Rigidbody/Init")]
private static void RigidbodyInit()
{
    //TODO
}
```

MenuCommand 用于获取当前操作的组件

- 如下，给自定义的组件 `PlayerHealth` 添加右键 `Init` 按钮
```

```
[MenuItem("CONTEXT/PlayerHealth/Init")]
static void Init(MenuCommand cmd)
{
    PlayerHealth health = cmd.contex as PlayerHealth;
}
```

ContextMenu 给某组件添加右边小齿轮菜单选项

- 给某组件添加右边小齿轮菜单选项
- 标记函数：在`Inspector`面板，右击包含这条标记的脚本，出现"菜单名"的菜单选项。
- 向`Inspector`面板中脚本`Script`的上下文菜单（快捷，右键），添加一条指令，当选择该命令时，函数会执行。
- 标记的函数可以添加 MenuCommand cmd 参数，cmd.context转换为当前组建类型后操作

```
public class ContextTesting : MonoBehaviour
{
    [ContextMenu ("Do Something")]
    void DoSomething ()
    {
        Debug.Log ("Perform operation");
    }
}
```



ContextMenuItem 给某属性添加右键菜单选项

- 给某属性添加右键菜单选项

```
[ContextMenuItem("Handle", "HandleHealth")]
public float health;

private void HandleHealth()
{
    //ToDo
}
```

> P.S: 这两个特性是在`UnityEngine`命名空间下的，而不像其他`[MenuItem]`、`Selection`是在`UnityEditor`下的。



Selection 用于获取选择的游戏物体

- `Selection.activeGameObject` 返回第一个选择的场景中的对象
- `Selection.gameObjects` 返回场景中选择的多个对象，包含预制体等
- `Selection.objects` 返回选择的多个对象

```
//遍历选择的对象，并立刻销毁
foreach(object obj in Selection.objects)
{
    DestroyImmediate(obj);
}
```

> P.S：`Destroy` 方法会将删除的对象放在缓存中，缓存满了，才完全删除，而在编辑器未运行的时候，是没有这片缓存的，所以需要用 `DestroyImmediate()`，立刻销毁。当然，可以直接使用 `Undo.DestroyObjectImmediate()` 来销毁对象并记录销毁操作

CustomEditor 对对象界面的扩展

- `[CustomEditor(YouScript)]`
- 用于对对象界面的扩展，比如一个对象的属性和方法
- 创建一个 `ExampleEditor` 脚本，在类上添加 `[CustomEditor(typeof(T))]` 属性，重写 `OnInspectorGUI` 方法，用于扩展 `Inspector`

```
using UnityEditor;
using UnityEngine;

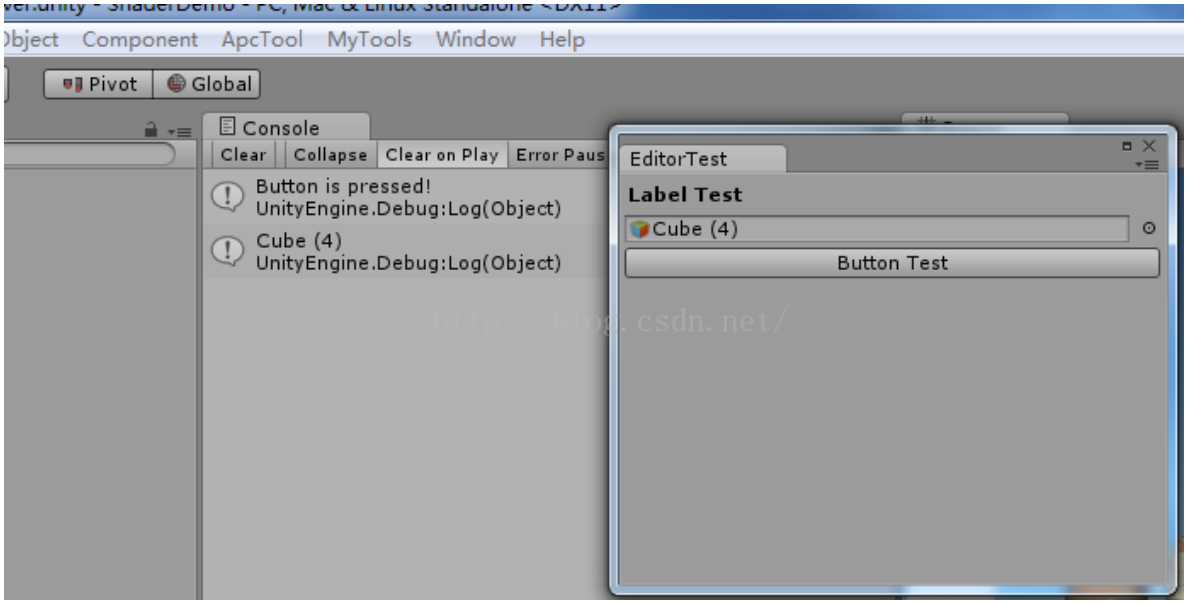
[CustomEditor(typeof(Example))]
public class ExampleEditor : Editor
{
    public override void OnInspectorGUI()
    {
        base.OnInspectorGUI();
        Example _example = target as Example;
        if (GUILayout.Button("执行Example方法"))
        {
            _example.LogError();
        }
        //扩展Inspector
    }
}
```

> `target` 就是你所添加的类型 `T`，需要强转为你需要的类型。其他的做法与 `OGUI` 类似

- 创建`Example`脚本，继承`MonoBehaviour`，用来挂载到`Object`上

```
using UnityEngine;
[ExecuteInEditMode]
public class Example : MonoBehaviour
{
    public void LogError()
    {
        Debug.LogError("执行Example方法");
    }

}
```

> `Example`中添加你需要处理的数据，或者函数。这样挂载`Example`在`GameObject`上，`Inspector`就可以查看和执行相关的函数了。如果你需要在编辑模式上运行周期函数，还可以在`Example`上添加`[ExecuteInEditMode]`属性

- 上述两个结合起效果如下



- 创建一个对话框
    - 如果我们嫌这个一个下拉菜单不爽，那我们就可以创建一个对话框，然后就可以在对话框里放更多的按钮或者一些工具条来调节相关内容，

```
using UnityEngine;
using UnityEditor;
using System.Collections;

//typeof(编辑器类名),继承EditorWindow
[CustomEditor(typeof(EditorTest))]
public class EditorTest : EditorWindow
{
    //通过MenuItem按钮来创建这样的一个对话框
    [MenuItem("MyTools/EditorTest")]
    public static void ConfigDialog()
    {
        //GetWindow创建
        EditorWindow.GetWindow(typeof(EditorTest));
    }

    public UnityEngine.Object go = null;

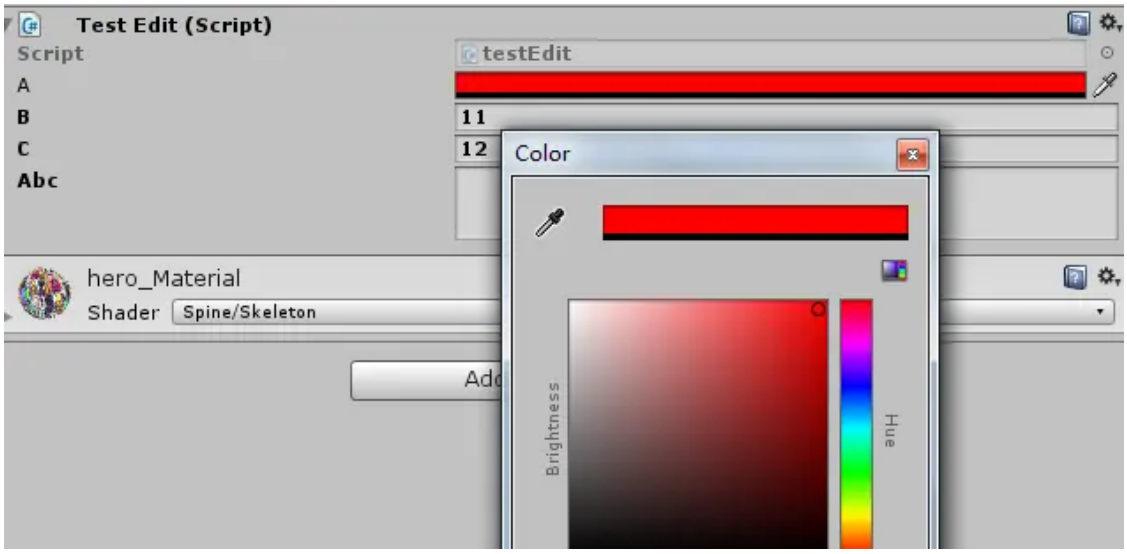
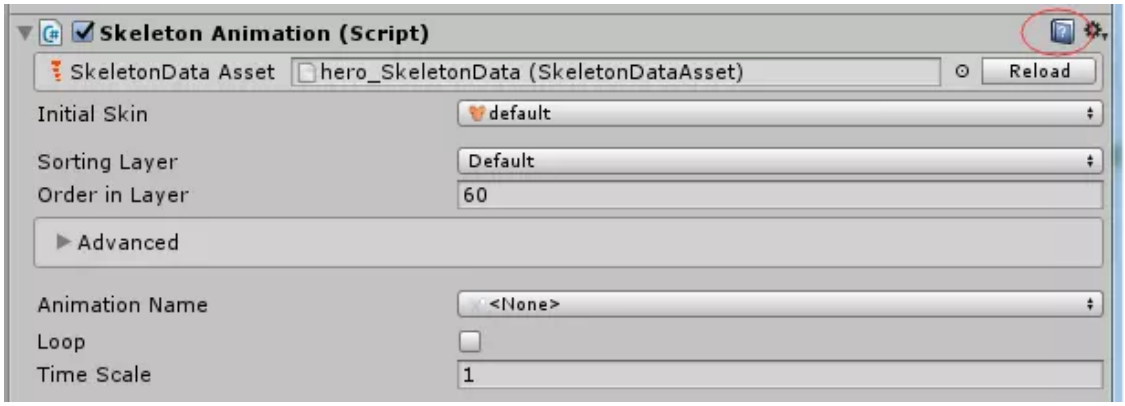
    //对话框中的各种内容通过OnGUI函数来设置
    void OnGUI()
    {
        //Label
        GUILayout.Label("Label Test", EditorStyles.boldLabel);
```

```
        //通过EditorGUILayout.ObjectField可以接受Object类型的参数进行相关操作
        go = EditorGUILayout.ObjectField(go, typeof(UnityEngine.Object), true);
        //Button
        if (GUILayout.Button("Button Test"))
        {
            Debug.Log(go.name);
        }
    }

}
```



ExecuteInEditMode

- `[ ExecuteInEditMode ]` 让`MonoBehaviour`脚本的所有实例，在编辑模式下可运行。
- 我们通常书写的脚本，并不会定义`[ExecuteInEditMode]`这个`Attribute`，所以`Awake()`和`Start()`都只有在`Runtime`中才会执行。给脚本添加`[ExecuteInEditMode]`特性可以在编辑器模式运行，在`ExecuteInEditMode`下两个方法的调用顺序和区别：
  - 在该`MonoBehaviour`在编辑器中被赋予给`GameObject`时，`Awake`和`Start`方法会被执行
  - 在`Play`按钮被按下开始之后，`Awake`和`Start`方法将被执行
  - 在`Play`按钮被抬起之后，`Awake`和`Start`方法将被执行
  - 在打开含有该`MonoBehaviour`脚本的场景的时候，`Awake`和`Start`将被执行
- 只有窗口在发生改变，接触新的事件，重绘后才会调用

```
using UnityEngine;

[ExecuteInEditMode]
public class PrintAwake : MonoBehaviour
{
    void Awake()
    {
        Debug.Log("Editor causes this Awake");
    }

    void Update()
    {
        Debug.Log("Editor causes this Update");
    }
}
```

AddComponentMenu添加组件菜单项

- 把添加Script的操作放在`Component`菜单下，来替代`Component/Scripts`，因为里面的脚本可能非常多，基本上没有实用价值
- `[AddComponentMenu("")]`
- 在编辑器添加一个用于添加组件的菜单项，将拥有该属性的脚本添加到选中的物体上

```
[AddComponentMenu("Example/testEdits")]
public class testEdit : MonoBehaviour
```

CreateAssetMenu /Creat 菜单下新建一个自定义子菜单

- `[CreateAssetMenu(menuName = “MySubMenue/Create XXX “)]`
- 标记类，可以给project面板下的Creat 菜单下新建一个自定义子菜单，用于新建自定义资源
- 快速的创建`ScriptableObject`派生类的实例，并存储成以`“.asset"`结尾的文件，`ScriptableObject`的派生类可以存储为外部的文件，图形化编辑对象数据，一些静态的数据，动态的加载，`ScriptableObject`是一种解决方案

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(fileName = "xxxx",menuName = "xxx/xxx")]
public class testEdit : ScriptableObject {
    public int a = 10;
    public int b = 11;
    public int c = 12;
    [Multiline][ContextMenuItem("Reset", "ResetString")]
    public string abc;

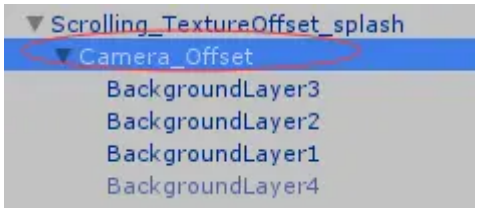
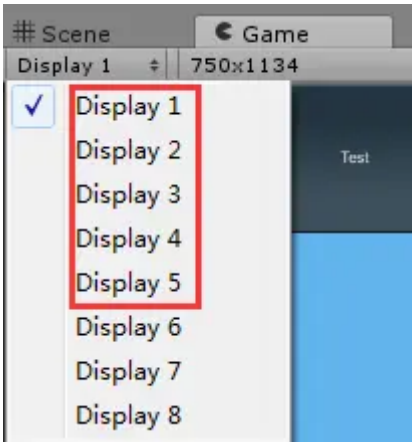
}
```

> 说明:
> fileName:生成asset文件的文件名。
> menuName:在Assets/Create上子菜单的名字。

ColorUsage

- 颜色选择器，`color picker`，只能应用在`Color`字段上
- 默认参数为是否显示alpha，具体使用看下官方文档的参数描述，这里不加代码了



AssemblyIsEditor

- 添加该特性的任意类，都会被视为`Editor`编辑器类。只有用于`Editor`模式下

属性面板常用特性

DisallowMultipleComponent

- `[ DisallowMultipleComponent]`
- 对一个MonoBehaviour的子类使用这个属性，那么在同一个`GameObject`上面，最多只能添加一个该`Class`的实例。尝试添加多个的时候，会出现提示
- 如果当前GameObject已经存在了多个相同的组件，是不会有影响的，但应用了特性以后，就无法再次添加

size属性隔离特性

- `[size=9.7541pt][Header(“XXX”)]`
- 在`Inspector`面板上给定义的字段的上一行加段描述，可以将属性隔离开，形成分组的感觉

Tooltip

- `[Tooltip(“XXX”)]`
- 在`Inspector`面板上鼠标移上定义的字段弹出描述

HelpURL

- 从字面意思理解，是查看帮助时，跳转到指定的页面



- `[HelpURL("URL地址")]`

Range

- `[Range(min, max)]`
- 限制数值变量的取值范围并以滑动条显示在`Inspector`中

Delayed

- 在运行时，我们修改`Inspector`面板中的字段，会即时返回新的值，应用`Delayed`特性，只有在用户按下回车`Enter`或是焦点离开时才会返回新值，通过用于调试阶段
- 只能应用于字段，不可在类或其它目标元素上使用

Space

- `[Space(50)]`间隔距离，在`Inspector`中，可以设置元素与元素之间的间隔。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class testEdit : MonoBehaviour {
    public int a = 10;
    [Space(50)]
    public int b = 11;
}
```

TextArea

- 可以在一个高自由度并且可滚动的文本区域里编辑`string`字符串，如果字符串比较长，比较适用
- 参数:
  - minLines:文本区域最小行数
  - maxLines:文本区域最大行数，超过最大行数，会出现滚动条

```
[TextArea(1,5)]
public string abc;
```

Multiline

- 在一个支持多行的文本区域内编辑`string`字符串，他和`TextArea`不同，`Multiline`的`TextArea`没有滚动条

```
[Multiline(1,5)]
public string abc;
```

Header

- 在`Inspector`面板中，为`field`字段添加头信息，增强描述

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour {
    [Header("Health Settings")]
    public int health = 0;
    public int maxHealth = 100;
    [Header("Shield Settings")]
    public int shield = 0;
    public int maxShield = 0;
}
```

DrawGizmo

- `[DrawGizmo]`
- 用于Gizmos渲染，将逻辑与调试代码分离

序列化特性

HideInInspector

- `[HideInInspector]`
- 使属性在`Inspector`中隐藏，但是还是可序列化，想赋值可以通过写程序赋值序列化

Serializable

- `[Serializable]`
- 使自定义的类或结构体能进行序列化，即当做一个`public`成员的时候可以在`Inspector`显示

SerializeField

- 强制unity去序列化一个私有域
- 这是一个内部的unity序列化功能，有时候我们需要`Serialize`一个`private`或者`protected`的属性，这个时候可以使用`[SerializeField]`

NonSerialized

- 标记一个变量或方法不会被序列化

脚本常用特性

RequireComponent 自动添加所要依赖的组件

- 自动添加所要依赖的组件，如将一个`Script`做为一个`GameObject`的组件，而这个`Script`需要访问`Rigidbody`组件，通过应用该属性，可以自动的添加`Rigidbody`组件到当前的`GameObject`中，避免设置错误
- 如果当前的`Script`已经添加到了`GameObject`，这时候你再应用`RequireComponent`特性是无效的， 你必须删除 再重新添加，才会检测

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody))]
public class PlayerScript : MonoBehaviour
{
    Rigidbody rb;
```

```
    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    void FixedUpdate()
    {
        rb.AddForce(Vector3.up);
    }
}
```

RuntimeInitializeOnLoadMethod

- 在运行时，当前类初始化完成，自动调用被该特性应用的静态函数,这和`static`静态构造函数还不一样，`static`静态构造函数是在所有的方法之前运行的，而`RuntimeInitializeOnLoadMethod`特性的方法是`Awake`方法之后执行的(如果是`MonoBehaviour`派生类)
- 如果一个类中有多个静态方法使用了`RuntimeInitializeOnLoadMethod`特性，执行顺序是不固定的

```
[RuntimeInitializeOnLoadMethod]
    static void OnRuntimeMethodLoad()
    {
        Debug.Log("After scene is loaded and game is running");
    }

    [RuntimeInitializeOnLoadMethod]
    static void OnSecondRuntimeMethodLoad()
    {
        Debug.Log("SecondMethod After scene is loaded and game is
running.");
    }
```

SelectionBase

- 设置基础选中对象，应用该标识一个对象为选中对象，当我们在`scene view`中选择一个`objects`的时候，`u3d`会返回给我们是适合的对象，比如你选中的对象是`prefab`的一部分，默认会返回节点的根对象，默认根对象被设置成了基础选中对象，你可以修改他，让其它的对象成为基础选中对象，比如根对象可能就是一个空的`GameObject`，而我们要实际查看编辑的对象是子节点，这样我们可以将子节点中添加的脚本应用`SelectionBase`特性。



- 将脚本加到`Camera_Offset`后，成为了默认的选中对象，这样每次我在场景中选中时，`Camera_Offset`会被选择，并高亮显示。

GUITarget

- 设置`OnGUI`方法在哪一个`Display`下显示，默认是所有的`Display`均显示

```
[GUITarget(0,1,new int[]{2,3,4})]
    void OnGUI()
    {
        if (GUI.Button (new Rect (0, 0, 128, 128), "Test")) {
            Debug.Log ("blahblahblah....");
        }
    }
```



> 说明：
> 提供了如下参数：
> displayIndex　Display index.display 索引
> displayIndex1　Display index. display索引
> displayIndexList　Display index list.display索引列表

自定义属性特性

```
using UnityEngine;

//定义特性
public class ShowTimeAttribute : PropertyAttribute
{
   public readonly bool ShowHour;

   //定义构造函数
   public ShowTimeAttribute(bool isShowHour = false)
   {
       ShowHour = isShowHour;
   }
}
```

```
using UnityEngine;
using UnityEditor;

//用于绘制特性，该类需要放到Editor中
```

```
[CustomPropertyDrawer(typeof(ShowTimeAttribute))]
public class TimeDrawer : PropertyDrawer
{
    //设置绘制的区域高度
    public override float GetPropertyHeight(SerializedProperty property,
GUIContent label)
    {
        return EditorGUI.GetPropertyHeight(property) * 2;
    }

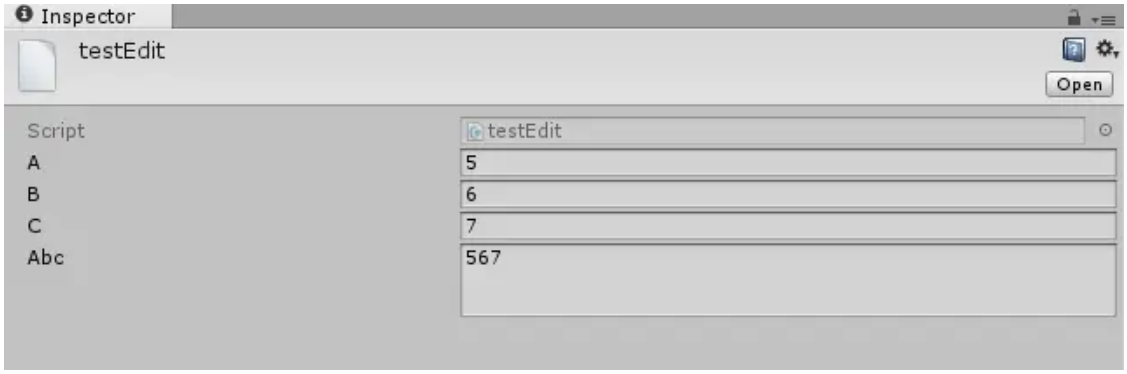
    public override void OnGUI(Rect position, SerializedProperty property,
GUIContent label)
    {
        if (property.propertyType == SerializedPropertyType.Integer)
        {
            property.intValue = EditorGUI.IntField(new Rect(position.x,
position.y, position.width, position.height / 2), label, Mathf.Max(0,
property.intValue));

            EditorGUI.LabelField(new Rect(position.x, position.y +
position.height / 2, position.width, position.height / 2), "",
TimeConvert(property.intValue));
        }
        else
        {
            EditorGUI.HelpBox(position, "To use the Time Atribute," +
label.ToString() + "must be int", MessageType.Error);
        }
    }

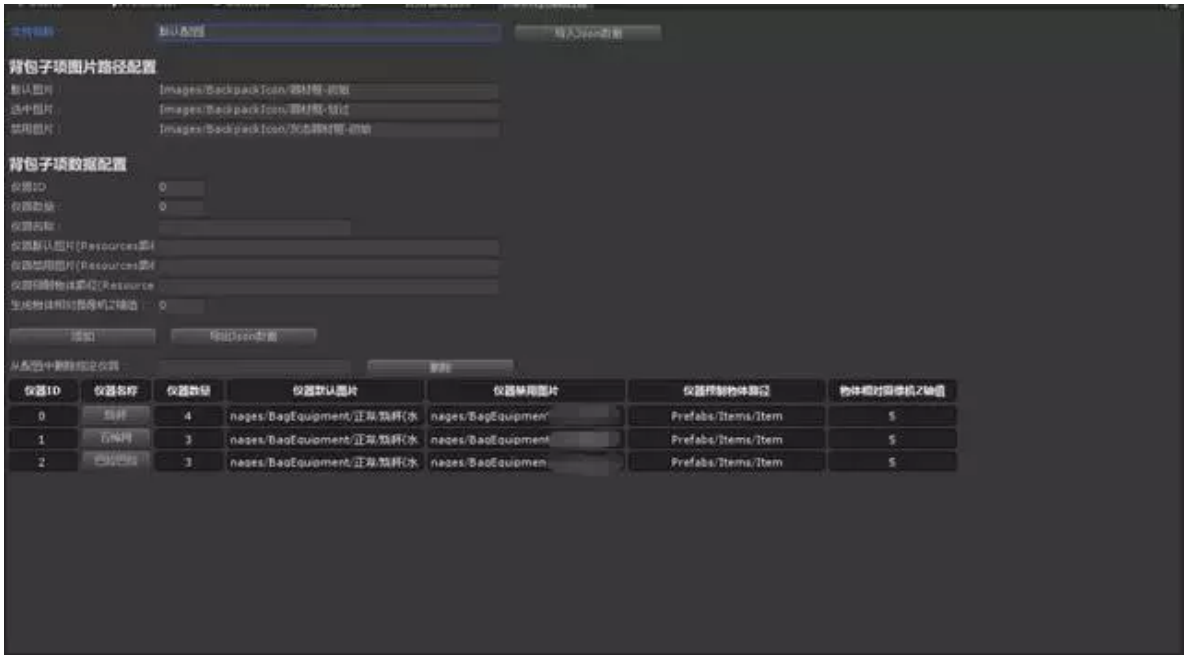
    private string TimeConvert(int value)
    {
        ShowTimeAttribute time = attribute as ShowTimeAttribute;
        if (time != null)
        {
            if (time.ShowHour)
            {
                int hours = value / (60 * 60);
                int minutes = (value % (60 * 60)) / 60;
                int seconds = value % 60;

                return string.Format("{0}:{1}:{2}(H:M:S)", hours,
minutes.ToString().PadLeft(2, '0'), seconds.ToString().PadLeft(2, '0'));
            }
        }
        else
        {
            int minutes = (value % (60 * 60)) / 60;
            int seconds = value % 60;

            return string.Format("{0}:{1}(M:S)", minutes.ToString().PadLeft(2,
'0'), seconds.ToString().PadLeft(2, '0'));
        }
        return string.Empty;
    }
}
```

```
//测试
public class Test : MonoBehaviour
{
    [ShowTime(true)]
    public int time = 3605;
}
```

编辑器常用函数

- `AssetDatabase.CreateAsset`可以帮住你从资源目录中创建一个资源实例
- `Selection.activeObject`返回当前选中的对象
- `EditorGUIUtility.PingObject`用来实现在Project窗口中点击某一项的操作
- `Editor.Repaint`用来重绘界面所有的控件
- `XXXImporter`用来设置某种资源的具体导入设置（例如在某些情况下你需要设置导入的贴图为可读的）
- `EditorUtility.UnloadUnusedAssets`用于释放没有使用的资源，避免你的插件产生内存泄漏
- `Event.Use`用来标记事件已经被处理结束了
- `EditorUtility.SetDirty`用来通知编辑器数据已被修改，这样在下次保存时新的数据将被存储

关于ScriptableObject的使用

ScriptableObject类型

- `ScriptableObject`类型经常用于存储一些`unity3d`本身不可以打包的一些`object`，比如字符串，一些类对象等。用这个类型的子类型，则可以用`BuildPipeline`打包成`assetbundle`包供后续使用，非常方便
- 以`.aaset`资源文件的形式，保存在项目中
- 默认情况下，`protected`，`private`，`internal`变量将不会被`serialize`
- `[System.NonSerialized]`：有时候我们需要定义一些`public`变量方便操作，但是又不希望这些变量保留。这个时候可以利用`[System.NonSerialized]`来完成这个操作
- 如果变量加入了`readonly`，`const`，`static`等修饰符，无论他的`serialize`设置如何，都将不会进行`serialize`

创建实例

代码创建

- 通过代码来创建实例，并完成初始化，最后存储到本地，后缀为`".asset"`

```
testEdit test = testEdit.CreateInstance<testEdit> ();
        test.name = "testEdit";
        test.a = 5;
        test.b = 6;
        test.c = 7;
        test.abc =""+test.a + test.b + test.c;
        UnityEditor.AssetDatabase.CreateAsset (test,
"Assets/"+test.name+".asset");
```

- 通过添加`CreateAssetMenu`，在`Assets/Create`上进行快捷创建

```
[CreateAssetMenu(fileName = "xxxx",menuName = "xxx/xxx")]
public class testEdit : ScriptableObject {
    public int a = 10;
    public int b = 11;
    public int c = 12;
    [Multiline][ContextMenuItem("Reset", "ResetString")]
    public string abc;

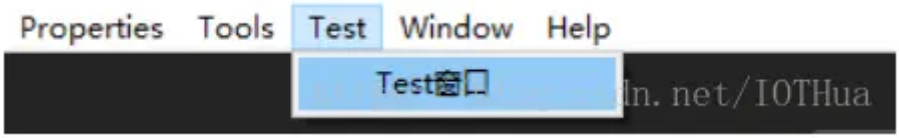
}
```

> 说明:
> fileName:生成asset文件的文件名。
> menuName:在Assets/Create上子菜单的名字。



- 创建后的资源文件可以直接图形化的设计对象参数的值。并保存。通过成后的`testEdit.asset`，可以设置`assetbudleName`来成`assetBundle`

读取实例信息

- 读取`testEdit.asset`，如果你设置成了`AssetBundle`，直接使用`AssetBundle`的方式加载即可，否则就放在`Resources`文件夹下，如下

```
testEdit t = (testEdit)Resources.Load ("testEdit");
        if (t != null) {
            Debug.Log (t.a + "," + t.b + "," + t.c + "," + t.abc);
        }
```

PreferBinarySerialization

- 只能用在`ScriptableObject`的派生类上（使用在其它类型上面会被忽略），修改序列化的模式为二进制，而不是`YAML`， 当你的资源比较大的时候，保存成二进制，生成的数据会更加的紧凑，从而提高程序的读写性能

```
[CreateAssetMenu]
[PreferBinarySerialization]
public class testEdit : ScriptableObject {

    public Color a;
    public int b = 11;
    public int c = 12;
    [Multiline][ContextMenuItem("Reset", "ResetString")]
    public string abc;

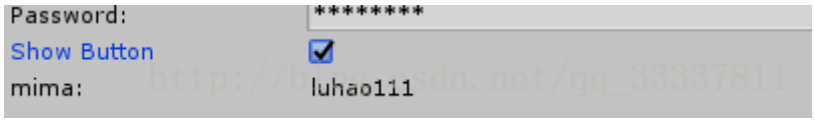
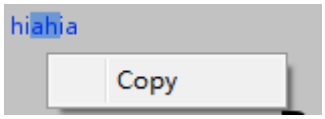
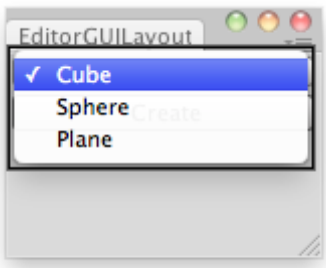
}
```

- 用记事本打开生成后的`asset`，会发现都是二进制的数据

说明

最后，`ScriptableObject`派生的类可以方便的存储成外部文件，并且以图形化的操作修改对象的属性数值。
比如一些静态的数据，如常量，关卡，任务，成就等等配置表，与将类序列化成字节流文件并运行时反序列化转换成对象的使用流程是一样的。
但如果说是用来做静态数据表，显然这是没有`EXCEL`里操作更为方便，非技术人员通常都喜欢用`EXCEL`来操作配置表，更灵活，功能更强大，自由度更高，且更方便使用，技术只要提供导出工具，不论是导出外部的数据文件，`xml`,`json`,`pb`，自定义等等，还是在导出的过程中，直接序列化成文件都可以，而且这些通常都有了比较成熟的解决方案，在unity引擎之前很多都是这么过来的。

我目前在项目中还没有使用过`ScriptableObject`的地方，有些功能确实是可以使用他来实现，但是已经有其它的解决办法了，本质上并没有什么区别。

听腾讯一位前端大拿说，不太推荐使用`ScriptableObject`，因为他的解析效率比较低，不及`pb`以及`Json`，但在开发编辑器时，可以把编辑器所需要的一些数据存下来，使用这种形式还是可以的

编辑器开发详细

窗口开发详细



- 创建一个`Editor`脚本
- 在`Editor`文件夹下新建一个类（`TestEditorWindows`），该类集成`EditorWindow`，还需要引用`UnityEditor`命名空间
- 创建类完毕之后，在编写如下代码

```
using UnityEngine;
using UnityEditor;

public class TestEditorWindows : EditorWindow
{
    //1.必须跟类型一样，这是窗口的名称
```

```
    private TestEditorWindows()
    {
        this.titleContent = new GUIContent("测试编辑器窗口");
    }
    //2.这是在哪里创建窗口

    [MenuItem("Test/Test窗口")]

    private static void CreateTestWindows()
    {
        EditorWindow.GetWindow(typeof(TestEditorWindows));
    }
}
```



- 目前窗口中什么都没有，因为我们还没开始写控件
- 如果需要在窗口中绘制控件，则需要在`OnGUI()`中去编写相关代码，在此之前我们需要了解`OnEnable()`方法

```
//窗口启动时，会调用此方法
private void OnEnable()
{
    //OnEnable()方法是一个比较重要的方法，在一般的窗口绘制中，可在这里进行相关数据的初始化。
}
//实时绘制相关控件
private void OnGUI()
{
    //OnGUI()方法则是实时的进行控件绘制，窗口控件绘制也就在这里进行代码编写。
}
```

- 在编写想要的窗口时，需要了解常用的控件，这些控件都是我们非常常用的。值得注意的是不管是在`Windows`窗口开发还是`Inspector`窗口开发，常用的控件基本都在`GUILayout`和`EditorGUILayout`这两个类中，有些控件两者都可以，不过一般都会用`EditorGUILayout`，因情况而定……所以别在开发中，网上的资料中有时用`GUILayout`，有时又用`EditorGUILayout`。

常用控件

- **GUILayout.BeginScrollView / GUILayout.EndScrollView**
    - 这个控件是添加窗口滚动条的，也就是给你这个窗口添加水平和垂直滚动条

```
Vector2 scrollPos = Vector2.zero;
private void OnGUI()
{
    //参数一：滚动坐标
    //参数二：窗口宽度
    //参数三：窗口高度
    //其中position是内置参数
    scrollPos = GUILayout.BeginScrollView(scrollPos,

 GUILayout.Width(position.width),

 GUILayout.Height(position.height));
    GUILayout.EndScrollView();
}
```

- **说明**：编写完毕之后，此时看窗口是没有滚动条的样式的，因为里面目前还没有任何的控件。在介绍后面两个控件时，我们需要说明下，在`Unity`编辑器中，如果我们要对控件进行控件的布局，比如有些控件水平显示，有些垂直显示。那么就需要应用后面的这两个控件，不过在一些真正的项目开发或者一些公司中，他们都会基于这两个布局控件，进行在封装一层，以方便灵活使用。在程序`UI`界中，比如`UGUI`、`GUI`、以及一些非`Unity`的`UI`控件插件等，都基本上有这种类似的布局。需要注意：这种布局控件，一般都是配对出现的，比如你用了`GUILayout.BeginVertical()`后，在后面你必须要有`GUILayout.EndVertical()`。否则窗口将绘制不出来并且提示错误提示。这点在编写复杂的窗口时，很重要，因为一不留神就出现错误。最好在编写代码的时候，将`BeginVertical`和`EndVertical`一起码出来，以便到时候去匹配

- **GUIContent：控件文字提示**
  - `GUIContent`也是一个非常常用的东西。给绘制的控件加别名与提示信息，在`Inspector`属性绘制中，有一些英文属性也许看名字不了解他是什么作用，但是如果用这个绘制出来，就可以很方便它的作用了。大部分控件都可以用这个，当然也有一些控件是不能用这个的
- **EditorGUILayout.PasswordField**
  - 密码字段
  - `EditorGUILayout.PasswordField("Password:",text);`



- **EditorGUILayout.IntField**
  - 整数字段
  - `EditorGUILayout.IntField("Number of clones:", clones);`
  - 只能输入整数，同时还有`float`字段
- **GUILayout.Button**
  - 当按下时，返回`true`。可在这里写自己的实现，同时可设置`Button`的宽高，利用`GUILayout.Width`和`GUILayout.Height`，也可以利用`GUIContent`给按钮添加提示，同时可利用`GUIStyle`给按钮添加样式
- **GUILayout.Box**
  - `Box`控件就是文字框/图片框了，指定一块框，其实跟`Button`差不多，只不过不能进行点击而已。
- **EditorGUILayout.LabelField**
  - `LabelField`控件，就是文本标签
- **EditorGUILayout.SelectableLabel**

- 可选择标签（通常用于显示只读信息，可以被复制粘贴）
- `EditorGUILayout.SelectableLabel(text);`



- **GUILayout.HelpBox**
  - 帮助信息
  - `HelpBox`类似`LabelField`标签，也是文字提示类的，但是它多了几个状态以及背景框，选择不同的状态显示不同的`UI`
  - `EditorGUILayout.HelpBox("这是提示信息",MessageType.Error,true);`
  - 

- **TextArea**
  - 文本输入框
  - `string info=GUILayout.TextArea(info,GUILayout.Width(500),GUILayout.Height(50));`
- **Toggle**
  - 单选框
  - `Toggle`控件一般用于接收`Bool`参数，比如你有一个`Bool`值参数，需要在窗口中显示出来，那么用`Toggle`可帮你接收到这个`Bool`值信息
- **Slider：进度条**
  - `Slider`控件，提供一个滑轮，指定最大值、最小值。那么可滑动来设置这些值，同时也可以手动设置，类似`MonoBehaviour`中的`Range`特性
- **EditorGUILayout.Popup**
  - `Popup`弹出选择菜单

```
//参数：label字段前面可选标签
//selectedIndex字段选项的索引
//displayedOptions弹出菜单选项的数组 style可选样式 options
//返回：int，用户选择的选项索引
index = EditorGUILayout.Popup(index, options);
```



- **EditorGUILayout.EnumPopup：枚举框**
  - 枚举控件也是在编写编辑器中，使用的比较多的，比如选择不同的枚举，绘制出来的控件都是不一样的
  - `(枚举)EditorGUILayout.EnumMaskPopup(new GUIContent("枚举值：","枚举的详细提示信息"),枚举变量);`
- **DropdownButton：下拉框**
  - 它的效果跟`EnumPopup`，不过`EnumPopup`是基于枚举类，序列化出来的，而`DropdownButton`是根据自定义添加子项
  - ```
    //外部String变量，用于接收下拉框值
    private string itemStr = "";

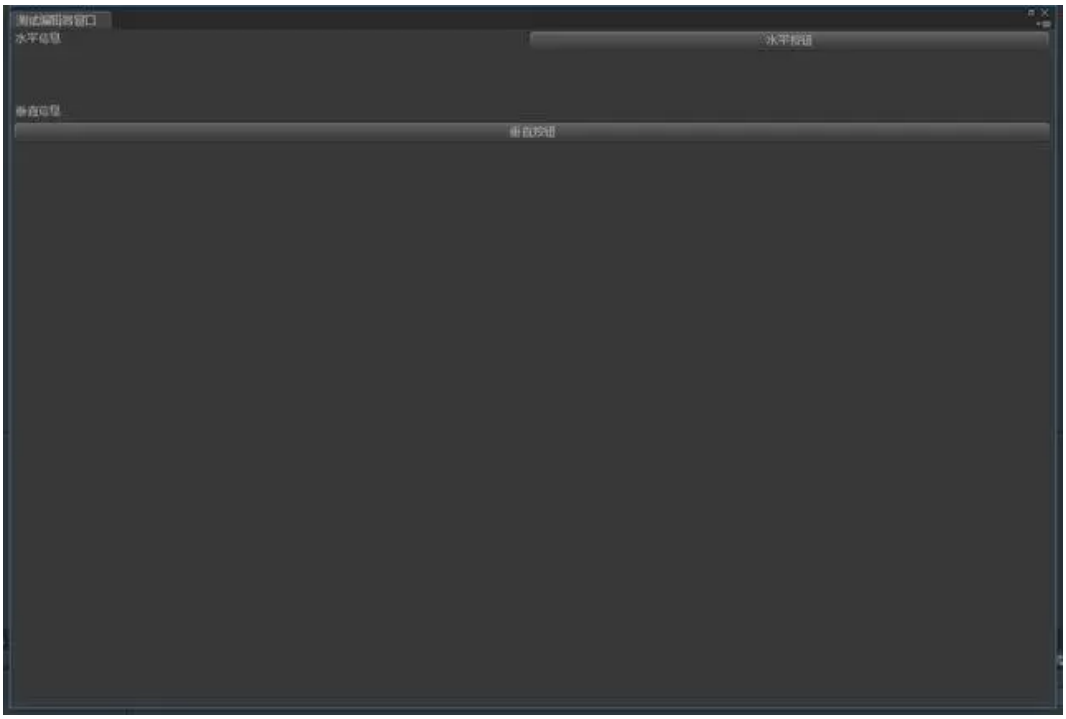
    //方法类的核心代码
    //其中new GUIContent(itemStr)，用于接收当前选中的值
    if (EditorGUILayout.DropdownButton(new GUIContent(itemStr), FocusType.Keyboard))
    {
        var alls = new string[4]{"A","B","C","D"};
        GenericMenu menu = new GenericMenu(); //实例化一个菜单
        //遍历字符串集合
        foreach (var all in alls)
         {
                if (string.IsNullOrEmpty(all)) continue;
                //添加子项
                AddMenuItemForValue(menu, all);
         }
         menu.ShowAsContext();//绘制出菜单
    }

    //添加子项
    void AddMenuItemForValue(GenericMenu menu, string value)
    {
        //添加子项的格式，itemStr.Equals(value)如果与目前相等的，就处于选中状态
        menu.AddItem(new GUIContent(value), itemStr.Equals(value),
               OnValueSelected, value);
    }

     //当选中某个值的时候，itemStr获取到选中值
    void OnValueSelected(object value)
    {
            itemStr = value.ToString();
    }
    ```
- **EditorGUILayout.ObjectField 序列化Object物体**
  - `EditorGUILayout.ObjectField`组件是用于显示一些针对继承`UnityEngine.Object`类的相关组件，比如`GameObject`、`Transform`、`Component`等相关组件，或者继承`MonoBehaviour`类的脚本。这个控件在编写编辑器时，也是经常会用到的一个东西
  - ```
    GameObject target=new GameObject();
    target=EditorGUILayout.ObjectField(new GUIContent("信息：","鼠标移入控件显示的提示信息"),target,typeof(GameObject),true) as GameObject;
    ```
- **GUILayout.BeginVertical/GUILayout.EndVertical**
  - 该控件是垂直布局控件，也就是说在这个区域内的控件，都将垂直排列
- **GUILayout.BeginHorizontal/GUILayout.EndHorizontal**
  - 该控件是水平布局控件，在这个区域内的控件，都将水平排列
  - ```
    //水平布局控件
    //在这里在来一个小技巧，给布局添加背景样式，以及定义宽高，这时候就可以显示窗口滚动条了。
    GUILayout.BeginHorizontal("box", GUILayout.Width(500));
    GUILayout.Label("水平信息");
    if (GUILayout.Button("水平按钮")) { }
    GUILayout.EndHorizontal();

    //空行
    //如果是在垂直布局内，就是垂直50单位，反之如果在水平布局内就是水平空50单位
    ```
    ```

```
GUILayout.Space(50);

//垂直布局控件
GUILayout.BeginVertical("box", GUILayout.Width(500),
GUILayout.Height(200));
GUILayout.Label("垂直信息");
if (GUILayout.Button("垂直按钮")) { }
GUILayout.EndVertical();
```

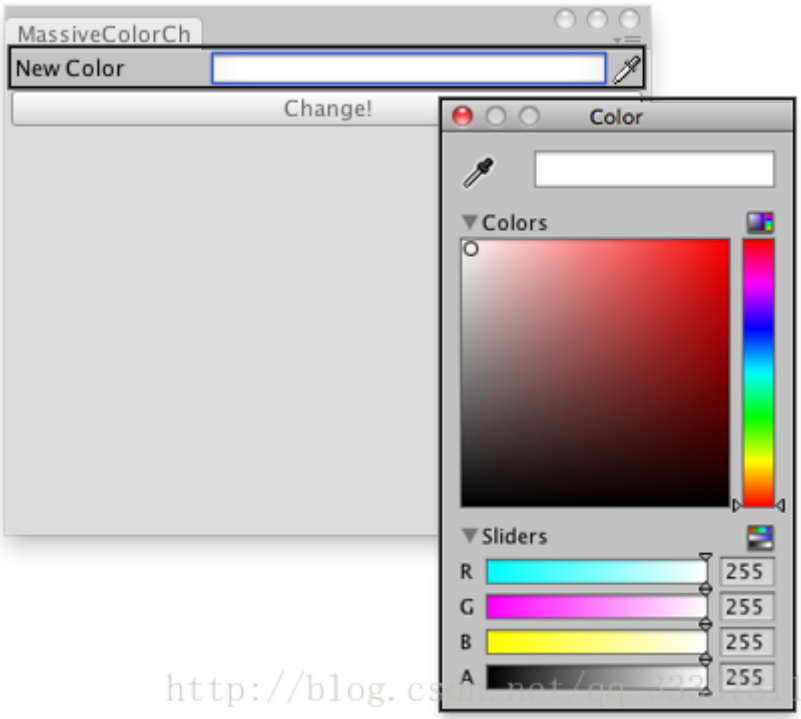


- 当窗口缩放到代码指定的控件宽高时，窗口会自动显示滚动条，当然前提是你在最开始前布局了`GUILayout.BeginScrollView/GUILayout.EndScrollView`。

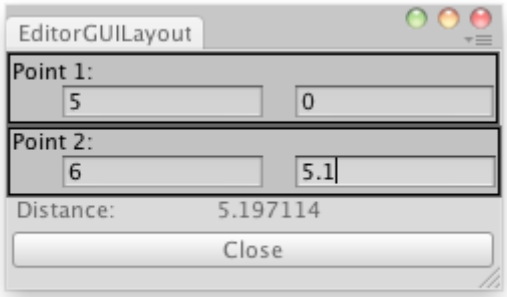
- **GUIStyle控件样式**
  - `GUIStyle`则是控件的样式，比如`Label`的字体大小，颜色等等。一般采用系统默认的，这个根据自身情况而定采用自定义的
- **GUILayout.Toolbar**
  - `Toolbar`工具栏

```
int m_SelectedPage=0;
string[] m_ButtonStr=new string[4]{"Combine Animation","Check
Part","Create RootMotion","CheckunUsedPrefab"};
void OnGUI()
{
     m_SelectedPage=GUILayout.Toolbar(
         m_SelectedPage,
         m_ButtonStr,
         GUILayout.Height(25));
}
```

- **EditorGUILayout.ColorField**
  - 颜色字段
  - `matColor = EditorGUILayout.ColorField("New Color", matColor);`



- **EditorGUILayout.Vector2Field**
  - 二维向量字段
  - `p1 = EditorGUILayout.Vector2Field("Point 1:", p1);`
  - 三维的一样



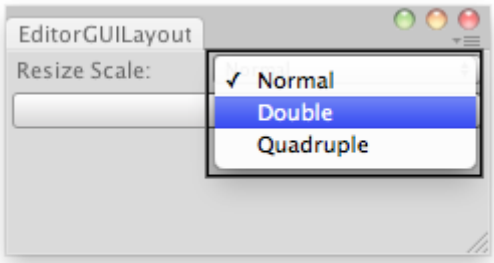
- **EditorGUILayout.TagField/LayerField**
  - 标签
  - `EditorGUILayout.TagField("Tag for Objects:", tagStr);`

```
 string tagStr = "";
int selectedLayer=0;
void OnGUI()
{  //为游戏物体设置
    tagStr = EditorGUILayout.TagField("Tag for Objects:", tagStr);
    if (GUILayout.Button("Set Tag!"))
        SetTags();
    if(GUILayout.Button("Set Layer!"))
        SetLayer();
}
void SetTags() {
    foreach(GameObject go in Selection.gameObjects)
        go.tag = tagStr;
}
void SetLayer() {
    foreach(GameObject go in Selection.gameObjects)
```

```
            go.laye = tagStr;
    }
```

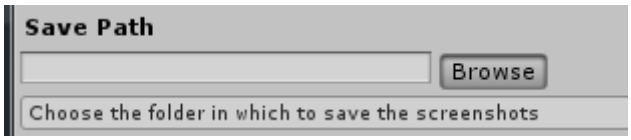
- **EditorGUILayout.IntPopup**
  - 整数弹出选择菜单
  - `EditorGUILayout.IntPopup("Resize Scale: ", selectedSize, names, sizes);`

```
 int selectedSize = 1;
string[] names = { "Normal","Double","Quadruple"};
int[] sizes = { 1,2,4};
void OnGUI()
{
    selectedSize = EditorGUILayout.IntPopup("Resize Scale: ", selectedSize,
names, sizes);
    if (GUILayout.Button("Scale"))
        ReScale();
}
void ReScale()
{
    if (Selection.activeTransform)
        Selection.activeTransform.localScale =new Vector3(selectedSize,
selectedSize, selectedSize);
    else Debug.LogError("No Object selected, please select an object to
scale.");
}
```



- **EditorUtility.SaveFolderPanel**
  - 打开保存位置文件夹

```
GUILayout.Label ("Save Path", EditorStyles.boldLabel);
EditorGUILayout.BeginHorizontal();
EditorGUILayout.TextField(path,GUILayout.ExpandWidth(false));
if(GUILayout.Button("Browse",GUILayout.ExpandWidth(false)))
            path = EditorUtility.SaveFolderPanel("Path to Save
Images",path,Application.dataPath);   //打开保存文件夹面板
EditorGUILayout.EndHorizontal();
```



- **Foldout**
  - 制作一个左侧带有箭头的折叠标签
  - `bool Foldout(bool value, string label)`
- **SelectionGrid**

- 选择网格
- `SelectionGrid(int 选择的索引,sting[] 显示文字数组，xCount，格式)`

```
Vector2 v2 = new Vector2(0,0);
Int32 v = 0;
string[] str = { "Message1", "Message2", "Message3", "Message4" };

GUIStyle textStyle = new GUIStyle("textfield");
GUIStyle buttonStyle = new GUIStyle("button");
textStyle.active = buttonStyle.active;
textStyle.onNormal = buttonStyle.onNormal;

v2 = GUILayout.BeginScrollView(v2, true, true, GUILayout.Width(300),
GUILayout.Height(100));
{
    v = GUILayout.SelectionGrid(v,str,1,textStyle);
}
GUILayout.EndScrollView();
```

- **GetControlRect**
  - 控制区域

```
//通过拖拽获取文件路径
string path;
Rect rect;
void OnGUI()
{
    EditorGUILayout.LabelField("路径");
    //获得一个长300的框
    rect = EditorGUILayout.GetControlRect(GUILayout.Width(300));
    //将上面的框作为文本输入框
    path = EditorGUI.TextField(rect, path);

    //如果鼠标正在拖拽中或拖拽结束时，并且鼠标所在位置在文本输入框内
    if ((Event.current.type == EventType.DragUpdated
         || Event.current.type == EventType.DragExited)
        && rect.Contains(Event.current.mousePosition))
    {
        //改变鼠标的外表
        DragAndDrop.visualMode = DragAndDropVisualMode.Generic;
        if (DragAndDrop.paths != null && DragAndDrop.paths.Length > 0)
        {
            path = DragAndDrop.paths[0];
        }
    }
}
```

- 生命周期

```
//更新
void Update()
{

}

void OnFocus()
{
    Debug.Log("当窗口获得焦点时调用一次");
}

void OnLostFocus()
{
    Debug.Log("当窗口丢失焦点时调用一次");
}

void OnHierarchyChange()
{
    Debug.Log("当Hierarchy视图中的任何对象发生改变时调用一次");
}

void OnProjectChange()
{
    Debug.Log("当Project视图中的资源发生改变时调用一次");
}

void OnInspectorUpdate()
{
    //Debug.Log("窗口面板的更新");
    //这里开启窗口的重绘，不然窗口信息不会刷新
    this.Repaint();
}

void OnSelectionChange()
{
    //当窗口出去开启状态，并且在Hierarchy视图中选择某游戏对象时调用
    foreach(Transform t in Selection.transforms)
    {
        //有可能是多选，这里开启一个循环打印选中游戏对象的名称
        Debug.Log("OnSelectionChange" + t.name);
    }
}

void OnDestroy()
{
    Debug.Log("当窗口关闭时调用");
}
```

- 打开一个通知栏
  - `this.ShowNotification(new GUIContent("This is a Notification"));`
- 关闭通知栏

    - `this.RemoveNotification();`
- 关闭面板
    - `this.Close();`
- 一些格式
    - `label`可以用字体:`EditorStyles.boldLabel`
    - 按钮什么的可以限制大小:
        - `GUILayout.MaxWidth(160)`
        - `GUILayout.MinHeight(60)`
        - `GUILayout.ExpandWidth(false)`

Inspector属性开发详细

- `Inspector`针对脚本进行编辑器绘制，所用的控件跟`Windows`窗口开发基本一致，只不过一些绘制方法、初始化等有所区别
- 

- 再创建一个`Editor`类

```
using UnityEditor;
[CustomEditor(typeof(MyButton))]
[CanEditMultipleObjects]
public class ButtonTest : Editor
{

}
```

  - `[CustomEditor(typeof(MyButton))]` 告诉编辑器，编辑哪个类
  - `[CanEditMultipleObjects]` 用于使自定义编辑器支持多对象编辑的属性
  - 同时才可以在 `MyButton` 类中（需要写编辑器的类）添加 `[ExecuteInEditMode]` 特性，这个特性的意思是在编辑器下，也会执行 `Awake()、Start()、Enable()、Update()`
- 可以根据选择不同的枚举信息进行绘制不同的窗口了

```
button.buttonType = (ButtonType)EditorGUILayout.EnumPopup("Button类型：", button.buttonType);

switch (button.buttonType)
{
    case ButtonType.Image:

        EditorGUI.BeginChangeCheck();

        EditorGUILayout.PropertyField(image, true, null);
        EditorGUILayout.PropertyField(normalSprite, true, null);
        EditorGUILayout.PropertyField(enterSprite, true, null);

        if (button.pressedSprite == null)...

        EditorGUILayout.PropertyField(pressedSprite, true, null);

        if (button.disableSprite == null)...

        EditorGUILayout.PropertyField(disableSprite, true, null);

        break;
    case ButtonType.Object:
        EditorGUI.BeginChangeCheck();

        EditorGUILayout.PropertyField(normalObject, true, null);
        EditorGUILayout.PropertyField(enterObject, true, null);

        if (button.pressedObject == null)
        {
            button.pressedObject = button.enterObject;
        }

        EditorGUILayout.PropertyField(pressedObject, true, null);

        if (button.disableObject == null)
        {
            button.disableObject = button.normalObject;
        }

        EditorGUILayout.PropertyField(disableObject, true, null);
```

https://blog.csdn.net/IOTHua

- 绘制继承 `UnityEngine.Object` 属性的两种方式
  - 在上述的图片中，看到很多 `EditorGUILayout.PropertyField()`，这个就是绘制出你编辑器的属性字段，它不用管你是什么类型，只要你对某个属性进行序列化查找赋值，那么它都可以在 `Inspector` 窗口中绘制出来 当然一般像 `int、bool、枚举、vector` 等都不用这个，因为 `Unity` 提供了这些基本的控件
  - 同时只要是你集成 `UnityEngine.Object` 的属性，比如 `GameObject、Component、Transform` 等那么你可以不用这么编写，用我们在【Windows窗口开发】下的 `EditorGUILayout.ObjectField()` 方法也是可以的，这样的话，你就不需要定义 `SerializedProperty` 字段以及在 `OnEnable` 初始化了，不过在编写 `Inspector` 窗口时，还是推荐用 `EditorGUILayout.PropertyField()`
- 绘制非 `UnityEngine.Object` 属性的方式

    - 当然如果是非`UnityEngine.Object`属性，比如`Unity`的事件`(UnityEvent)`事件，那么你就不能`EditorGUILayout.ObjectField()`，因为这是绘制不出来的，你只能采用`EditorGUILayout.PropertyField()`。
- 以下是详细
- 创建目标类，挂载在场景对象中

```
using UnityEngine;

public enum Course
{
    Chinese,
    Mathematics,
    English
}

public class InspectorExample : MonoBehaviour
{
    public int intValue;
    public float floatValue;
    public string stringValue;
    public bool boolValue;
    public Vector3  vector3Value;
    public Course enumValue = Course.Chinese;
    public Color colorValue = Color.white;
    public Texture textureValue;
}
```

- 绘制方式一

```
using UnityEngine;
using UnityEditor;

[CustomEditor(typeof(InspectorExample))]
public class InspectorExampleEditor : Editor
{
    //target指该编辑器类绘制的目标类，需要将它强转为目标类
    private InspectorExample _target { get { return target as
InspectorExample; } }

    //GUI重新绘制
    public override void OnInspectorGUI()
    {

 //EditorGUILayout.LabelField("IntValue",_target.intValue.ToString(),EditorS
tyles.boldLabel);
        //_target.intValue = EditorGUILayout.IntSlider(new
GUIContent("Slider"),_target.intValue, 0, 10);
        //_target.floatValue = EditorGUILayout.Slider(new
GUIContent("FloatValue"), _target.floatValue, 0, 10);
        _target.intValue = EditorGUILayout.IntField("IntValue",
_target.intValue);
        _target.floatValue = EditorGUILayout.FloatField("FloatValue",
_target.floatValue);
        _target.stringValue = EditorGUILayout.TextField("StringValue",
_target.stringValue);
```

```
        _target.boolValue = EditorGUILayout.Toggle("BoolValue",
_target.boolValue);
        _target.vector3Value = EditorGUILayout.Vector3Field("Vector3Value",
_target.vector3Value);
        _target.enumValue = (Course)EditorGUILayout.EnumPopup("EnumValue",
(Course)_target.enumValue);
        _target.colorValue = EditorGUILayout.ColorField(new
GUIContent("ColorValue"), _target.colorValue);
        _target.textureValue =
(Texture)EditorGUILayout.ObjectField("TextureValue", _target.textureValue,
typeof(Texture), true);
    }
}
```

- 绘制方式二

```
using UnityEditor;

[CustomEditor(typeof(InspectorExample))]
public class InspectorExampleEditor : Editor
{
    //定义序列化属性
    private SerializedProperty intValue;
    private SerializedProperty floatValue;
    private SerializedProperty stringValue;
    private SerializedProperty boolValue;
    private SerializedProperty vector3Value;
    private SerializedProperty enumValue;
    private SerializedProperty colorValue;
    private SerializedProperty textureValue;

    private void OnEnable()
    {
        //通过名字查找被序列化属性。
        intValue = serializedObject.FindProperty("intValue");
        floatValue = serializedObject.FindProperty("floatValue");
        stringValue = serializedObject.FindProperty("stringValue");
        boolValue = serializedObject.FindProperty("boolValue");
        vector3Value = serializedObject.FindProperty("vector3Value");
        enumValue = serializedObject.FindProperty("enumValue");
        colorValue = serializedObject.FindProperty("colorValue");
        textureValue = serializedObject.FindProperty("textureValue");
    }

    public override void OnInspectorGUI()
    {
        //表示更新序列化物体
        serializedObject.Update();
        EditorGUILayout.PropertyField(intValue);
        EditorGUILayout.PropertyField(floatValue);
        EditorGUILayout.PropertyField(stringValue);
        EditorGUILayout.PropertyField(boolValue);
        EditorGUILayout.PropertyField(vector3Value);
        EditorGUILayout.PropertyField(enumValue);
        EditorGUILayout.PropertyField(colorValue);
        EditorGUILayout.PropertyField(textureValue);
```

```
            //应用修改的属性值，不加的话，Inspector面板的值修改不了
            serializedObject.ApplyModifiedProperties();
        }
    }
```

> P.S.：第二种绘制方式相较于第一种，显示的效果是差不多的。虽然脚本内容多了一点，但是方式比较简单。不用根据每个变量的数据类型选择相对应的属性API绘制。

Hierarchy右键菜单开发详细

- 使用`[MenuItem()]`
- 使用`[CreateAssetMenu()]`

Gizmos辅助调试工具

- `Gizmos`是`Scene`场景的可视化调试或辅助工具，可以通过两种方式实现

通过OnDrawGizmo或者OnDrawGizmosSelected方法

- `OnDrawGizmos`在每一帧都会被调用，其渲染的`Gizmos`是一直可见的
- 而`OnDrawGizmosSelected`是当物体被选中的时候才会显示

```
public class GizmosExample : MonoBehaviour
{
    //绘制效果一直显示
    private void OnDrawGizmos()
    {
        var color = Gizmos.color;
        Gizmos.color = Color.white;
        Gizmos.DrawCube(transform.position, Vector3.one);
        Gizmos.color = color;
    }
    //绘制效果在选中对象时显示
    private void OnDrawGizmosSelected()
    {
        var color = Gizmos.color;
        Gizmos.color = Color.white;
        Gizmos.DrawWireCube(transform.position, Vector3.one);
        Gizmos.color = color;
    }
}
```

> P.S：`Gizmos.color`作为全局的静态变量，为了防止这里的`color`修改会对其他地方的绘制造成影响，所以在绘制完`Gizmos`的时候，将`Gizmos.color`修改为原先的值。
> `OnDrawGizmo`或者`OnDrawGizmosSelected`作为`MonoBehaviour`的方法，可以直接放在业务逻辑脚本中

通过DrawGIzmos特性

- 新建一个表示业务逻辑的类`TargetExample`，将其挂载在场景中的对象身上

```
using UnityEngine;
```

```
using UnityEditor;

public class GizmosTest
{
    //表示物体显示并且被选择的时候，绘制Gizmos
    [DrawGizmo(GizmoType.Active | GizmoType.Selected)]
    //第一个参数需要指定目标类，目标类需要挂载在场景对象中
    private static void MyCustomOnDrawGizmos(TargetExample target, GizmoType
gizmoType)
    {
        var color = Gizmos.color;
        Gizmos.color = Color.white;
        //target为挂载该组件的对象
        Gizmos.DrawCube(target.transform.position, Vector3.one);
        Gizmos.color = color;
    }
}
```

> P.S： 该方法需要将该类放在`Editor`文件夹内，使用特性的方法可以将业务逻辑和调试脚本分开。由于其针对的是编辑器组件的方法，需要设置为`Static`方法。

GizmosType	描述
Active	如果激活，则绘制
SelectedOrChild	如果被选择或者选择子物体时，则绘制
NotSelected	如果全没选择，则绘制
Selected	如果选择，则绘制
Pickable	在编辑器中gizmo可以被点选

常用Gizmos的方法

- `Gizmos.DrawCube()`绘制实体立方体
- `Gizmos.DrawWireCube()`绘制立方体边框
- `Gizmos.DrawRay()`绘制射线
- `Gizmos.DrawLine()`绘制直线
- `Gizmos.DrawIcon()`绘制`Icon`,`Icon`素材需要放在`Gizmos`文件夹中
- `Gizmos.DrawFrustum()`绘制摄像机视椎体的视野范围

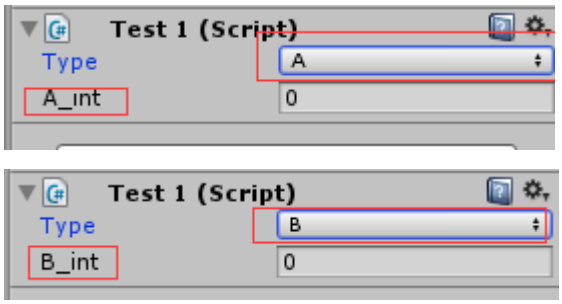
拓展：实现场景中一直显示主摄像机的视野范围

```
private Camera mainCamera;

private void OnDrawGizmos()
{
    if(mainCamera == null)
        mainCamera = Camera.main;
    Gizmos.color = Color.green;
    //设置gizmos的矩阵
    Gizmos.matrix = Matrix4x4.TRS(mainCamera.transform.position,
mainCamera.transform.rotation, Vector3.one);
    Gizmos.DrawFrustum(Vector3.zero, mainCamera.fieldOfView,
mainCamera.farClipPlane, mainCamera.nearClipPlane, mainCamera.aspect);
}
```

实际应用

选择不同枚举类型，inspector面板出现不同信息

- 效果：



- 带有枚举的脚本

```
using UnityEngine;
using System.Collections;

public class test1 : MonoBehaviour {
    public enum type1 {
        a,
        b
    }
    public type1 m_type;
    public int a_int;
    public int b_int;
}
```

- 创建一个`Editor`文件夹下的脚本

```
using UnityEngine;
using UnityEditor;
[CustomEditor(typeof(test1))]//关联之前的脚本
public class NewBehaviourScript1 : Editor {

    private SerializedObject test;//序列化
    private SerializedProperty m_type,a_int, b_int;//定义类型，变量a，变量b
    void OnEnable()
```

```
    {
        test = new SerializedObject(target);
        m_type = test.FindProperty("m_type");//获取m_type
        a_int = test.FindProperty("a_int");//获取a_int
        b_int = test.FindProperty("b_int");//获取b_int
    }
    public override void OnInspectorGUI()
    {
        test.Update();//更新test
        EditorGUILayout.PropertyField(m_type);
        if (m_type.enumValueIndex == 0) {//当选择第一个枚举类型
            EditorGUILayout.PropertyField(a_int);
        }
        else if (m_type.enumValueIndex == 1) {
            EditorGUILayout.PropertyField(b_int);
        }
        //serializedObject.ApplyModifiedProperties();
        test.ApplyModifiedProperties();//应用
    }
}
```

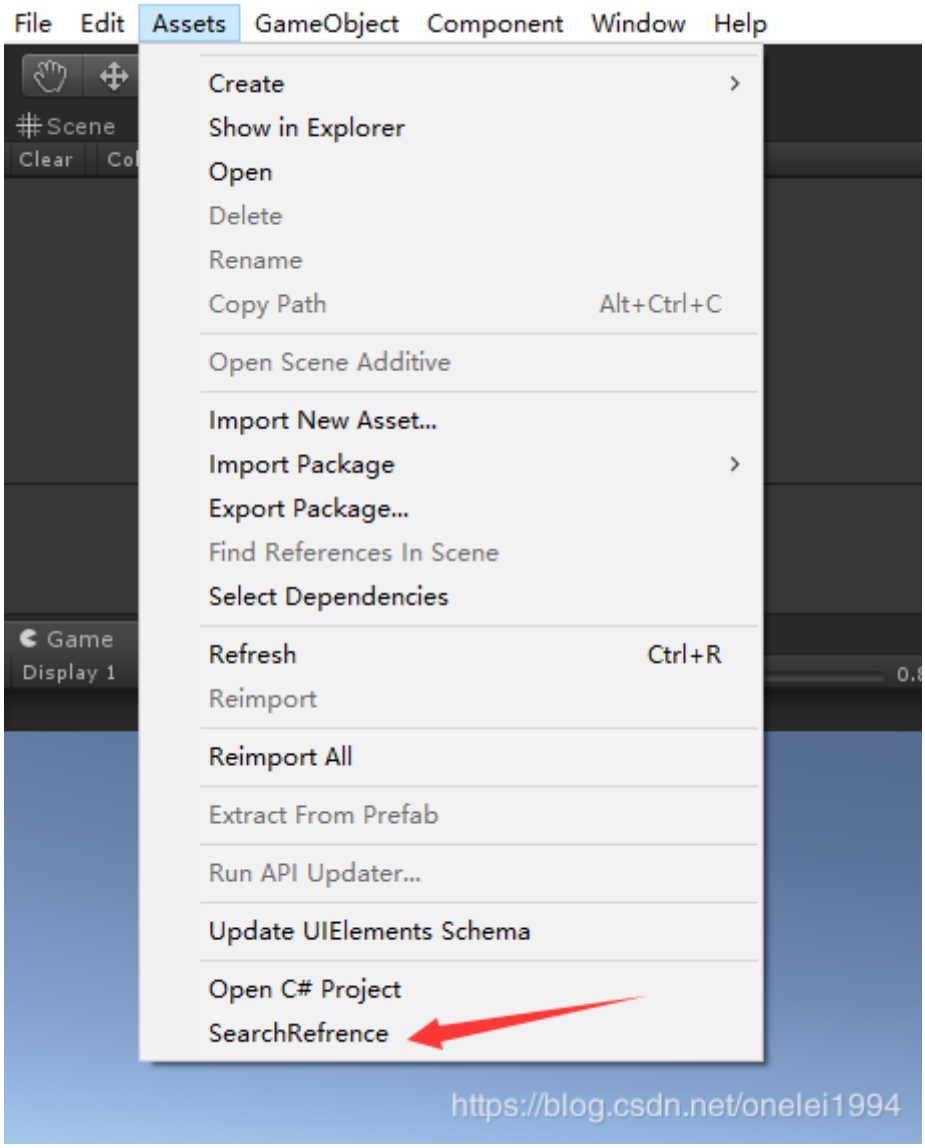
Unity查找图片被哪个Prefab引用

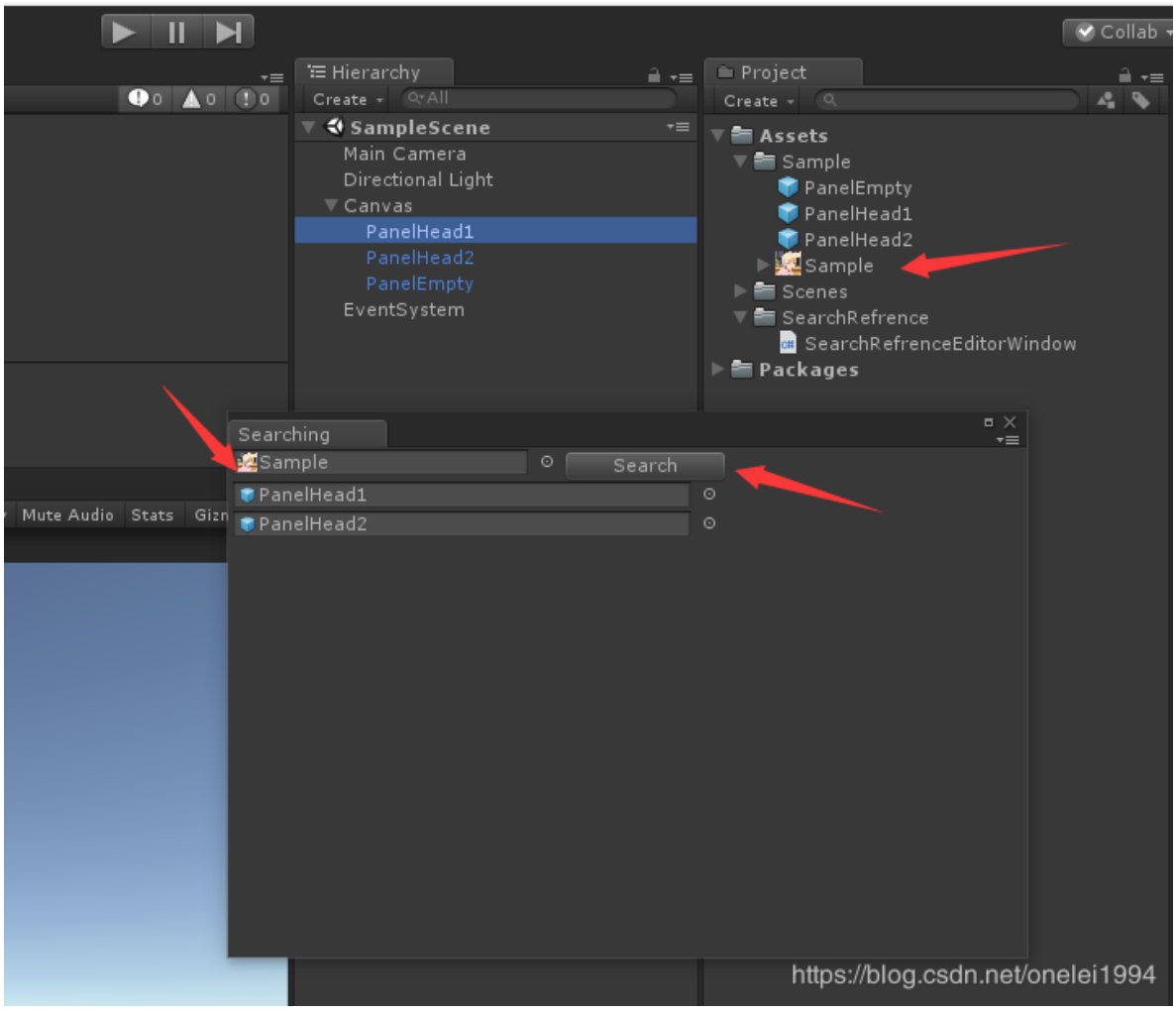
- Unity中的每个物体都有一个唯一Guid ID , Prefab里面都存有引用到的Guild ID 。 所以我们只要在Prefab信息中找到该图片的Guid ID就表明该Prefab引用到了该图片

```
using UnityEditor;
using UnityEngine;
using System.Collections.Generic;
using System.IO;

public class SearchRefrenceEditorWindow : EditorWindow
{
    /// <summary>
    /// 查找引用
    /// </summary>
    [MenuItem("Assets/SearchRefrence")]
    static void SearchRefrence()
    {
        SearchRefrenceEditorWindow window =
(SearchRefrenceEditorWindow)EditorWindow.GetWindow(typeof(SearchRefrenceEditorWi
ndow), false, "Searching", true);
        window.Show();
    }

private static Object searchObject;
private List<Object> result = new List<Object>();
private void OnGUI()
{
    EditorGUILayout.BeginHorizontal();
    searchObject = EditorGUILayout.ObjectField(searchObject, typeof(Object),
true, GUILayout.Width(200));
    if (GUILayout.Button("Search", GUILayout.Width(100)))
    {
        result.Clear();
```

```
        if (searchObject == null)
            return;

        string assetPath = AssetDatabase.GetAssetPath(searchObject);
        string assetGuid = AssetDatabase.AssetPathToGUID(assetPath);
        //只检查prefab
        string[] guids = AssetDatabase.FindAssets("t:Prefab", new[] { "Assets"
});

        int length = guids.Length;
        for (int i = 0; i < length; i++)
        {
            string filePath = AssetDatabase.GUIDToAssetPath(guids[i]);
            EditorUtility.DisplayCancelableProgressBar("Checking", filePath, i /
length * 1.0f);

            //检查是否包含guid
            string content = File.ReadAllText(filePath);
            if(content.Contains(assetGuid))
            {
                Object fileObject = AssetDatabase.LoadAssetAtPath(filePath,
typeof(Object));
                result.Add(fileObject);
            }
        }
        EditorUtility.ClearProgressBar();
    }
    EditorGUILayout.EndHorizontal();

    //显示结果
    EditorGUILayout.BeginVertical();
    for (int i = 0; i < result.Count; i++)
    {
        EditorGUILayout.ObjectField(result[i], typeof(Object), true,
GUILayout.Width(300));
    }
    EditorGUILayout.EndHorizontal();
}

}
```

- 打开菜单栏“Assets/SearchRefrence”，将要查找的图片拖入图中的位置，点击查找按钮即可


File
Edit
Assets
GameObject
Component
Window
Help
Create
Show in Explorer
Open
Delete
Rename
Copy Path
Alt+Ctrl+C
Open Scene Additive
Import New Asset...
Import Package
Export Package...
Find References In Scene
Select Dependencies
Refresh
Ctrl+R
Reimport
Reimport All
Extract From Prefab
Run API Updater...
Update UIElements Schema
Open C# Project
SearchRefrence
Scene
Clear
Game
Display 1
https://blog.csdn.net/onelei1994



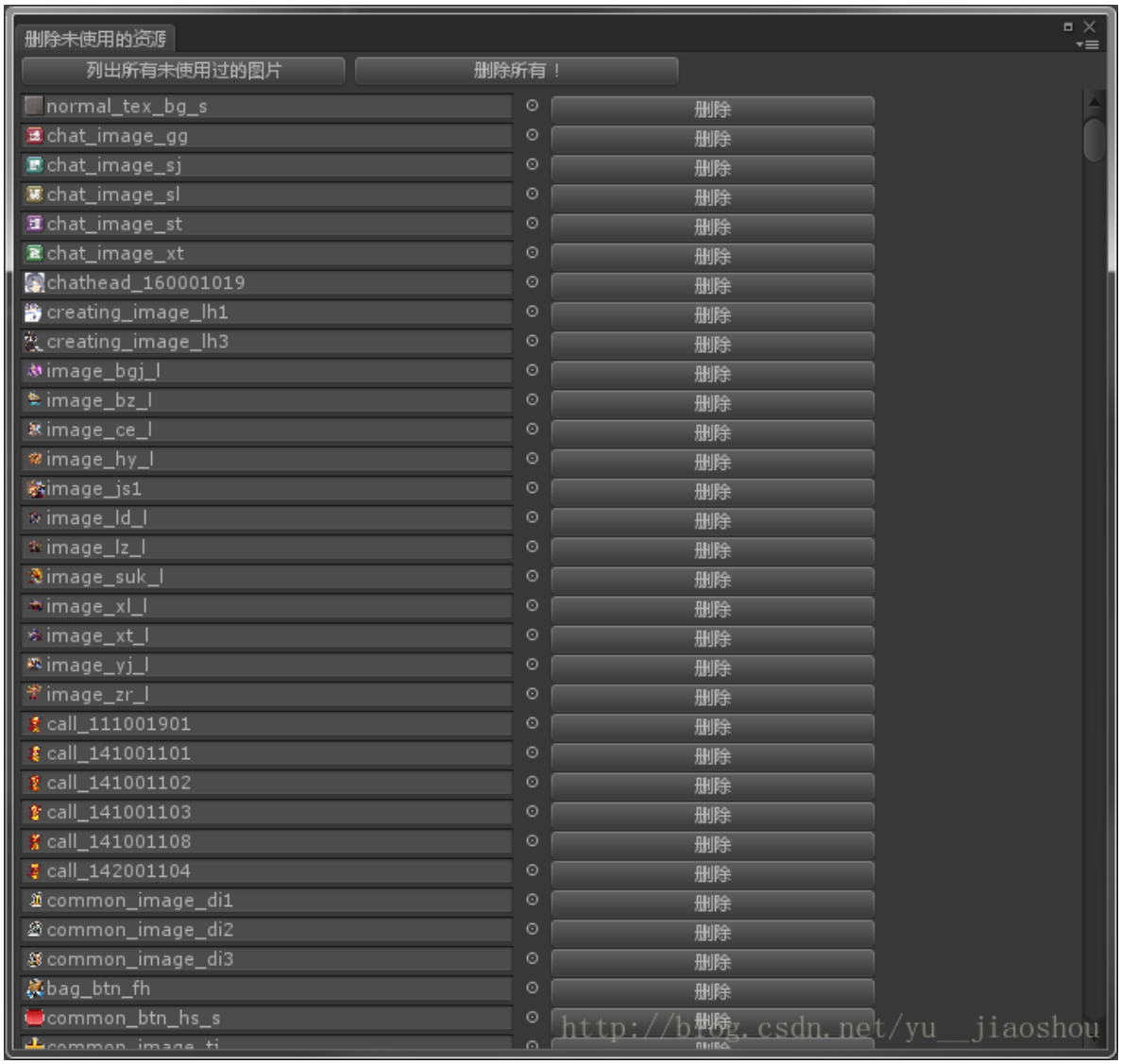
编辑器扩展之删除未使用资源

```
using System.Collections.Generic;
using UnityEngine;
using UnityEditor;
using System.IO;
using System.Linq;

namespace CZGame
{
    public class ShowAllNotUseResource : EditorWindow
    {
        private List<string> checkType = new List<string>();
        private List<string> allResource = new List<string>();  //所有使用资源
        private List<string> notUseReource = new List<string>();  //没有使用过的资源

        Dictionary<string, List<string>> mapReferenceOtherModulePref = new Dictionary<string, List<string>>();
        Vector2 scrollPos = Vector2.zero;
        private void StartCheck()
        {
            List<string> withoutExtensions = new List<string>() { ".prefab" };
            string[] files = Directory.GetFiles(Application.dataPath + "/GameRes/UIPrefab", "*.*", SearchOption.AllDirectories)
                .Where(s => withoutExtensions.Contains(Path.GetExtension(s).ToLower())).ToArray();
```

```
            List<string> withoutExtensions2 = new List<string>() { ".png",".jpg"
};
            //所有project 里面的资源
            string[] UIReource = Directory.GetFiles(Application.dataPath +
"/GameRes/UI", "*.*", SearchOption.AllDirectories)
                 .Where(s =>
withoutExtensions2.Contains(Path.GetExtension(s).ToLower())).ToArray();

            for (int i = 0; i < UIReource.Length; i++)
            {
                UIReource[i] = UIReource[i].Replace("\\", "/");
                int index = UIReource[i].IndexOf("Assets");
                UIReource[i] = UIReource[i].Substring(index);
            }

            mapReferenceOtherModulePref.Clear();
            float count = 0;
            foreach (string file in files)
            {
                count++;
                EditorUtility.DisplayProgressBar("Processing...", "检索中....",
count / files.Length);

                string strFile =
file.Substring(file.IndexOf("Asset")).Replace('\\', '/');
                string[] dependenciesFiles =
AssetDatabase.GetDependencies(strFile, false);

                foreach (string depFile in dependenciesFiles)
                {
                    bool isNeedShow = false;
                    foreach (string type in checkType)
                    {
                        //存在设置类型  需要显示
                        if (depFile.IndexOf(type) > -1)
                        {
                            isNeedShow = true;
                            break;
                        }
                    }
                    if (isNeedShow == false)
                    {
                        continue;
                    }
                    allResource.Add(depFile);
                }
            }

            EditorUtility.ClearProgressBar();


            for (int i = 0; i < UIReource.Length; i++)
            {
                bool isUse = false;
                foreach (string usePath in allResource)
                {
                    if (UIReource[i] == usePath)
```

```
                    {
                        isUse = true;
                        break;
                    }
                }

                if (isUse == false)
                {
                    notUseReource.Add(UIReource[i]);
                }

            }
        }

        void OnGUI()
        {

            EditorGUILayout.BeginHorizontal();
            if (GUILayout.Button("列出所有未使用过的图片", GUILayout.Width(200)))
            {
                allResource = new List<string>();
                notUseReource = new List<string>();
                checkType.Clear();
                checkType.Add(".jpg");
                checkType.Add(".png");
                StartCheck();
            }

            if (GUILayout.Button("删除所有 !", GUILayout.Width(200)))
            {

                float count = 0;
                foreach (var path in notUseReource)
                {
                    count++;
                    EditorUtility.DisplayProgressBar("Processing...", "删除中...
("+ count + "/ " + notUseReource.Count + ")", count / notUseReource.Count);
                    File.Delete(path);
                    AssetDatabase.Refresh();
                }

                EditorUtility.ClearProgressBar();
            }
            EditorGUILayout.EndHorizontal();

            scrollPos = EditorGUILayout.BeginScrollView(scrollPos);
            foreach (var path in notUseReource)
            {
                EditorGUILayout.BeginHorizontal();
                Texture2D t = (Texture2D)AssetDatabase.LoadAssetAtPath(path,
typeof(Texture2D));
                EditorGUILayout.ObjectField(t, typeof(Texture2D), false);

                if (GUILayout.Button("删除", GUILayout.Width(200)))
                {
                    File.Delete(path);
                    AssetDatabase.Refresh();
                }
```

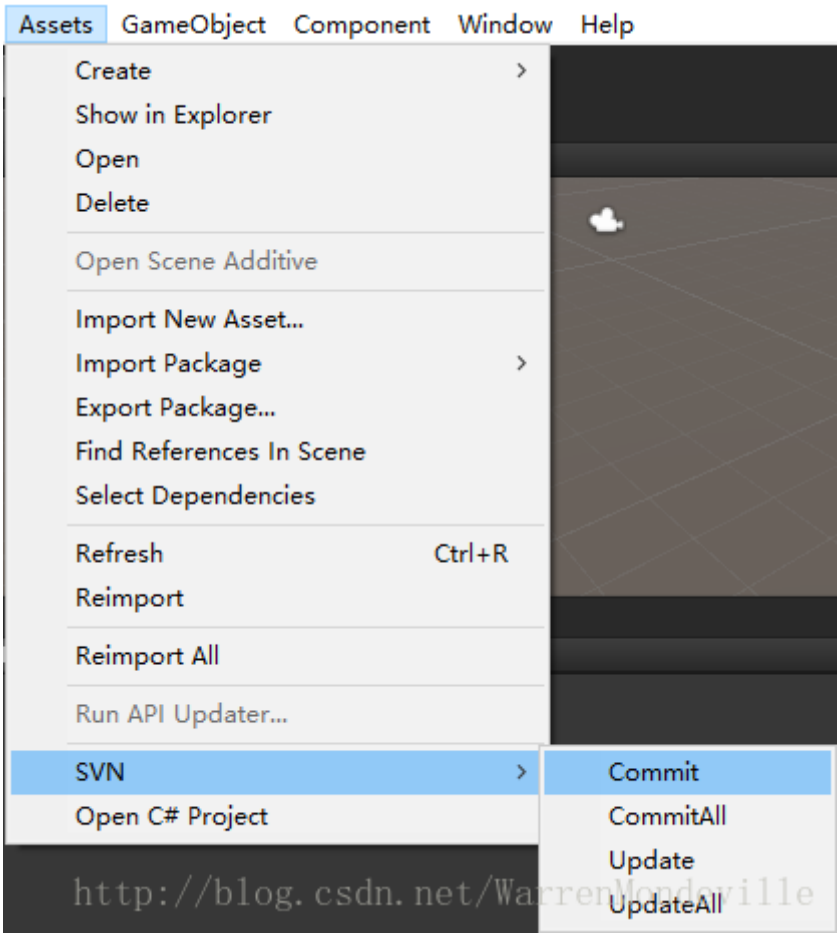
```
                EditorGUILayout.Space();

                EditorGUILayout.EndHorizontal();


            }
            EditorGUILayout.EndScrollView();



        }
    }
}
```



数组或list集合的显示方式

- 数组或list集合的显示方式

将TortoiseSVN的基础操作内嵌Unity




```
/*
 * 将TortoiseSVN的基础操作内嵌Unity
 * 这里只是举了几个简单的例子
 * 具体命令可参见TortoiseSVN的help功能
 *
 */

using UnityEngine;
using UnityEditor;
using System.Diagnostics;

public class UnitySVN
{
    private const string COMMIT = "commit";
    private const string UPDATE = "update";

    private const string SVN_COMMIT = "Assets/SVN/Commit";
    private const string SVN_COMMIT_ALL = "Assets/SVN/CommitAll";
    private const string SVN_UPDATE = "Assets/SVN/Update";
    private const string SVN_UPDATE_ALL = "Assets/SVN/UpdateAll";

    /// <summary>
    /// 创建一个SVN的cmd命令
    /// </summary>
    /// <param name="command">命令(可在help里边查看)</param>
    /// <param name="path">命令激活路径</param>
    public static void SVNCommand(string command, string path)
    {
        //closeonend 2 表示假设提交没错，会自动关闭提交界面返回原工程，详细描述可在
        //TortoiseSVN/help/TortoiseSVN/Automating TortoiseSVN里查看
```

```
        string c = "/c tortoiseproc.exe /command:{0} /path:\"{1}\" /closeonend
2";
        c = string.Format(c, command, path);
        ProcessStartInfo info = new ProcessStartInfo("cmd.exe", c);
        info.WindowStyle = ProcessWindowStyle.Hidden;
        Process.Start(info);
    }
    /// <summary>
    /// 提交选中内容
    /// </summary>
    [MenuItem(SVN_COMMIT)]
    public static void SVNCommit()
    {
        SVNCommand(COMMIT, GetSelectedObjectPath());
    }
    /// <summary>
    /// 提交全部Assets文件夹内容
    /// </summary>
    [MenuItem(SVN_COMMIT_ALL)]
    public static void SVNCommitAll()
    {
        SVNCommand(COMMIT, Application.dataPath);
    }
    /// <summary>
    /// 更新选中内容
    /// </summary>
    [MenuItem(SVN_UPDATE)]
    public static void SVNUpdate()
    {
        SVNCommand(UPDATE, GetSelectedObjectPath());
    }
    /// <summary>
    /// 更新全部内容
    /// </summary>
    [MenuItem(SVN_UPDATE_ALL)]
    public static void SVNUpdateAll()
    {
        SVNCommand(UPDATE, Application.dataPath);
    }

    /// <summary>
    /// 获取全部选中物体的路径
    /// 包括meta文件
    /// </summary>
    /// <returns></returns>
    private static string GetSelectedObjectPath()
    {
        string path = string.Empty;

        for (int i = 0; i < Selection.objects.Length; i++)
        {
            path +=
AssetsPathToFilePath(AssetDatabase.GetAssetPath(Selection.objects[i]));
            //路径分隔符
            path += "*";
            //meta文件
```

```
            path +=
AssetsPathToFilePath(AssetDatabase.GetAssetPath(Selection.objects[i])) +
".meta";
            //路径分隔符
            path += "*";
        }

        return path;
    }
    /// <summary>
    /// 将Assets路径转换为File路径
    /// </summary>
    /// <param name="path">Assets/Editor/...</param>
    /// <returns></returns>
    public static string AssetsPathToFilePath(string path)
    {
        string m_path = Application.dataPath;
        m_path = m_path.Substring(0, m_path.Length - 6);
        m_path += path;

        return m_path;
    }

}
```

更多场景应用

- 编辑器开发的更多应用