Unity Editor 实践

一. 简单的可重新排序的列表

我们知道，在代码中直接使用 List，那么 List 里面元素的位置是无法在 Inspector 窗口中进行修改的。为了使 List 里的元素位置能在 Inspector 面板上修改，我们就需要创建一个自己的 Editor 脚本，并在脚本中使用 ReorderableList。

首先我们先创建一个 MonoBehaviour 脚本，在这个脚本中创建一个存放颜色的 List。

PS：

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

public class TestList : MonoBehaviour
{
    public List<Color> colorList = new List<Color>();
}
```

1. 创建 Editor 脚本

有了 MonoBehaviour 脚本后，我们就可以在 Editor 文件夹中创建 Editor 脚本了。如果没有 Editor 文件夹，我们就需要自己创建一个名为 Editor 的文件夹。

我们首先需要将新建脚本中的 MonoBehaviour 改为 Editor，并引入 UnityEditor 和 UnityEditorInternal 命名空间，以确保自定义编辑器的正确运行。同时，我们需要在类前面使用 CustomEditor 特性，来标记这个类是我们的自定义编辑类。

PS：

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

[CustomEditor(typeof(TestList))]
public class TestListEditor : Editor
{
}
```

虽然我们现在已经有了一个指向 TestList 类型的自定义编辑器，但是在 Inspector 面板上依旧是原始的模样。

图 1. Inspector 面板没有任何改变

Inspector 面板没有发生变化是因为我们还没有重写 OnInspectorGUI 函数，只要重写了 OnInspectorGUI 函数，我们就可以按照自己的习惯去显示内容了。

PS：

```
public override void OnInspectorGUI()
    {
        EditorGUILayout.Space();
        serializedObject.Update();
        //应用属性修改
        serializedObject.ApplyModifiedProperties();
    }
```

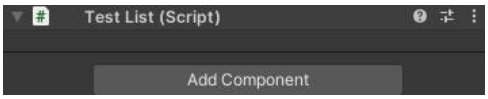

图 2.　　重写后 Inspector 面板发生了改变

因为我们只是单纯的重写了 OnInspectorGUI 函数，还没有向里面添加自定义的显示内容，所以现在的 Inspector 面板就是什么都没有。

2. 定义 ReorderableList

接下来我们就可以定义一个 ReorderableList，并在 OnEnable 函数中设置 ReorderableList。

PS：

```
//定义 ReorderableList
private ReorderableList colorList;
private void OnEnable()
{
}
```

定义了 ReorderableList 后，我们可以在 OnEnable 函数中将其实例化了。ReorderableList 具有多个构造函数，我们这里使用的是参数最多的一个。该构造函数的第三个参数控制是否可拖拽排序，第四个参数控制是否显示 Header，第五个参数控制是否显示添加元素按钮，第六个参数控制是否显示移除元素按钮。

PS：

```
colorList = new ReorderableList(serializedObject, serializedObject.FindProperty("colorList"), true, true, true, true);
```

当我们完成了实例化以后，我们就可以继续设置它的回调函数了。利用回调函数，我们可以设置事情发生后要采取的一系列行为。

我们先设置表头绘制完成后的回调，即 drawHeaderCallback。在这个回调函数里，我们将自己定义显示在 Inspector 面板上的列表名称。

PS：

```
//绘制表头
colorList.drawHeaderCallback = (Rect rect) =>
{
    //自定义列表表头的字符串
    GUI.Label(rect, "颜色");
};
```

完成这个回调函数的内容后，我们回到 Unity 中并没有看到有任何的变化。这是因为，我们还没有在 OnInspectorGUI 函数中，去执行 ReorderableList 的绘制。我们只需要在更新序列化对象之后，应用属性修改之前，去调用 ReorderableList 的绘制函数就可以了。

PS：

```
public override void OnInspectorGUI()
```

```
        {
            EditorGUILayout.Space();
            serializedObject.Update();
            //执行列表的绘制
            colorList.DoLayoutList();
            //应用属性修改
            serializedObject.ApplyModifiedProperties();
        }
```

现在，我们就可以看到自定义的效果了。

图 3.　　列表名称变成了我们自定义的颜色

但是，我们列表里的元素却不是我们想要的样子，接下来我们就将自定义元素的绘制了。新增绘制元素的回调 drawElementCallback，在回调函数里首先需要从列表中获取到元素，然后再设置元素的高度和初始位置，最后再使用 PropertyField 重新设置字段的显示。

PS：

```
//绘制元素
            colorList.drawElementCallback = (Rect rect, int index, bool selected, bool focused)
=>
            {
                SerializedProperty itemData =
colorList.serializedProperty.GetArrayElementAtIndex(index);
                //设置元素绘制的原点 Y 坐标
                rect.y += 2;
                //设置绘制的高度
                rect.height = EditorGUIUtility.singleLineHeight;
                EditorGUI.PropertyField(rect, itemData, GUIContent.none);
            };
```

图 4.　　自定义的元素显示

因为我们在实例化时设置了可显示增加和移除按钮，在 Inspector 面板上我们就可以看到这两个按钮了。虽然我们现在就可以正常使用这两个功能，但是我们还是希望能够自定义增加和移除的行为。

我们先添加 onRemoveCallback 回调，它会在点击移除按钮时响应。我们在方法体内编写提示弹窗的逻辑，当我们点击提示弹窗中的确认按钮，才会去执行元素的移除操作。通过这样的设置，我们可以避免因误操作导致的元素移除。

PS：

```
//当移除元素时回调
        colorList.onRemoveCallback = (ReorderableList list) =>
        {
            //弹出一个对话框
            if (EditorUtility.DisplayDialog("警告", "是否确定删除该颜色", "是", "否"))
            {
                //当点击“是”，移除元素
                ReorderableList.defaultBehaviours.DoRemoveButton(list);
            }
        };
```

接着，我们再添加 onAddCallback 回调，它会在点击新增按钮时响应。在函数体里，我们想要自定义新增元素的颜色，首先就需使列表容量自增。然后通过索引去获取到新增的元素，因为我们的元素类型 color，我们就设置它的 colorValue 即可。

PS：

```
        //添加按钮回调
        colorList.onAddCallback = (ReorderableList list) =>
        {
            if (list.serializedProperty != null)
            {
                //列表大小增加一
                list.serializedProperty.arraySize++;
                //设置列表末尾的索引，索引从 0 开始所以新的索引是容量减一
                list.index = list.serializedProperty.arraySize - 1;
                //根据索引获取列表元素
                SerializedProperty itemData =
list.serializedProperty.GetArrayElementAtIndex(list.index);
                itemData.colorValue = Color.red;
            }
        };
```

现在回到 Unity 中，再次进行删除和新增操作时，就可以看见我们的 Editor 应用效果了。

3. 完整的 Editor 代码

PS：

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

[CustomEditor(typeof(TestList))]
public class TestListEditor : Editor
{
    //定义 ReorderableList
    private ReorderableList colorList;
    private void OnEnable()
    {
```

```
            /*
             * 第三个参数控制是否可拖曳排序
             * 第四个参数控制是否显示 Header
             * 第五个参数控制是否显示添加元素按钮
             * 第六个参数控制是否显示移除元素按钮
             */
            colorList = new ReorderableList(serializedObject,
serializedObject.FindProperty("colorList"), true, true, true, true);
            //绘制元素
            colorList.drawElementCallback = (Rect rect, int index, bool selected, bool focused)
=>
            {
                SerializedProperty itemData =
colorList.serializedProperty.GetArrayElementAtIndex(index);
                //设置元素绘制的原点 Y 坐标
                rect.y += 2;
                //设置绘制的高度
                rect.height = EditorGUIUtility.singleLineHeight;
                EditorGUI.PropertyField(rect, itemData, GUIContent.none);
            };
            //绘制表头
            colorList.drawHeaderCallback = (Rect rect) =>
            {
                //自定义列表表头的字符串
                GUI.Label(rect, "颜色");
            };
            //当移除元素时回调
            colorList.onRemoveCallback = (ReorderableList list) =>
            {
                //弹出一个对话框
                if (EditorUtility.DisplayDialog("警告", "是否确定删除该颜色", "是", "否"))
                {
                    //当点击“是”，移除元素
                    ReorderableList.defaultBehaviours.DoRemoveButton(list);
                }
            };
            //添加按钮回调
            colorList.onAddCallback = (ReorderableList list) =>
            {
                if (list.serializedProperty != null)
                {
                    //列表大小增加一
                    list.serializedProperty.arraySize++;
                    //设置列表末尾的索引，索引从 0 开始所以新的索引是容量减一
```

```
                    list.index = list.serializedProperty.arraySize - 1;
                    //根据索引获取列表元素
                    SerializedProperty itemData =
list.serializedProperty.GetArrayElementAtIndex(list.index);
                    itemData.colorValue = Color.red;
                }
                else
                {
                    //执行添加操作
                    ReorderableList.defaultBehaviours.DoAddButton(list);
                }
            };
        }

        public override void OnInspectorGUI()
        {
            EditorGUILayout.Space();
            serializedObject.Update();
            //执行列表的绘制
            colorList.DoLayoutList();
            //应用属性修改
            serializedObject.ApplyModifiedProperties();
        }
    }
```

二. 重命名工具

1. 创建工具窗口

我们首先需要创建一个派生自 EditorWindow 的脚本，并使用宏定义保证相关的代码只在 UNITY_EDITOR 下可运行。我们编写一个静态方法，配合 MenuItem 来打开我们的重命名窗口。

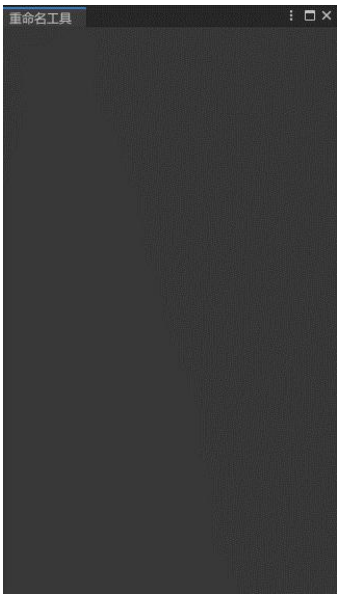

图 1.　　打开的空无一物的窗口

PS：

```
#if UNITY_EDITOR
using System.IO;
using UnityEditor;
using UnityEngine;


public class GameTool : EditorWindow
{
    #region  打开重命名窗口
    [MenuItem("游戏工具/重命名工具")]
    private static void OpenWindow()
    {
        GetWindow<GameTool>("重命名工具").Show();
    }
    #endregion
}
#endif
```

2. 编写界面显示

我们现在已经可以打开重命名窗口了，只不过目前这个窗口里面还没有任何的内容，我们后面所需要做的就是向里面填充内容。

首先在 OnGUI 函数里，设置标题的颜色、字体大小和对齐的方式。

PS：

```
GUI.color = Color.cyan;
GUILayout.Space(10);
GUI.skin.label.fontSize = 24;
//居中对齐
GUI.skin.label.alignment = TextAnchor.MiddleCenter;
GUILayout.Label("重命名工具");
```

设置完成后回到 Unity，再次打开重命名窗口，我们就可以看见已经出现了设置好的标题。

图 2.　　窗口中的 Title

接下来，我们将继续新增用于输入的区域，这时我们就将使用到水平布局和垂直布局。因为我们在前面构建标题时设置了 GUI 的样式，如果我们不在后续内容中修改这些样式，那么后续的内容就都是标题的样式。

使用 TextField，我们就创建了一个可供编辑的单行文本区域，在这个区域内我们就可以输自己想要的文本内容。对于命名的后缀，我们希望他只能是数值。我们就需要在输入后缀不正确时，即异常状态时打印出警告，并使输入无效。

PS：

```
GUI.skin.label.fontSize = 16;
//左对齐
GUI.skin.label.alignment = TextAnchor.MiddleLeft;
GUILayout.Space(20);
#endregion

//开启垂直布局
GUILayout.BeginVertical();
{
    #region  文本输入
    //开启水平布局
    GUILayout.BeginHorizontal();
    {
        GUI.color = Color.yellow;
        GUILayout.Label("名称前缀：");
        //在当前布局组中插入空白元素。
        GUILayout.FlexibleSpace();
        GUI.color = Color.white;
        targetName = GUILayout.TextField(targetName,
GUILayout.Width(140));
    }
    GUILayout.EndHorizontal();

    GUILayout.Space(10);

    GUILayout.BeginHorizontal();
    {
        GUI.color = Color.yellow;
        GUILayout.Label("后缀起始数值：  ");
        GUILayout.FlexibleSpace();
        GUI.color = Color.white;

        startNum_str = GUILayout.TextField(startNum_str,
GUILayout.Width(140));
        if (startNum_str != string.Empty)
        {
            try
            {
                startNum_int = int.Parse(startNum_str);
            }
            catch
            {
                Debug.LogError("解析错误,请使用数值");
```

```
                    startNum_str = string.Empty;
                }
            }
        }
        GUILayout.EndHorizontal();
        #endregion
    }
    GUILayout.EndVertical();
```

完成后回到 Unity 中，我们再次打开窗口时就有了两个可供编辑的文本域了。当我们的后缀输入不是数值时，我们就会在 Unity 中看到警告，并置空文本域。

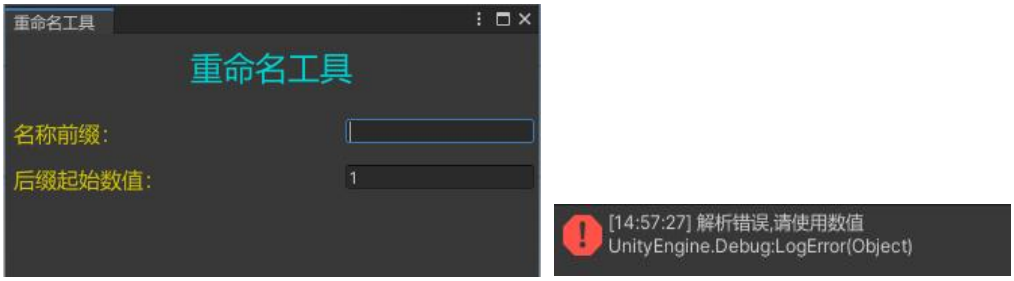

图 3.　可供编辑的文本域，右图是错误输入警告

3. 新增重命名按钮

有了前缀和后缀的文本域后，我们最后就还只差一个确认按钮了。我们希望，只有选择了至少一个对象才能进行重命名，未选择对象时不能进行重命名操作。从视觉层面来说，在可以点击按钮时，我们将按钮的 GUI 颜色设置为绿色，不可用时就设置为红色。

PS：

```
GUILayout.Space(30);
//判断是否有选择的对象
bool hasObject = (Selection.objects.Length > 0);
GUI.enabled = hasObject;

GUILayout.FlexibleSpace();
if (!hasObject)
{
    GUI.color = Color.red;
    GUILayout.Button("没有选择任何对象!");
    GUI.color = Color.white;

}
else
{
    GUI.color = Color.green;
    if (GUILayout.Button("重命名"))
    {
        Rename(targetName, startNum_int);
    }
}
```

```
            GUILayout.Space(20);
```

当按钮处于可用状态时，点击按钮就会调用重命名方法。在重命名方法中去遍历选中的对象，依次判断选中的对象是否是拥有实际存储路径的资源。如果是 Asset 资源，我们就需要在遍历完成后重新资源保存一次；不是 Asset 资源，我们就直接命名即可。

PS：

```
/// <summary>
    ///  重命名
    /// </summary>
    /// <param name="prefixName">前缀名称</param>
    /// <param name="suffixIndex">后缀数值</param>
    private void Rename(string prefixName, int suffixIndex)
    {
        //去除头尾空白字符串
        string name = prefixName.Trim();
        int index = suffixIndex;
        //若名字不为空
        if ((name + index) != string.Empty)
        {
            bool isAssetsObject = false;//flag, 是否是 assets 文件夹的资源

            foreach (Object theObject in Selection.objects)
            {
                //获得选中对象的路径
                string assetPath = AssetDatabase.GetAssetPath(theObject);
                //查看路径后缀，若后缀不为空，则为 assets 文件夹物体
                if (Path.GetExtension(assetPath) != "")
                {
                    //修改 Asset 资源的名称
                    AssetDatabase.RenameAsset(assetPath, $"{name}_{index}");
                    if (!isAssetsObject)
                    {
                        isAssetsObject = true;//修改 flag
                    }
                }
                else//后缀为空，是场景中的物体
                {
                    theObject.name = $"{name}_{index}";
                }
                index++;
            }
            //若是 assets 文件夹资源，则保存后再刷新
            if (isAssetsObject)
            {
                //将所有未保存的资源更改写入磁盘
```

```
                AssetDatabase.SaveAssets();
                AssetDatabase.Refresh();
            }
        }
    }
```

4. 完整 Editor 代码

PS：

```
#if UNITY_EDITOR
using System.IO;
using UnityEditor;
using UnityEngine;


public class RenameTool : EditorWindow
{
    #region 成员变量
    private string targetName = "";
    private string startNum_str = "1";
    private int startNum_int = 0;
    #endregion

    #region 打开重命名窗口
    [MenuItem("游戏工具/重命名工具")]
    private static void OpenWindow()
    {
        GetWindow<RenameTool>("重命名工具").Show();
    }
    #endregion

    #region 构建窗口
    private void OnGUI()
    {
        #region 工具标题
        GUI.color = Color.cyan;
        GUILayout.Space(10);
        GUI.skin.label.fontSize = 24;
        //居中对齐
        GUI.skin.label.alignment = TextAnchor.MiddleCenter;
        GUILayout.Label("重命名工具");


        GUI.skin.label.fontSize = 16;
        //左对齐
        GUI.skin.label.alignment = TextAnchor.MiddleLeft;
```

```
            GUILayout.Space(20);
            #endregion

            //开启垂直布局
            GUILayout.BeginVertical();
            {
                #region  文本输入
                //开启水平布局
                GUILayout.BeginHorizontal();
                {
                    GUI.color = Color.yellow;
                    GUILayout.Label("名称前缀：");
                    //在当前布局组中插入空白元素。
                    GUILayout.FlexibleSpace();
                    GUI.color = Color.white;
                    targetName = GUILayout.TextField(targetName,
GUILayout.Width(140));
                }
                GUILayout.EndHorizontal();

                GUILayout.Space(10);

                GUILayout.BeginHorizontal();
                {
                    GUI.color = Color.yellow;
                    GUILayout.Label("后缀起始数值：  ");
                    GUILayout.FlexibleSpace();
                    GUI.color = Color.white;

                    startNum_str = GUILayout.TextField(startNum_str,
GUILayout.Width(140));
                    if (startNum_str != string.Empty)
                    {
                        try
                        {
                            startNum_int = int.Parse(startNum_str);
                        }
                        catch
                        {
                            Debug.LogError("解析错误,请使用数值");
                            startNum_str = string.Empty;
                        }
                    }
                }
```

```
            GUILayout.EndHorizontal();
            #endregion

            #region 构建按钮
            GUILayout.Space(30);
            //判断是否有选择的对象
            bool hasObject = (Selection.objects.Length > 0);
            GUI.enabled = hasObject;

            GUILayout.FlexibleSpace();
            if (!hasObject)
            {
                GUI.color = Color.red;
                GUILayout.Button("没有选择任何对象!");
                GUI.color = Color.white;
            }
            else
            {
                GUI.color = Color.green;
                if (GUILayout.Button("重命名"))
                {
                    Rename(targetName, startNum_int);
                }
            }
            GUILayout.Space(20);
            #endregion
        }
        GUILayout.EndVertical();
    }
    #endregion

    #region 重命名方法
    /// <summary>
    /// 重命名
    /// </summary>
    /// <param name="prefixName">前缀名称</param>
    /// <param name="suffixIndex">后缀数值</param>
    private void Rename(string prefixName, int suffixIndex)
    {
        //去除头尾空白字符串
        string name = prefixName.Trim();
        int index = suffixIndex;
        bool prefixIsEmpty = false;
```

```
            if (name == "")
            {
                prefixIsEmpty = true;
                Debug.LogWarning("您似乎没有输入前缀，如果需要前缀记得加上哦!!!
");
            }

            //若名字不为空
            if ((name + index) != string.Empty)
            {
                bool isAssetsObject = false;//flag, 是否是 assets 文件夹的资源

                foreach (Object theObject in Selection.objects)
                {
                    //获得选中对象的路径
                    string assetPath = AssetDatabase.GetAssetPath(theObject);
                    //查看路径后缀，若后缀不为空, 则为 assets 文件夹物体
                    if (Path.GetExtension(assetPath) != "")
                    {
                        //修改 Asset 资源的名称
                        if (prefixIsEmpty)
                            AssetDatabase.RenameAsset(assetPath, $"{index}");
                        else
                            AssetDatabase.RenameAsset(assetPath, $"{name}_{index}");
                        if (!isAssetsObject)
                        {
                            isAssetsObject = true;//修改 flag
                        }
                    }
                    else//后缀为空, 是场景中的物体
                    {
                        if (prefixIsEmpty)
                            theObject.name = $"{index}";
                        else
                            theObject.name = $"{name}_{index}";
                    }
                    index++;
                }
                //若是 assets 文件夹资源, 则保存后再刷新
                if (isAssetsObject)
                {
                    //将所有未保存的资源更改写入磁盘
                    AssetDatabase.SaveAssets();
                    AssetDatabase.Refresh();
```

```
                }
            }
        }
        #endregion
    }
    #endif
```

三. 场景选择工具

在实际的游戏开发中，我们可能会拥有非常多的场景。为了方便查找与切换场景，我们就可以编写一个场景选择工具来进行辅助。

图 4. 场景选择工具弹窗

1. SceneUtility

在开始编写场景选择工具窗口之前，我们还需要编写两个工具类来进行辅助，它们都将被存放在同一个命名空间下。我们首先要编写的工具类是场景工具，利用这个工具类就可以非常方便的执行打开场景的操作了。

首先，先编写检查指定场景是否打开的静态方法。我这里是通过路径获取的名称，各位也可以更改为直接比较名称。

PS：

```
/// <summary>
///  检查指定场景是否打开
/// </summary>
/// <param name="scenepath">场景路径</param>
/// <returns></returns>
public static bool SceneIsOpened(string scenepath)
{
    //获取当前激活的场景
    Scene activescene = EditorSceneManager.GetActiveScene();
    //返回不具有扩展名的指定路径字符串的文件名
    string scenename = Path.GetFileNameWithoutExtension(scenepath);
```

```
            return activescene.name.Equals(scenename);
        }
```

检测的方法编写完成后，我们就可以继续编写打开场景的方法了。在方法体内，我们需要判断当前是处于运行状态还是非运行状态，如果是非运行状态就直接调用 Editor 下的 OpenScene。如果是运行状态，就弹窗提示是否进行运行时场景的切换，确认需要运行时切换就调用运行时的场景切换代码。

需要注意的一点是，运行时切换场景，需要对应场景已经包含在 Build Settings 中，否则将无法切换场景。

PS：

```
/// <summary>
        ///  打开指定场景
        /// </summary>
        /// <param name="scenepath">场景路径</param>
        public static void OpenScene(string scenepath)
        {
            if (!SceneIsOpened(scenepath))
            {
                //因为是在 Editor 下打开场景的工具，如果在运行中使用就需要停止运行，或者改用运行时的 SceneManager
                if (EditorApplication.isPlaying)
                {
                    //弹出一个对话框
                    if (EditorUtility.DisplayDialog("警告", "是否需要在运行时切换场景，点击否会停止运行然后加载场景，点击是会立即切换场景", "是", "否"))
                    {
                        //当点击“是”，执行运行时场景加载
                        SceneManager.LoadScene(scenepath);
                    }
                    else
                    {
                        EditorApplication.isPlaying = false;
                        void Load(PlayModeStateChange state)
                        {
                            if (!EditorApplication.isPlaying)
                            {
                                EditorSceneManager.OpenScene(scenepath);
                                EditorApplication.playModeStateChanged -= Load;
                            }
                        }
                        EditorApplication.delayCall = () =>
                        {
                            if (!EditorApplication.isPlaying)
                                EditorSceneManager.OpenScene(scenepath);
                            else
```

```
                                        EditorApplication.playModeStateChanged += Load;
                                    };
                                }
                                return;
                            }
                            else
                                EditorSceneManager.OpenScene(scenepath);
                        }
                    }
```

2. FileUtility

我们现在就已经编写完成了一个工具类，接下来就可以继续编写用于文件操作的工具类了。在这个工具类里，我们首先需要编写的静态方法，就是将完整的 Windows 下的路径格式转换成 Unity 下的路径格式。

PS：

```
/// <summary>
        /// 路径格式转换成 Unity 下的路径格式
        /// </summary>
        /// <param name="path">原始路径</param>
        /// <returns></returns>
        public static string FormatToUnityPath(string path)
        {
            return path.Replace("\\", "/");
        }
```

接着，我们继续编写下一个静态方法。即，将完整路径转换为 Asset 内的相对路径。

PS：

```
/// <summary>
        /// 全路径转换为 Asset 内相对路径
        /// </summary>
        /// <param name="fullPath">全路径</param>
        /// <returns>Asset 内相对路径</returns>
        public static string FullPathToAssetPath(string fullPath)
        {
            return FormatToUnityPath(fullPath).Replace(Application.dataPath, "Assets");
        }
```

这个方法编写完成后，我们就可以编写获取文件及文件名称的方法了。

PS：

```
/// <summary>
        /// 获取文件路径
        /// </summary>
        /// <param name="directPath">直接路径</param>
        /// <param name="searchPattern"></param>
        /// <param name="searchOption">搜索选项，AllDirectories 为在搜索操作中包括当前目录和所有它的子目录。</param>
        /// <returns></returns>
```

```
        public static string[] GetFilesPaths(string directPath, string searchPattern = "*",
SearchOption searchOption = SearchOption.AllDirectories)
        {
            List<string> paths = new List<string>();
            FileInfo[] files = GetFiles(directPath, searchPattern, searchOption);
            string tempStr;
            if (files != null && files.Length > 0)
            {
                foreach (FileInfo file in files)
                {
                    //将完整路径转换为 Unity 里的相对路径
                    tempStr = FullPathToAssetPath(file.FullName);
                    paths.Add(tempStr);
                }
            }
            return paths.ToArray();
        }

        /// <summary>
        ///  获取文件
        /// </summary>
        /// <param name="directPath">绝对路径</param>
        /// <param name="searchPattern">要匹配的字符</param>
        /// <param name="searchOption">搜索选项，AllDirectories 为在搜索操作中包括
当前目录和所有它的子目录。</param>
        /// <returns></returns>
        public static FileInfo[] GetFiles(string directPath, string searchPattern = "*",
SearchOption searchOption = SearchOption.AllDirectories)
        {
            //测试目录是否存在或者在尝试确定指定目录是否存在时出错
            if (Directory.Exists(directPath))
            {
                DirectoryInfo directoryInfo = new DirectoryInfo(directPath);
                return directoryInfo.GetFiles(searchPattern, searchOption);
            }

            Debug.LogError($"文件夹不存在{directPath}");
            return null;
        }

        /// <summary>
        ///  获取文件名称
        /// </summary>
        /// <param name="filePath">文件路径</param>
```

```
        /// <param name="withoutExtension">是否不需要扩展名称</param>
        /// <returns></returns>
        public static string GetFileName(string filePath, bool withoutExtension = true)
        {
            if (withoutExtension)
            {
                return
FormatToUnityPath(Path.GetFileNameWithoutExtension(filePath));
            }

            return FormatToUnityPath(Path.GetFileName(filePath));
        }
```

3. 编写场景选择窗口

现在，我们所有的工具类就都编写完成了。下面，我们就将开始编写我们的 EditorWindow 了。在 EditorWindow 中，一定不要忘记引用我们的命名空间。我们之前所编写的工具类都在同一个命名空间里，如果不引用命名空间就无法使用这些工具类。

在实际编写窗口之前，我们需要先把所需要用到的成员变量添加完成。

PS：

```
        /// <summary>
        ///  滑动列表的起始位置
        /// </summary>
        private Vector2 startScrollPos = Vector2.zero;
        /// <summary>
        ///  场景名称列表
        /// </summary>
        private List<string> scenesNames;
        /// <summary>
        ///  场景路径列表
        /// </summary>
        private List<string> scenesPaths;
        private bool isInitOver = false;
```

添加完所需的成员变量后，我们就可以使用 MenuItem 和静态方法去完成窗口的构建了。

PS：

```
        [MenuItem("游戏工具/场景选择工具", false, 0)]
        public static void Open()
        {
            ScenesChooseTool Window = GetWindow<ScenesChooseTool>();
            Window.titleContent = new
GUIContent(EditorGUIUtility.IconContent("TerrainInspector.TerrainToolRaise"))
            {
                text = "场景选择工具",
            };
            Window.Init();
        }
```

当我们打开窗口时，我们就需要初始化所需的数据。我们使用之前构建的工具类方法，将场景的路径和名称添加进我们的列表中，我们就完成了数据的初始化。

PS：

```
/// <summary>
///  初始化窗口数据
/// </summary>
private void Init()
{
    scenesName = new List<string>();
    scenesPath = new List<string>();
    //实际遍历的路径
    List<string> paths = new List<string>()
    {
        Application.dataPath+"/Scenes",
    };

    foreach (string path in paths)
    {
        string[] files = FileUtility.GetFilesPaths(path, "*Unity");
        if (files != null && files.Length > 0)
        {
            foreach (string file in files)
            {
                scenesPath.Add(file);
                scenesName.Add(FileUtility.GetFileName(file));
            }
        }
    }
    isInitOver = true;
}
```

接下来，就是利用这些数据，在 OnGUI 函数中去进行显示了。在 OnGUI 函数里，没有太多的内容，就只是简单的遍历数据，然后根据数据去创建按钮。

PS：

```
public void OnGUI()
    {
        //使用 GroupBox 的 GUI 样式
        GUILayout.BeginVertical("GroupBox");
        #region  大标题
        GUI.color = Color.green;
        GUILayout.Space(10);
        GUI.skin.label.fontSize = 24;
        //居中对齐
        GUI.skin.label.alignment = TextAnchor.MiddleCenter;
        GUILayout.Label("请选择场景");
```

```
            GUI.color = Color.white;
            #endregion
            startScrollPos = GUILayout.BeginScrollView(startScrollPos);

            if (isInitOver && scenesPaths != null)
            {
                for (int i = 0; i < scenesPaths.Count; i++)
                {
                    if (GUILayout.Button(scenesNames[i]))
                    {
                        SceneUtility.OpenScene(scenesPaths[i]);
                    }
                }
            }

            GUI.skin.label.fontSize = 16;
            GUI.skin.label.alignment = TextAnchor.MiddleLeft;
            GUILayout.EndScrollView();
            GUILayout.EndVertical();
        }
```

4. 完整代码

1) SceneUtility

PS：

```
using System.IO;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEditor.SceneManagement;
using UnityEditor;
using System;
namespace GameUtil
{
    public class SceneUtility
    {
        /// <summary>
        /// 检查指定场景是否打开
        /// </summary>
        /// <param name="scenepath">场景路径</param>
        /// <returns></returns>
        public static bool SceneIsOpened(string scenepath)
        {
            //获取当前激活的场景
            Scene activescene = EditorSceneManager.GetActiveScene();
            //返回不具有扩展名的指定路径字符串的文件名
            string scenename = Path.GetFileNameWithoutExtension(scenepath);
```

```
        return activescene.name.Equals(scenename);
    }

    /// <summary>
    ///  打开指定场景
    /// </summary>
    /// <param name="scenepath">场景路径</param>
    public static void OpenScene(string scenepath)
    {
        if (!SceneIsOpened(scenepath))
        {
            //因为是在 Editor 下打开场景的工具，如果在运行中使用就需要停止运行，或者改用运行时的 SceneManager
            if (EditorApplication.isPlaying)
            {
                //弹出一个对话框
                if (EditorUtility.DisplayDialog("警告", "是否需要在运行时切换场景，点击否会停止运行然后加载场景，点击是会立即切换场景", "是", "否"))
                {
                    //当点击“是”，执行运行时场景加载
                    SceneManager.LoadScene(scenepath);
                }
                else
                {
                    EditorApplication.isPlaying = false;
                    void Load(PlayModeStateChange state)
                    {
                        if (!EditorApplication.isPlaying)
                        {
                            EditorSceneManager.OpenScene(scenepath);
                            EditorApplication.playModeStateChanged -= Load;
                        }
                    }
                    EditorApplication.delayCall = () =>
                    {
                        if (!EditorApplication.isPlaying)
                            EditorSceneManager.OpenScene(scenepath);
                        else
                            EditorApplication.playModeStateChanged += Load;
                    };
                }
                return;
            }
            else
```

```
                    EditorSceneManager.OpenScene(scenepath);
                }
            }
        }
    }
```

2) FileUtility

PS：

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

namespace GameUtil
{
    public class FileUtility
    {
        /// <summary>
        ///  全路径转换为 Asset 内相对路径
        /// </summary>
        /// <param name="fullPath">全路径</param>
        /// <returns>Asset 内相对路径</returns>
        public static string FullPathToAssetPath(string fullPath)
        {
            return FormatToUnityPath(fullPath).Replace(Application.dataPath, "Assets");
        }

        /// <summary>
        ///  获取文件路径
        /// </summary>
        /// <param name="directPath">直接路径</param>
        /// <param name="searchPattern"></param>
        /// <param name="searchOption">搜索选项，AllDirectories 为在搜索操作中包括当前目录和所有它的子目录。</param>
        /// <returns></returns>
        public static string[] GetFilesPaths(string directPath, string searchPattern = "*", SearchOption searchOption = SearchOption.AllDirectories)
        {
            List<string> paths = new List<string>();
            FileInfo[] files = GetFiles(directPath, searchPattern, searchOption);
            string tempStr;
            if (files != null && files.Length > 0)
            {
                foreach (FileInfo file in files)
                {
```

```
                    //将完整路径转换为 Unity 里的相对路径
                    tempStr = FullPathToAssetPath(file.FullName);
                    paths.Add(tempStr);
                }
            }
            return paths.ToArray();
        }

        /// <summary>
        ///  获取文件
        /// </summary>
        /// <param name="directPath">绝对路径</param>
        /// <param name="searchPattern">要匹配的字符</param>
        /// <param name="searchOption">搜索选项，AllDirectories 为在搜索操作中包括当前目录和所有它的子目录。</param>
        /// <returns></returns>
        public static FileInfo[] GetFiles(string directPath, string searchPattern = "*", SearchOption searchOption = SearchOption.AllDirectories)
        {
            //测试目录是否存在或者在尝试确定指定目录是否存在时出错
            if (Directory.Exists(directPath))
            {
                DirectoryInfo directoryInfo = new DirectoryInfo(directPath);
                return directoryInfo.GetFiles(searchPattern, searchOption);
            }

            Debug.LogError($"文件夹不存在{directPath}");
            return null;
        }

        /// <summary>
        ///  路径格式转换成 Unity 下的路径格式
        /// </summary>
        /// <param name="path">原始路径</param>
        /// <returns></returns>
        public static string FormatToUnityPath(string path)
        {
            return path.Replace("\\", "/");
        }

        /// <summary>
        ///  获取文件名称
        /// </summary>
        /// <param name="filePath">文件路径</param>
```

```
            /// <param name="withoutExtension">是否不需要扩展名称</param>
            /// <returns></returns>
            public static string GetFileName(string filePath, bool withoutExtension = true)
            {
                if (withoutExtension)
                {
                    return
FormatToUnityPath(Path.GetFileNameWithoutExtension(filePath));
                }

                return FormatToUnityPath(Path.GetFileName(filePath));
            }
        }
    }
```

3) ScenesChooseTool

PS：

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

public class ScenesChooseTool : EditorWindow
{
    /// <summary>
    /// 滑动列表的起始位置
    /// </summary>
    private Vector2 startScrollPos = Vector2.zero;
    /// <summary>
    /// 场景名称列表
    /// </summary>
    private List<string> scenesNames;
    /// <summary>
    /// 场景路径列表
    /// </summary>
    private List<string> scenesPaths;
    private bool isInitOver = false;

    [MenuItem("游戏工具/场景选择工具", false, 0)]
    public static void Open()
    {
        ScenesChooseTool Window = GetWindow<ScenesChooseTool>();
        Window.titleContent = new
GUIContent(EditorGUIUtility.IconContent("TerrainInspector.TerrainToolRaise"))
```

```
        {
            text = "场景选择工具",
        };
        Window.Init();
    }

    /// <summary>
    ///  初始化窗口数据
    /// </summary>
    private void Init()
    {
        scenesName = new List<string>();
        scenesPath = new List<string>();
        //实际遍历的路径
        List<string> paths = new List<string>()
        {
            Application.dataPath+"/Scenes",
        };

        foreach (string path in paths)
        {
            string[] files = FileUtility.GetFilesPaths(path, "*Unity");
            if (files != null && files.Length > 0)
            {
                foreach (string file in files)
                {
                    scenesPath.Add(file);
                    scenesName.Add(FileUtility.GetFileName(file));
                }
            }
        }
        isInitOver = true;
    }

    public void OnGUI()
    {
        //使用 GroupBox 的 GUI 样式
        GUILayout.BeginVertical("GroupBox");
        #region  大标题
        GUI.color = Color.green;
        GUILayout.Space(10);
        GUI.skin.label.fontSize = 24;
        //居中对齐
        GUI.skin.label.alignment = TextAnchor.MiddleCenter;
```

```
            GUILayout.Label("请选择场景");
            GUI.color = Color.white;
            #endregion
            startScrollPos = GUILayout.BeginScrollView(startScrollPos);

            if (isInitOver && scenesPaths != null)
            {
                for (int i = 0; i < scenesPaths.Count; i++)
                {
                    if (GUILayout.Button(scenesNames[i]))
                    {
                        SceneUtility.OpenScene(scenesPaths[i]);
                    }
                }
            }

            GUI.skin.label.fontSize = 16;
            GUI.skin.label.alignment = TextAnchor.MiddleLeft;
            GUILayout.EndScrollView();
            GUILayout.EndVertical();
        }
    }
```

四. 参考图添加工具

1. 初始化逻辑代码参数

对于这一个工具，我们需要先在类前添加 InitializeOnLoad 特性。使用 InitializeOnLoad 特性，在编辑器启动的时候就会调用此类的构造函数。

接着添加一系列的常量，这些常量将在主逻辑代码里得到应用。

PS：

```
        #region Normal Properties
        /// <summary>
        /// 默认查找的参考图放置节点名称
        /// </summary>
        private const string DefaultUIRefName = "参考图 RawImage";
        /// <summary>
        /// 没有默认放置节点，新创建的节点名称
        /// </summary>
        private const string NewUIRefName = "New-参考图 RawImage";

        private const string UseGUIReinforceMenuItemPath = "游戏工具/GUI 增强/开启 Scene 视图右键菜单";
        private const string AddRefImgInProjectPath = "Assets/设置为参考图";
        private const string AddRefImgMenuItemPath = "GameObject/参考图";

        /// <summary>
```

```
        /// 是否是鼠标右键点击
        /// </summary>
        private static bool enableRightClick = false;

        //选择参考图的文件过滤
        private static string[] refImageFilter = new string[]
        {
            "图片文件", "png,jpg,jpeg,gif"
        };

        //允许的参考图后缀
        private static List<string> enableRefPictureSuffix = new List<string>()
        {
            "png", "jpg", "jpeg", "gif"
        };
        #endregion
```

2. 初始化编辑器参数

因为，我们需要使这一个工具能够记住用户之前的选择。因此，我们就需要像使用 Unity PlayerPrefs 一样，根据关键字去获取与设置持久化数据。

PS：

```
        #region EditorPrefs Properties
        //是否开启 Scene 右键菜单的 key
        private const string SceneGenericMenuKey = "EnableSceneGenericMenu";
        private static int enableSceneGenericMenu = -1;
        /// <summary>
        /// 使用属性对变量 enableSceneGenericMenu 进行封装
        /// </summary>
        internal static bool EnableSceneGenericMenu
        {
            get
            {
                if (enableSceneGenericMenu == -1)
                {
                    //根据关键字获取 Unity editor 偏好设置文件中的值，偏好设置文件中不存在此关键字就返回默认值
                    enableSceneGenericMenu = EditorPrefs.GetInt(SceneGenericMenuKey, 1);
                }
                return enableSceneGenericMenu == 1;
            }
            set
            {
                int newValue = value ? 1 : 0;
                //设置的新值和旧值不相等时，我们才执行赋值操作
```

```
                if (newValue != enableSceneGenericMenu)
                {
                    enableSceneGenericMenu = newValue;
                    EditorPrefs.SetInt(SceneGenericMenuKey, enableSceneGenericMenu);
                }
            }
        }

        //上一次选择的参考图路径，默认为空
        private static string lastChoosePath = null;
        private const string LastChoosePathKey = "RefImgLastChoosePath";
        /// <summary>
        ///  使用属性对变量 lastChoosePath 进行封装
        /// </summary>
        private static string LastChoosePath
        {
            get
            {
                //上次的选择路径为空是，我们就根据关键字去获取一下 Unity editor 偏好设置文件中的值
                if (lastChoosePath == null)
                    lastChoosePath = EditorPrefs.GetString(LastChoosePathKey, Application.dataPath);
                return lastChoosePath;
            }
            set
            {
                //设置的新值和旧值不相等时，我们才执行赋值操作
                if (!lastChoosePath.Equals(value))
                {
                    lastChoosePath = value;
                    EditorPrefs.SetString(LastChoosePathKey, lastChoosePath);
                }
            }
        }
        #endregion
```

3. 主逻辑代码

1) 添加 Scene 视图的事件处理器

订阅了 duringSceneGui 事件，就可以在 Scene 视图每次调用 OnGUI 方法时执行对应的事件处理器。

在这个事件处理器中，我们主要是判断是否是右键点击，如果是右键点击且开启了 Scene 视图扩展，就在 Scene 视图中显示菜单。

PS：

```
        private static void OnSceneViewGUI(SceneView sceneView)
```

```
        {
            Event curEvent = Event.current;
            //根据将被处理的当前事件类型，去执行不同的行为
            switch (curEvent.type)
            {
                case EventType.MouseDown:
                    //点击鼠标右键
                    enableRightClick = (curEvent.button == 1);
                    break;
                case EventType.MouseUp:
                    if (EnableSceneGenericMenu && enableRightClick)
                    {
                        if (!CheckIsPlayingAndIsInPrefab())
                        {
                            //使用 GenericMenu  我们可以创建自定义上下文菜单和下拉菜单，这里用于 Scene 视图右键菜单扩展
                            GenericMenu menu = new GenericMenu();
                            //添加一个菜单项，第一个参数为显示的文本，第三个参数为要调用的函数
                            menu.AddItem(new GUIContent("添加参考图"), false, AddRefImage);
                            menu.AddItem(new GUIContent("撤销参考图"), false, RemoveRefImage);
                            //右键单击时在鼠标下显示菜单。
                            menu.ShowAsContext();
                            curEvent.Use();
                        }
                        else
                        {
                            Debug.LogWarning("Prefab 视图无法打开右键菜单");
                        }
                    }
                    break;
                case EventType.MouseDrag:
                    //拖动鼠标后不显示菜单
                    enableRightClick = false;
                    break;
            }
        }
```

2) 添加参考图

要添加参考图片，我们首先需要先从磁盘中读取图片，然后再在场景中查找挂载参考图的对象并在代码中调整 RawImage 的各项设置。

PS：

```
        #region  添加图片
```

```
/// <summary>
///  从磁盘中读取图片
/// </summary>
/// <param name="path">路径</param>
/// <param name="width">纹理的宽度</param>
/// <param name="height">纹理的高度</param>
/// <returns></returns>
private static Texture2D LoadImage(string path, int width, int height)
{
    if (path == null || path.Length == 0)
    {
        Debug.LogError("待加载的图片路径为空!");
        return null;
    }

    FileStream fs = new FileStream(path, FileMode.Open, FileAccess.Read);
    //将文件的指针移动到开头。
    fs.Seek(0, SeekOrigin.Begin);
    byte[] bytes = new byte[fs.Length];
    //读取文件
    fs.Read(bytes, 0, (int)fs.Length);
    fs.Close();
    //使用 Dispose 销毁对象并释放资源
    fs.Dispose();

    Texture2D tex = new Texture2D(width, height, TextureFormat.RGBA32, false);
    tex.LoadImage(bytes);
    return tex;
}

/// <summary>
///  加载并添加参考图
/// </summary>
/// <param name="path">路径</param>
private static void LoadAndAddRefImage(string path)
{
    Texture2D tex = LoadImage(path, 1280, 720);

    if (tex != null)
    {
        //先找 NewUIRefName，为空则找 UIRefDefaultName
        GameObject root = GameObject.Find(NewUIRefName);
        if (null == root)
        {
```

```
                if ((root = GameObject.Find(DefaultUIRefName)) == null)
                {
                    //在场景中查找 Canvas
                    Canvas canvas = Object.FindObjectOfType(typeof(Canvas)) as
Canvas;
                    if (canvas == null)
                    {
                        Debug.LogError("场景中不存在 Canvas 组件!");
                        return;
                    }
                    //新建一个 GameObject，后面的参数为在创建时要添加到
GameObject 的 Components 列表。
                    root = new GameObject(NewUIRefName, typeof(RectTransform),
typeof(RawImage));
                    root.transform.SetParent(canvas.gameObject.transform);
                    //前面已经明确了，组件里有 RectTransform，因此就不需要使用
TryGetComponent 了
                    RectTransform rect = root.GetComponent<RectTransform>();
                    //设置描点位置
                    rect.anchorMin = new Vector2(0.5f, 0.5f);
                    rect.anchorMax = new Vector2(0.5f, 0.5f);
                    rect.anchoredPosition = new Vector2(0.5f, 0.5f);
                    //设置 RectTransform 相对于锚点之间距离的大小
                    rect.sizeDelta = new Vector2(1920, 1080);
                    rect.transform.localPosition = Vector3.zero;
                    rect.transform.localScale = Vector3.one;
                }
            }
            RawImage rawImage;
            root.TryGetComponent<RawImage>(out rawImage);
            if (rawImage == null)
            {
                Debug.Log(string.Format("未找到 RawImage 或 AorRawImage 组件:
{0}，将添加 RawImage 组件!", root.name));
                rawImage = root.AddComponent<RawImage>();
            }
            //设置纹理的各向异性过滤级别，值范围为 1 到 9，其中 1 等于不应用
过滤，9 等于应用完全过滤。
            tex.anisoLevel = 8;
            //设置纹理的过滤模式
            tex.filterMode = FilterMode.Point;
            Undo.RecordObject(rawImage, "新增参考图像名称：" + rawImage.name);
            rawImage.texture = tex;
            rawImage.color = new Color(1, 1, 1, 1);
```

```
                rawImage.enabled = true;
                //将 rawImage 的大小设置为纹理的大小
                rawImage.SetNativeSize();
                root.SetActive(true);
            }
            else
            {
                Debug.LogError("图片读取失败: " + path);
            }
        }
        #endregion
```

3) 移除参考图

移除参考图就相当的简单了，只要将对象上的 RawImage 纹理设置为空并取消组件的使用就可以了。

PS：

```
        #region  移除参考图
        /// <summary>
        ///  从节点上移除参考图
        /// </summary>
        /// <param name="root">要执行移除行为的节点</param>
        private static void RemoveTexture(GameObject root)
        {
            if (root == null)
                return;
            RawImage rawImage;
            //尝试获取对象里的 RawImage 组件
            root.TryGetComponent<RawImage>(out rawImage);
            if (rawImage == null)
            {
                Debug.Log(string.Format("未找到 RawImage 组件: {0}，参考图移除失败!",
root.name));
                return;
            }
            Undo.RecordObject(rawImage, "取消设置 rawImage" + rawImage.name);
            Texture tex = rawImage.texture;
            rawImage.texture = null;
            rawImage.color = new Color(1, 1, 1, 0);
            rawImage.enabled = false;
        }
        #endregion
```

4. 添加菜单项

如果 MenuItem 的 validate 设为 ture，则在点击菜单前就会进行调用，主要用于检测按钮是否要可用。如果可以点击，那么点击后就会执行对应的方法，去加载或者移除参考图。

1) 设置是否启用 Scene 视图右键菜单

PS：

```
#region 设置是否启用 Scene 视图右键菜单
[MenuItem(UseGUIReinforceMenuItemPath, false)]
public static void UseSceneGenericMenu()
{
    EnableSceneGenericMenu = !EnableSceneGenericMenu;
}

[MenuItem(UseGUIReinforceMenuItemPath, true)]
public static bool SetUseSceneGenericMenu()
{
    //设置菜单的勾选状态
    Menu.SetChecked(UseGUIReinforceMenuItemPath, EnableSceneGenericMenu);
    return true;
}
#endregion
```

2) 添加参考图

PS：

```
#region 添加参考图
[MenuItem(AddRefImgMenuItemPath + "/添加", priority = 25, validate = false)]
private static void AddRefImage()
{
    //使用 OpenFilePanelWithFilters“打开文件”对话框并返回所选的路径名称。
    string path = EditorUtility.OpenFilePanelWithFilters("选择参考图",
LastChoosePath, refImageFilter);
    if (path == null || path.Length == 0)
    {
        return;
    }
    LastChoosePath = path;
    LoadAndAddRefImage(path);
}

//validate 设为 ture 则点击菜单前就会进行调用，主要用于检测按钮是否要显示
[MenuItem(AddRefImgMenuItemPath + "/添加", priority = 25, validate = true)]
private static bool SetAddRefImageItemState()
{
    return !CheckIsPlayingAndIsInPrefab();
}
#endregion
```

3) 在 project 视图里选择图像作为参考图

PS：

```
#region  在 project 视图里选择图像作为参考图
[MenuItem(AddRefImgInProjectPath, priority = 25, validate = false)]
private static void AddRefImageInProject()
{
    string[] guids = Selection.assetGUIDs;
    if (CheckRefImageSuffix(guids))
    {
        string path = AssetDatabase.GUIDToAssetPath(guids[0]);
        LoadAndAddRefImage(path);
    }
}

[MenuItem(AddRefImgInProjectPath, priority = 25, validate = true)]
private static bool SetAddRefImageInProjectItemState()
{
    string[] guids = Selection.assetGUIDs;
    return !CheckIsPlayingAndIsInPrefab() && CheckRefImageSuffix(guids);
}
#endregion
```

4) 移除参考图

PS：

```
[MenuItem(AddRefImgMenuItemPath + "/移除", priority = 25, validate = false)]
private static void RemoveRefImage()
{
    //在场景中查找节点
    GameObject root = GameObject.Find(DefaultUIRefName);
    GameObject newRoot = GameObject.Find(NewUIRefName);
    if (root == null && newRoot == null)
    {
        Debug.Log(string.Format("未找到默认的 UI 参考图根节点: {0}或{1}!",
DefaultUIRefName, NewUIRefName));
        return;
    }
    RemoveTexture(root);
    RemoveTexture(newRoot);
}

[MenuItem(AddRefImgMenuItemPath + "/移除", priority = 25, validate = true)]
private static bool SetRemoveRefImageItemState()
{
    return !CheckIsPlayingAndIsInPrefab();
}
```

5. 完整代码

PS：

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

/// <summary>
/// 参考图片添加工具
/// </summary>
//使用 InitializeOnLoad 特性，在编辑器启动的时候调用此类的构造函数
[InitializeOnLoad]
public class AddReferenceImageTool
{
    #region Normal Properties
    /// <summary>
    /// 默认查找的参考图放置节点名称
    /// </summary>
    private const string DefaultUIRefName = "参考图 RawImage";
    /// <summary>
    /// 没有默认放置节点，新创建的节点名称
    /// </summary>
    private const string NewUIRefName = "New-参考图 RawImage";

    private const string UseGUIReinforceMenuItemPath = "游戏工具/GUI 增强/开启
Scene 视图右键菜单";
    private const string AddRefImgInProjectPath = "Assets/设置为参考图";
    private const string AddRefImgMenuItemPath = "GameObject/参考图";

    /// <summary>
    /// 是否是鼠标右键点击
    /// </summary>
    private static bool enableRightClick = false;

    //选择参考图的文件过滤
    private static string[] refImageFilter = new string[]
    {
        "图片文件", "png,jpg,jpeg,gif"
    };

    //允许的参考图后缀
    private static List<string> enableRefPictureSuffix = new List<string>()
    {
        "png", "jpg", "jpeg", "gif"
    };
```

```
        #endregion

        #region EditorPrefs Properties
        //是否开启 Scene 右键菜单的 key
        private const string SceneGenericMenuKey = "EnableSceneGenericMenu";
        private static int enableSceneGenericMenu = -1;
        /// <summary>
        ///  使用属性对变量 enableSceneGenericMenu 进行封装
        /// </summary>
        internal static bool EnableSceneGenericMenu
        {
            get
            {
                if (enableSceneGenericMenu == -1)
                {
                    //根据关键字获取 Unity editor 偏好设置文件中的值，偏好设置文件中不存在此关键字就返回默认值
                    enableSceneGenericMenu = EditorPrefs.GetInt(SceneGenericMenuKey, 1);
                }
                return enableSceneGenericMenu == 1;
            }
            set
            {
                int newValue = value ? 1 : 0;
                //设置的新值和旧值不相等时，我们才执行赋值操作
                if (newValue != enableSceneGenericMenu)
                {
                    enableSceneGenericMenu = newValue;
                    EditorPrefs.SetInt(SceneGenericMenuKey, enableSceneGenericMenu);
                }
            }
        }

        //上一次选择的参考图路径，默认为空
        private static string lastChoosePath = null;
        private const string LastChoosePathKey = "RefImgLastChoosePath";
        /// <summary>
        ///  使用属性对变量 lastChoosePath 进行封装
        /// </summary>
        private static string LastChoosePath
        {
            get
            {
```

```
                //上次的选择路径为空是，我们就根据关键字去获取一下 Unity editor 偏好设置文件中的值
                if (lastChoosePath == null)
                    lastChoosePath = EditorPrefs.GetString(LastChoosePathKey, Application.dataPath);
                return lastChoosePath;
            }
            set
            {
                //设置的新值和旧值不相等时，我们才执行赋值操作
                if (!lastChoosePath.Equals(value))
                {
                    lastChoosePath = value;
                    EditorPrefs.SetString(LastChoosePathKey, lastChoosePath);
                }
            }
        }
        #endregion

        //无参构造函数
        static AddReferenceImageTool()
        {
            //订阅 duringSceneGui 事件，可以在 Scene 视图每次调用 OnGUI 方法时执行对应的事件处理器
            SceneView.duringSceneGui += OnSceneViewGUI;
        }

        //析构函数，析构函数的名称与类名相同，不过需要在名称的前面加上一个波浪号~作为前缀
        ~AddReferenceImageTool()
        {
            SceneView.duringSceneGui -= OnSceneViewGUI;
        }

        /// <summary>
        ///  检测是否在预制体里或者处于游戏运行状态
        /// </summary>
        /// <returns></returns>
        private static bool CheckIsPlayingAndIsInPrefab()
        {

return !(UnityEditor.Experimental.SceneManagement.PrefabStageUtility.GetCurrentPrefabStage() == null && !Application.isPlaying);
    // return !(UnityEditor.SceneManagement.PrefabStageUtility.GetCurrentPrefabStage() ==
```

```
null && !Application.isPlaying);//2022 版本使用
        }

        #region 添加与移除参考图的逻辑代码
        private static void OnSceneViewGUI(SceneView sceneView)
        {
            Event curEvent = Event.current;
            //根据将被处理的当前事件类型，去执行不同的行为
            switch (curEvent.type)
            {
                case EventType.MouseDown:
                    //点击鼠标右键
                    enableRightClick = (curEvent.button == 1);
                    break;
                case EventType.MouseUp:
                    if (EnableSceneGenericMenu && enableRightClick)
                    {
                        if (!CheckIsPlayingAndIsInPrefab())
                        {
                            //使用 GenericMenu 我们可以创建自定义上下文菜单和下
拉菜单，这里用于 Scene 视图右键菜单扩展
                            GenericMenu menu = new GenericMenu();
                            //添加一个菜单项，第一个参数为显示的文本，第三个参数
为要调用的函数
                            menu.AddItem(new GUIContent("添加参考图"), false,
AddRefImage);
                            menu.AddItem(new GUIContent("撤销参考图"), false,
RemoveRefImage);
                            //右键单击时在鼠标下显示菜单。
                            menu.ShowAsContext();
                            curEvent.Use();
                        }
                        else
                        {
                            Debug.LogWarning("Prefab 视图无法打开右键菜单");
                        }
                    }
                    break;
                case EventType.MouseDrag:
                    //拖动鼠标后不显示菜单
                    enableRightClick = false;
                    break;
            }
        }
```

```
#region 添加图片
/// <summary>
/// 从磁盘中读取图片
/// </summary>
/// <param name="path">路径</param>
/// <param name="width">纹理的宽度</param>
/// <param name="height">纹理的高度</param>
/// <returns></returns>
private static Texture2D LoadImage(string path, int width, int height)
{
    if (path == null || path.Length == 0)
    {
        Debug.LogError("待加载的图片路径为空!");
        return null;
    }

    FileStream fs = new FileStream(path, FileMode.Open, FileAccess.Read);
    //将文件的指针移动到开头。
    fs.Seek(0, SeekOrigin.Begin);
    byte[] bytes = new byte[fs.Length];
    //读取文件
    fs.Read(bytes, 0, (int)fs.Length);
    fs.Close();
    //使用 Dispose 销毁对象并释放资源
    fs.Dispose();

    Texture2D tex = new Texture2D(width, height, TextureFormat.RGBA32, false);
    tex.LoadImage(bytes);
    return tex;
}

/// <summary>
/// 加载并添加参考图
/// </summary>
/// <param name="path">路径</param>
private static void LoadAndAddRefImage(string path)
{
    Texture2D tex = LoadImage(path, 1280, 720);

    if (tex != null)
    {
        //先找 NewUIRefName，为空则找 UIRefDefaultName
        GameObject root = GameObject.Find(NewUIRefName);
```

```
                if (null == root)
                {
                    if ((root = GameObject.Find(DefaultUIRefName)) == null)
                    {
                        //在场景中查找 Canvas
                        Canvas canvas = Object.FindObjectOfType(typeof(Canvas)) as
Canvas;
                        if (canvas == null)
                        {
                            Debug.LogError("场景中不存在 Canvas 组件!");
                            return;
                        }
                        //新建一个 GameObject，后面的参数为在创建时要添加到
GameObject 的 Components 列表。
                        root = new GameObject(NewUIRefName, typeof(RectTransform),
typeof(RawImage));
                        root.transform.SetParent(canvas.gameObject.transform);
                        //前面已经明确了，组件里有 RectTransform，因此就不需要使用
TryGetComponent 了
                        RectTransform rect = root.GetComponent<RectTransform>();
                        //设置描点位置
                        rect.anchorMin = new Vector2(0.5f, 0.5f);
                        rect.anchorMax = new Vector2(0.5f, 0.5f);
                        rect.anchoredPosition = new Vector2(0.5f, 0.5f);
                        //设置 RectTransform 相对于锚点之间距离的大小
                        rect.sizeDelta = new Vector2(1920, 1080);
                        rect.transform.localPosition = Vector3.zero;
                        rect.transform.localScale = Vector3.one;
                    }
                }
                RawImage rawImage;
                root.TryGetComponent<RawImage>(out rawImage);
                if (rawImage == null)
                {
                    Debug.Log(string.Format("未找到 RawImage 或 AorRawImage 组件:
{0}，将添加 RawImage 组件!", root.name));
                    rawImage = root.AddComponent<RawImage>();
                }
                //设置纹理的各向异性过滤级别，值范围为 1 到 9，其中 1 等于不应用
过滤，9 等于应用完全过滤。
                tex.anisoLevel = 8;
                //设置纹理的过滤模式
                tex.filterMode = FilterMode.Point;
                Undo.RecordObject(rawImage, "新增参考图像名称：" + rawImage.name);
```

```
                rawImage.texture = tex;
                rawImage.color = new Color(1, 1, 1, 1);
                rawImage.enabled = true;
                //将 rawImage 的大小设置为纹理的大小
                rawImage.SetNativeSize();
                root.SetActive(true);
            }
            else
            {
                Debug.LogError("图片读取失败: " + path);
            }
        }
        #endregion

        #region  移除参考图
        /// <summary>
        ///  从节点上移除参考图
        /// </summary>
        /// <param name="root">要执行移除行为的节点</param>
        private static void RemoveTexture(GameObject root)
        {
            if (root == null)
                return;
            RawImage rawImage;
            //尝试获取对象里的 RawImage 组件
            root.TryGetComponent<RawImage>(out rawImage);
            if (rawImage == null)
            {
                Debug.Log(string.Format("未找到 RawImage 组件: {0}，参考图移除失败!",
root.name));
                return;
            }
            Undo.RecordObject(rawImage, "取消设置 rawImage" + rawImage.name);
            Texture tex = rawImage.texture;
            rawImage.texture = null;
            rawImage.color = new Color(1, 1, 1, 0);
            rawImage.enabled = false;
        }
        #endregion
        #endregion

        #region Editor Menu Item

        /// <summary>
```

```
        /// 检查参考图的文件后缀
        /// </summary>
        /// <param name="guids">GUID 数组</param>
        /// <returns></returns>
        private static bool CheckRefImageSuffix(string[] guids)
        {
            //每次只允许选择一张图
            if (guids.Length != 1)
            {
                Debug.LogWarning("每次只允许选择一张图");
                return false;
            }
            string[] splitedPath = AssetDatabase.GUIDToAssetPath(guids[0]).Split('.');
            return enableRefPictureSuffix.Contains(splitedPath[splitedPath.Length - 1]);
        }

        #region 设置是否启用 Scene 视图右键菜单
        [MenuItem(UseGUIReinforceMenuItemPath, false)]
        public static void UseSceneGenericMenu()
        {
            EnableSceneGenericMenu = !EnableSceneGenericMenu;
        }

        [MenuItem(UseGUIReinforceMenuItemPath, true)]
        public static bool SetUseSceneGenericMenu()
        {
            //设置菜单的勾选状态
            Menu.SetChecked(UseGUIReinforceMenuItemPath, EnableSceneGenericMenu);
            return true;
        }
        #endregion

        #region 添加参考图
        [MenuItem(AddRefImgMenuItemPath + "/添加", priority = 25, validate = false)]
        private static void AddRefImage()
        {
            //使用 OpenFilePanelWithFilters“打开文件”对话框并返回所选的路径名称。
            string path = EditorUtility.OpenFilePanelWithFilters("选择参考图",
LastChoosePath, refImageFilter);
            if (path == null || path.Length == 0)
            {
                return;
            }
            LastChoosePath = path;
```

```
            LoadAndAddRefImage(path);
        }

        //validate 设为 ture 则点击菜单前就会进行调用，主要用于检测按钮是否要显示
        [MenuItem(AddRefImgMenuItemPath + "/添加", priority = 25, validate = true)]
        private static bool SetAddRefImageItemState()
        {
            return !CheckIsPlayingAndIsInPrefab();
        }
        #endregion

        #region  在 project 视图里选择图像作为参考图
        [MenuItem(AddRefImgInProjectPath, priority = 25, validate = false)]
        private static void AddRefImageInProject()
        {
            string[] guids = Selection.assetGUIDs;
            if (CheckRefImageSuffix(guids))
            {
                string path = AssetDatabase.GUIDToAssetPath(guids[0]);
                LoadAndAddRefImage(path);
            }
        }

        [MenuItem(AddRefImgInProjectPath, priority = 25, validate = true)]
        private static bool SetAddRefImageInProjectItemState()
        {
            string[] guids = Selection.assetGUIDs;
            return !CheckIsPlayingAndIsInPrefab() && CheckRefImageSuffix(guids);
        }
        #endregion

        #region  移除参考图
        [MenuItem(AddRefImgMenuItemPath + "/移除", priority = 25, validate = false)]
        private static void RemoveRefImage()
        {
            //在场景中查找节点
            GameObject root = GameObject.Find(DefaultUIRefName);
            GameObject newRoot = GameObject.Find(NewUIRefName);
            if (root == null && newRoot == null)
            {
                Debug.Log(string.Format("未找到默认的 UI 参考图根节点: {0}或{1}!",
DefaultUIRefName, NewUIRefName));
                return;
            }
```

```
            RemoveTexture(root);
            RemoveTexture(newRoot);
        }

        [MenuItem(AddRefImgMenuItemPath + "/移除", priority = 25, validate = true)]
        private static bool SetRemoveRefImageItemState()
        {
            return !CheckIsPlayingAndIsInPrefab();
        }
        #endregion
        #endregion
    }
```

五. Hierarchy 增强显示

1. 效果说明

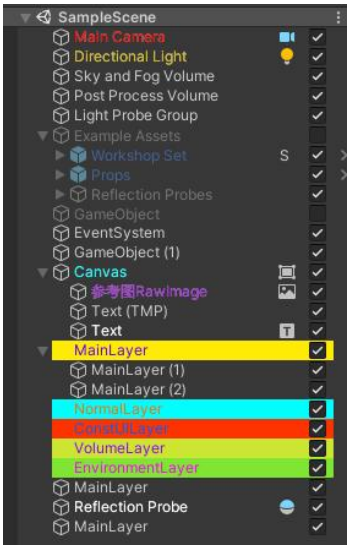

图 5. 效果图

从上面可以看到，开启了增强显示后的 Hierarchy 窗口，具有了一系列额外的内在右侧显示出了组件的图标和是否激活的选项。组件的增强显示效果，是交给用户自定义的。

具体的数据是使用 ScriptableObject 进行存储的，如果项目中没有该 ScriptableObject，我们可以在 Project 窗口中点击鼠标右键，在配置创建选项菜单下，找到创建自定义 Hierarchy 配置就可以创建一个新的配置文件了。

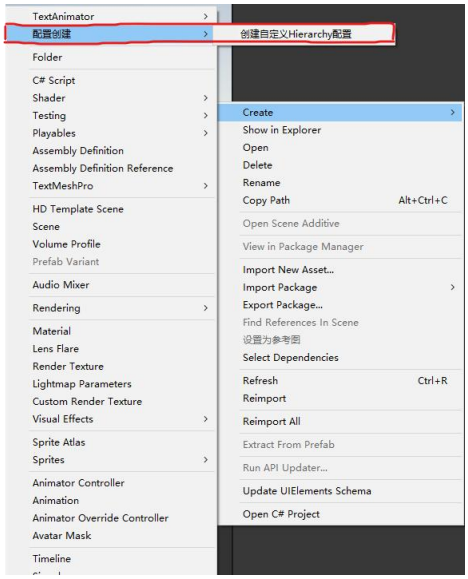

图 6.　　创建增强显示的自定义配置

不过有一点需要注意，那就是配置的读取路径是写死在代码中的。如果要移动配置的位置，则代码中的路径也需要一并修改。

```
/// <summary>
/// 自定义Hierarchy的配置路径
/// </summary>
private const string CustomConfigPath = "Assets/Editor/GUI/CustomHiererchyConfig.asset";
```

图 7.　　CustomHierarchy 中的配置文件读取路径

在该配置上方可以设置一些基本的增强显示参数，在下方就是自定义区块颜色和组件了。如果我们启用了区块显示，Hierarchy 中与配置名称相同的对象，其背景颜色就会变成自定义的区块颜色。

在配置的最底部是组件的自定义增强显示。我们需要设置正确的程序集名称和旗下的组件名称，在代码中就会通过反射去判断 Hierarchy 对象上，是否具有这些组件。如果具有这些组件，则按照自定义颜色去设置对象在 Hierarchy 面板上的字体颜色，并在其右侧添加上对应的内置图标。

图 8.　　ScriptableObject 配置

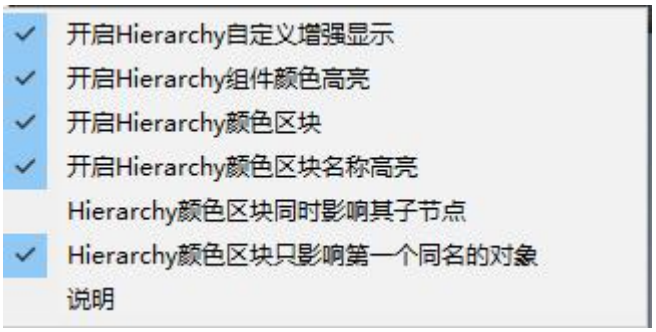

图 9.　　增强效果选项

2. 创建字段特性

在 Unity 内置的特性中，目前还没有可以为参数设置别名，并将别名显示在 Inspector 面板上的特性。这个特性就需要我们自己去编写了，首先需要让其继承 PropertyAttribute，并为其设置一系列的构造函数。

PS：

```
using UnityEngine;

/// <summary>
/// 继承 PropertyAttribute 的类，便能使用一个自定义的 PropertyDrawer 类
/// 去控制其在 Inspector 面板上的显示。
/// </summary>
public class LabelAttribute : PropertyAttribute
{
```

```
        public string Name { get; private set; }

        /// <summary>
        ///  构造函数
        /// </summary>
        /// <param name="inputName">Label 名称</param>
        public LabelAttribute(string inputName)
        {
            this.Name = inputName;
        }
    }
```

与之配套的，我们需要使用 PropertyDrawer，去设置该特性在 Inspector 面板上应该以何种方式进行显示。

PS：

```
    using UnityEditor;
    using UnityEngine;

    /// <summary>
    ///  使用 CustomPropertyDrawer 自定义 PropertyAttribute 的显示。
    ///  只对可序列化的类有效，非可序列化的类没法在 Inspector 面板中显示。
    /// </summary>
    [CustomPropertyDrawer(typeof(LabelAttribute))]
    public class LabelAttributeDrawer : PropertyDrawer
    {
        /// <summary>
        /// OnGUI 方法里只能使用 GUI 相关方法，不能使用 Layout 相关方法。
        /// </summary>
        /// <param name="position">位置</param>
        /// <param name="property">参数</param>
        /// <param name="label">显示的名称</param>
        public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)
        {
            LabelAttribute attr = attribute as LabelAttribute;
            if (attr.Name.Length > 0)
            {
                label.text = attr.Name;
            }
            EditorGUI.PropertyField(position, property, label);
        }
    }
```

3. 创建增强显示的配置

为了便于在 Unity 中进行修改，配置文件就使用 ScriptableObject 去进行实现。对于那些已经确定好要进行增强显示的组件，我们最好让其在 ScriptableObject 中进行初始化。这样

操作后，我们恢复默认值就会非常的方便，不需要再重新去在添加一次组件配置了。

PS：

```
using UnityEngine;
using System;

[CreateAssetMenu(menuName = "配置创建/创建自定义 Hierarchy 配置", fileName = "CustomHiererchyConfig")]
public class CustomHierarchyConfig : ScriptableObject
{
    /// <summary>
    ///  图标 Rect 大小(图标尺寸再-2)
    /// </summary>
    [Label("图标大小")]
    [Space()]
    public int iconRectSize = 18;

    /// <summary>
    ///  第一个(最右侧)图标离右侧距离
    /// </summary>
    [Label("最右侧图标离右侧距离")]
    [Space()]
    public int firstSpacing = 20;

    /// <summary>
    ///  图标之间间距
    /// </summary>
    [Label("图标间距")]
    [Space()]
    public int iconSpacing = 26;

    /// <summary>
    /// insID 距离右侧距离
    /// </summary>
    [Label("InsID 距右侧距离")]
    [Space()]
    public float insIDRightMargin = 200;

    /// <summary>
    ///  图标最小显示宽度
    /// </summary>
    [Label("图标最小显示宽度")]
    [Space()]
    public float minIconShowWidth = 150;
```

```
/// <summary>
///  最小显示宽度
/// </summary>
[Label("最小显示宽度")]
[Space()]
public float minShowWidth = 130;

/// <summary>
///  字体颜色偏移 X
/// </summary>
[Label("字体颜色偏移 X (默认值 18)")]
[Space()]
public float fontColorOffsetX = 18f;

/// <summary>
///  字体颜色 Y 偏移量
/// </summary>
[Label("字体颜色 Y 偏移量  (默认值 0)")]
[Space()]
public float fontColorOffsetY = .0f;

#region  区块颜色
/// <summary>
///  区块颜色模块
/// </summary>
[Header("区块颜色")]
public AreaColorModel[] blockModels =
{
    new AreaColorModel("MainLayer",Color.yellow),
    new AreaColorModel("NormalLayer",Color.cyan),
    new AreaColorModel("ConstUILayer",new Color(1,0.2f,0,1)),
    new AreaColorModel("VolumeLayer",new Color(0.8f,0.9f,0.2f,1)),
    new AreaColorModel("EnvironmentLayer",new Color(0.5f,0.9f,0.2f,1)),
};

/// <summary>
///  区域颜色模块
/// </summary>
[Serializable]
public class AreaColorModel
{
    [Label("目标物体名")]
    public string targetName;
    [Label("区块颜色")]
```

```
        public Color blockColor;

        public AreaColorModel()
        {
            targetName = "";
            blockColor = Color.white;
        }

        public AreaColorModel(string inputName,Color inputColor)
        {
            targetName = inputName;
            blockColor = inputColor;
        }
    }
    #endregion

    #region  设置在 Inpector 上显示的组件图标以及字体颜色
    /// <summary>
    ///  组件颜色高亮模块
    /// </summary>
    [Header("组件图标及颜色高亮")]
    public HighlightColorModel[] highlightModels =
    {
        new HighlightColorModel("UnityEngine",
            new HighlightColorItem[]
            { new HighlightColorItem("Canvas",Color.cyan),
              new HighlightColorItem("Camera",Color.red),
              new HighlightColorItem("SpriteRenderer",new
Color(0.7254f,0.4f,0.8862f,1)),
              new HighlightColorItem("Light",new Color(0.83f,0.73f,0.1f,1)),
              new HighlightColorItem("ReflectionProbe",Color.white),
            }),
        new HighlightColorModel("UnityEngine.UI",
            new HighlightColorItem[]
            {
              new HighlightColorItem("Button",new Color(0,1,0.6f,1)),
              new HighlightColorItem("InputField",Color.white),
              new HighlightColorItem("ScrollRect",Color.white),
              new HighlightColorItem("Scrollbar",Color.white),
              new HighlightColorItem("Toggle",Color.white),
              new HighlightColorItem("ToggleGroup",Color.white),
              new HighlightColorItem("HorizontalLayoutGroup",Color.white),
              new HighlightColorItem("VerticalLayoutGroup",Color.white),
              new HighlightColorItem("RawImage",new Color(0.6274f,0.149f,0.866f,1)),
```

```
                new HighlightColorItem("Image",new Color(0.6274f,0.149f,0.866f,1)),
                new HighlightColorItem("Text",Color.white),
            }),
        new HighlightColorModel("Assembly-CSharp",
            new HighlightColorItem[0]),
};

/// <summary>
///  组件颜色高亮模块
/// </summary>
[Serializable]
public struct HighlightColorModel
{
    [Label("程序集名")]
    public string assemblyName;
    public HighlightColorItem[] items;

    public HighlightColorModel(string inputName, HighlightColorItem[] inputArray)
    {
        assemblyName = inputName;
        items = inputArray;
    }
}

[Serializable]
public class HighlightColorItem
{
    [Label("组件名")]
    public string componentName;
    [Label("名称颜色")]
    public Color color;

    public HighlightColorItem()
    {
        color = Color.white;
    }

    public HighlightColorItem(string inputName, Color inputColor)
    {
        componentName = inputName;
        color = inputColor;
    }
}
#endregion
```

```
}
```

4. 完整代码

1) LabelAttribute

PS：

```
using UnityEngine;

/// <summary>
/// 继承 PropertyAttribute 的类，便能使用一个自定义的 PropertyDrawer 类
/// 去控制其在 Inspector 面板上的显示。
/// </summary>
public class LabelAttribute : PropertyAttribute
{
    public string Name { get; private set; }

    /// <summary>
    /// 构造函数
    /// </summary>
    /// <param name="inputName">Label 名称</param>
    public LabelAttribute(string inputName)
    {
        this.Name = inputName;
    }
}
```

2) LabelAttributeDrawer

PS：

```
using UnityEditor;
using UnityEngine;

/// <summary>
/// 使用 CustomPropertyDrawer 自定义 PropertyAttribute 的显示。
/// 只对可序列化的类有效，非可序列化的类没法在 Inspector 面板中显示。
/// </summary>
[CustomPropertyDrawer(typeof(LabelAttribute))]
public class LabelAttributeDrawer : PropertyDrawer
{
    /// <summary>
    /// OnGUI 方法里只能使用 GUI 相关方法，不能使用 Layout 相关方法。
    /// </summary>
    /// <param name="position">位置</param>
    /// <param name="property">参数</param>
    /// <param name="label">显示的名称</param>
    public override void OnGUI(Rect position, SerializedProperty property, GUIContent
label)
    {
```

```
            LabelAttribute attr = attribute as LabelAttribute;
            if (attr.Name.Length > 0)
            {
                label.text = attr.Name;
            }
            EditorGUI.PropertyField(position, property, label);
        }
    }
```

3) CustomHierarchyConfig

PS：

```
    using UnityEngine;
    using System;
    using TMPro;

    [CreateAssetMenu(menuName = "配置创建/创建自定义 Hierarchy 配置", fileName =
"CustomHiererchyConfig")]
    public class CustomHierarchyConfig : ScriptableObject
    {
        /// <summary>
        ///  图标 Rect 大小(图标尺寸再-2)
        /// </summary>
        [Label("图标大小")]
        [Space()]
        public int iconRectSize = 18;

        /// <summary>
        ///  第一个(最右侧)图标离右侧距离
        /// </summary>
        [Label("最右侧图标离右侧距离")]
        [Space()]
        public int firstSpacing = 20;

        /// <summary>
        ///  图标之间间距
        /// </summary>
        [Label("图标间距")]
        [Space()]
        public int iconSpacing = 26;

        /// <summary>
        /// insID 距离右侧距离
        /// </summary>
        [Label("InsID 距右侧距离")]
        [Space()]
```

```
public float insIDRightMargin = 200;

/// <summary>
///  图标最小显示宽度
/// </summary>
[Label("图标最小显示宽度")]
[Space()]
public float minIconShowWidth = 150;

/// <summary>
///  最小显示宽度
/// </summary>
[Label("最小显示宽度")]
[Space()]
public float minShowWidth = 130;

/// <summary>
///  字体颜色偏移 X
/// </summary>
[Label("字体颜色偏移 X (默认值 18)")]
[Space()]
public float fontColorOffsetX = 18f;

/// <summary>
///  字体颜色 Y 偏移量
/// </summary>
[Label("字体颜色 Y 偏移量  (默认值 0)")]
[Space()]
public float fontColorOffsetY = .0f;

#region  区块颜色
/// <summary>
///  区块颜色模块
/// </summary>
[Header("区块颜色")]
public AreaColorModel[] blockModels =
{
    new AreaColorModel("MainLayer",Color.yellow),
    new AreaColorModel("NormalLayer",Color.cyan),
    new AreaColorModel("ConstUILayer",new Color(1,0.2f,0,1)),
    new AreaColorModel("VolumeLayer",new Color(0.8f,0.9f,0.2f,1)),
    new AreaColorModel("EnvironmentLayer",new Color(0.5f,0.9f,0.2f,1)),
};
```

```
        /// <summary>
        ///  区域颜色模块
        /// </summary>
        [Serializable]
        public class AreaColorModel
        {
            [Label("目标物体名")]
            public string targetName;
            [Label("区块颜色")]
            public Color blockColor;

            public AreaColorModel()
            {
                targetName = "";
                blockColor = Color.white;
            }

            public AreaColorModel(string inputName,Color inputColor)
            {
                targetName = inputName;
                blockColor = inputColor;
            }
        }
        #endregion

        #region  设置在 Inpector 上显示的组件图标以及字体颜色
        /// <summary>
        ///  组件颜色高亮模块
        /// </summary>
        [Header("组件图标及颜色高亮")]
        public HighlightColorModel[] highlightModels =
        {
            new HighlightColorModel("UnityEngine",
                new HighlightColorItem[]
                { new HighlightColorItem("Canvas",Color.cyan),
                  new HighlightColorItem("Camera",Color.red),
                  new HighlightColorItem("SpriteRenderer",new
Color(0.7254f,0.4f,0.8862f,1)),
                  new HighlightColorItem("Light",new Color(0.83f,0.73f,0.1f,1)),
                  new HighlightColorItem("ReflectionProbe",Color.white),
                }),
            new HighlightColorModel("UnityEngine.UI",
                new HighlightColorItem[]
                {
```

```
                new HighlightColorItem("Button",new Color(0,1,0.6f,1)),
                new HighlightColorItem("InputField",Color.white),
                new HighlightColorItem("ScrollRect",Color.white),
                new HighlightColorItem("Scrollbar",Color.white),
                new HighlightColorItem("Toggle",Color.white),
                new HighlightColorItem("ToggleGroup",Color.white),
                new HighlightColorItem("HorizontalLayoutGroup",Color.white),
                new HighlightColorItem("VerticalLayoutGroup",Color.white),
                new HighlightColorItem("RawImage",new Color(0.6274f,0.149f,0.866f,1)),
                new HighlightColorItem("Image",new Color(0.6274f,0.149f,0.866f,1)),
                new HighlightColorItem("Text",Color.white),
            }),
        new HighlightColorModel("Assembly-CSharp",
            new HighlightColorItem[0]),
};

/// <summary>
///  组件颜色高亮模块
/// </summary>
[Serializable]
public struct HighlightColorModel
{
    [Label("程序集名")]
    public string assemblyName;
    public HighlightColorItem[] items;

    public HighlightColorModel(string inputName, HighlightColorItem[] inputArray)
    {
        assemblyName = inputName;
        items = inputArray;
    }
}

[Serializable]
public class HighlightColorItem
{
    [Label("组件名")]
    public string componentName;
    [Label("名称颜色")]
    public Color color;

    public HighlightColorItem()
    {
        color = Color.white;
```

```
            }

            public HighlightColorItem(string inputName, Color inputColor)
            {
                componentName = inputName;
                color = inputColor;
            }
        }
        #endregion
    }
```

4) CustomHierarchy

PS：

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Reflection;
using System.Diagnostics;
using System.Text;
using UnityEngine;
using UnityEditor;
//因为前面引用了 Diagnostics，这里必须指明使用的是哪一个的 Debug
using Debug = UnityEngine.Debug;

//使用 InitializeOnLoad 特性，在编辑器启动的时候调用此类的构造函数
[InitializeOnLoad]
public class CustomHierarchy
{
    #region NormalProperties
    private static GUIStyle iconStyle;
    private static GUIStyle colorLabelStyle;

    private static CustomHierarchyConfig customConfig;
    /// <summary>
    ///  图标 Rect 大小
    /// </summary>
    private static int iconRectSize = 18;
    /// <summary>
    ///  第一个(最右侧)图标离右侧间距
    /// </summary>
    private static int firstSpacing = 20;
    /// <summary>
    ///  图标之间间距
    /// </summary>
    private static int iconSpacing = 26;
```

```
        /// <summary>
        ///  图标最小显示宽度
        /// </summary>
        private static float minIconShowWidth = 170;
        /// <summary>
        ///  最小显示宽度
        /// </summary>
        private static float minShowWidth = 120;

        //颜色字体偏移
        private static float colorFontOffsetX = 18f;
        private static float colorFontOffsetY = 0f;
        /// <summary>
        /// HSV 数值在 0-1 之间 H 为色相，S 为饱和度，V 为明度
        /// </summary>
        private static float H, S, V;

        /// <summary>
        /// Hierarchy 窗口原始背景的颜色
        /// </summary>
        private static Color baseColor;
        /// <summary>
        ///  自定义 Hierarchy 的配置路径
        /// </summary>
        private const string CustomConfigPath =
"Assets/Editor/GUI/CustomHiererchyConfig.asset";

        /// <summary>
        ///  自定义组件颜色高亮列表
        /// </summary>
        private static List<IconInfoModel> iconInfoList = new List<IconInfoModel>();

        /// <summary>
        ///  区块颜色的数据字典，主要是将原本的数组转为字典，便于根据关键字去查找
        /// </summary>
        private static Dictionary<string, Color> areaColorDic = new Dictionary<string,
Color>();
        /// <summary>
        ///  已完成添加的区块名称
        /// </summary>
        private static List<string> alreadyAdditionBlockName = new List<string>();
        /// <summary>
        ///  子物体颜色表
        /// </summary>
```

```
        private static Dictionary<int, Color> subColorDic = new Dictionary<int, Color>();
        #endregion

        #region EditorProperties
        //是否开启自定义 Hierarchy 视图
        private const string CustomHierarchyKey = "EnableCustomHierarchy";
        private static int enableCustomHierarchy = -1;
        internal static bool EnableCustomHierarchy
        {
            get
            {
                if (enableCustomHierarchy == -1)
                {
                    enableCustomHierarchy = EditorPrefs.GetInt(CustomHierarchyKey, 0);
                }
                return enableCustomHierarchy == 1;
            }
            set
            {
                int newValue = value ? 1 : 0;
                if (newValue != enableCustomHierarchy)
                {
                    enableCustomHierarchy = newValue;
                    EditorPrefs.SetInt(CustomHierarchyKey, enableCustomHierarchy);
                }
            }
        }

        //是否开启自定义 Hierarchy 颜色高亮
        private const string CustomHierarchyColorKey = "EnableCustomHierarchyColor";
        private static int enableCustomHierarchyColor = -1;
        internal static bool EnableCustomHierarchyColor
        {
            get
            {
                if (enableCustomHierarchyColor == -1)
                {
                    enableCustomHierarchyColor =
EditorPrefs.GetInt(CustomHierarchyColorKey, 1);
                }
                return enableCustomHierarchyColor == 1;
            }
            set
            {
```

```
                int newValue = value ? 1 : 0;
                if (newValue != enableCustomHierarchyColor)
                {
                    enableCustomHierarchyColor = newValue;
                    EditorPrefs.SetInt(CustomHierarchyColorKey,
enableCustomHierarchyColor);
                }
            }
        }

        //是否开启自定义 Hierarchy 颜色区块
        private const string CustomHierarchyAreaKey = "EnableCustomHierarchyArea";
        private static int enableCustomHierarchyArea = -1;
        internal static bool EnableCustomHierarchyArea
        {
            get
            {
                if (enableCustomHierarchyArea == -1)
                {
                    enableCustomHierarchyArea =
EditorPrefs.GetInt(CustomHierarchyAreaKey, 1);
                }
                return enableCustomHierarchyArea == 1;
            }
            set
            {
                int newValue = value ? 1 : 0;
                if (newValue != enableCustomHierarchyArea)
                {
                    enableCustomHierarchyArea = newValue;
                    EditorPrefs.SetInt(CustomHierarchyAreaKey,
enableCustomHierarchyArea);
                }
            }
        }

        //是否开启自定义颜色区块名称高亮
        private const string CustomHierarchyAreaNameKey =
"EnableCustomHierarchyAreaName";
        private static int enableCustomHierarchyAreaName = -1;
        internal static bool EnableCustomHierarchyAreaName
        {
            get
            {
```

```
                if (enableCustomHierarchyAreaName == -1)
                {
                    enableCustomHierarchyAreaName =
EditorPrefs.GetInt(CustomHierarchyAreaNameKey, 0);
                }
                return enableCustomHierarchyAreaName == 1;
            }
            set
            {
                int newValue = value ? 1 : 0;
                if (newValue != enableCustomHierarchyAreaName)
                {
                    enableCustomHierarchyAreaName = newValue;
                    EditorPrefs.SetInt(CustomHierarchyAreaNameKey,
enableCustomHierarchyAreaName);
                }
            }
        }

        //区块效果是否影响子节点
        private const string CustomHierarchyAreaEffectContainChildKey =
"EnableCustomHierarchyAreaEffectContainChild";
        private static int enableAreaEffectContainChild = -1;
        internal static bool EnableAreaEffectContainChild
        {
            get
            {
                if (enableAreaEffectContainChild == -1)
                {
                    enableAreaEffectContainChild =
EditorPrefs.GetInt(CustomHierarchyAreaEffectContainChildKey, 0);
                }
                return enableAreaEffectContainChild == 1;
            }
            set
            {
                int newValue = value ? 1 : 0;
                if (newValue != enableAreaEffectContainChild)
                {
                    enableAreaEffectContainChild = newValue;
                    EditorPrefs.SetInt(CustomHierarchyAreaEffectContainChildKey,
enableAreaEffectContainChild);
                }
            }
```

```
        }

        //区块效果是否只对场景中的第一个同名对象有用
        private const string CustomHierarchyOnlyFindOneKey =
"EnableCustomHierarchyOnlyFindOne";
        private static int enableOnlyFindOneSameNameGameObject = -1;
        internal static bool EnableOnlyFindOneSameNameGameObject
        {
            get
            {
                if (enableOnlyFindOneSameNameGameObject == -1)
                {
                    enableOnlyFindOneSameNameGameObject =
EditorPrefs.GetInt(CustomHierarchyOnlyFindOneKey, 0);
                }
                return enableOnlyFindOneSameNameGameObject == 1;
            }
            set
            {
                int newValue = value ? 1 : 0;
                if (newValue != enableOnlyFindOneSameNameGameObject)
                {
                    enableOnlyFindOneSameNameGameObject = newValue;
                    EditorPrefs.SetInt(CustomHierarchyOnlyFindOneKey,
enableOnlyFindOneSameNameGameObject);
                }
            }
        }
        #endregion

        #region 结构
        /// <summary>
        /// 组件名字高亮，图标
        /// </summary>
        private struct IconInfoModel
        {
            public IconInfoModel(Type type) : this(type, Color.white, false) { }

            public IconInfoModel(Type type, Color color) : this(type, color, true) { }

            private IconInfoModel(Type type, Color color, bool changeColor)
            {
                this.type = type;
                this.color = color;
```

```
                this.changeColor = changeColor;
            }

            public Type type;
            public bool changeColor;
            public Color color;
        }
        #endregion

        #region 构造函数与析构函数
        static CustomHierarchy()
        {
            //hierarchy 窗口中的每个可见物体的 OnGUI 触发时，就会派发
hierarchyWindowItemOnGUI 事件，然后就可以去执行对应的事件处理器了
            EditorApplication.hierarchyWindowItemOnGUI +=
HierarchyWindowOnGUIHandler;
            //Hierarchy 窗口变化回调（创建对象以及对其进行重命名、重定父级或销毁，
以及加载、卸载、重命名或重新排序已加载的场景）
            EditorApplication.hierarchyChanged += HierarchyChangedHandler;

            colorLabelStyle = new GUIStyle();
            iconStyle = new GUIStyle();
            iconStyle.fixedWidth = iconRectSize - 2;
            iconStyle.fixedHeight = iconRectSize - 2;

            baseColor = GUI.backgroundColor;

            InitConfigAndData();
        }

        ~CustomHierarchy()
        {
            EditorApplication.hierarchyWindowItemOnGUI -=
HierarchyWindowOnGUIHandler;
            EditorApplication.hierarchyChanged -= HierarchyChangedHandler;

            GUI.backgroundColor = baseColor;
        }
        #endregion

        /// <summary>
        /// 初始化配置
        /// </summary>
        public static void InitConfigAndData()
```

```
        {
            //加载配置表
            customConfig =
(CustomHierarchyConfig)AssetDatabase.LoadAssetAtPath<ScriptableObject>(CustomConfigPat
h);
            areaColorDic.Clear();

            if (customConfig != null)
            {
                iconRectSize = customConfig.iconRectSize;
                firstSpacing = customConfig.firstSpacing;
                iconSpacing = customConfig.iconSpacing;
                minIconShowWidth = customConfig.minIconShowWidth;
                minShowWidth = customConfig.minShowWidth;
                colorFontOffsetX = customConfig.fontColorOffsetX;
                colorFontOffsetY = customConfig.fontColorOffsetY;

                CustomHierarchyConfig.AreaColorModel[] blockModels =
customConfig.blockModels;
                for (int i = 0; i < blockModels.Length; i++)
                {
                    if (blockModels[i].targetName != null
&& !blockModels[i].targetName.Equals(""))
                    {
                        areaColorDic.Add(blockModels[i].targetName,
blockModels[i].blockColor);
                    }
                }

                iconInfoList.Clear();
                CustomHierarchyConfig.HighlightColorModel[] highlightModels =
customConfig.highlightModels;
                for (int i = 0; i < highlightModels.Length; i++)
                {
                    if (highlightModels[i].assemblyName != null
&& !highlightModels[i].assemblyName.Equals(""))
                    {
                        Assembly assembly;
                        try
                        {
                            //尝试加载程序集
                            assembly =
Assembly.Load(highlightModels[i].assemblyName);
                        }
```

```
                catch (Exception e)
                {
                    Debug.LogError(e.StackTrace + "\n" + e.Message);
                    continue;
                }

                if (assembly == null)
                {
                    Debug.LogWarning("未找到程序集: " +
highlightModels[i].assemblyName);
                    continue;
                }

                CustomHierarchyConfig.HighlightColorItem[] highlightItems =
highlightModels[i].items;
                for (int j = 0; j < highlightItems.Length; j++)
                {
                    if (highlightItems[j].componentName != null
&& !highlightItems[j].componentName.Equals(""))
                    {
                        //利用反射，从程序集中获取组件类型
                        Type type =
assembly.GetType(highlightModels[i].assemblyName + "." + highlightItems[j].componentName);
                        if (type == null && (type =
assembly.GetType(highlightItems[j].componentName)) == null)
                        {
                            Debug.LogWarning($"程序集
{highlightModels[i].assemblyName}中未包含: {highlightItems[j].componentName}，无法设置
组件颜色高亮");
                            continue;
                        }
                        iconInfoList.Add(new IconInfoModel(type,
highlightItems[j].color));
                    }
                }
            }
        }
        UpdateAreaColorList();
    }
    else
        Debug.LogWarning("您开启了 Hierarchy 增强，但是却没有对应的自定义
配置");
}
```

```
        #region  图标绘制相关
        //根据 index 排序获取对应 Rect 大小
        private static Rect CreateRect(Rect selectionRect, ref int index)
        {
            Rect rect = new Rect(selectionRect);
            rect.x += rect.width - firstSpacing - (iconSpacing * index);
            rect.width = iconRectSize;
            index++;
            return rect;
        }

        //根据类型绘制图标
        private static void DrawComponentIcon(Rect rect, Type type, GUIStyle customStyle =
null)
        {
            //获得 Unity 内置的图标
            Texture icon = EditorGUIUtility.ObjectContent(null, type).image;
            if (icon == null)
            {
                return;
            }
            if (customStyle != null)
            {
                GUI.Label(rect, icon, customStyle);
            }
            else
            {
                Color origin = GUI.color;
                GUI.color = GUI.backgroundColor;
                GUI.Label(rect, icon, iconStyle);
                GUI.color = origin;
            }
        }

        /// <summary>
        ///  根据类型绘制对应图标和颜色
        /// </summary>
        /// <param name="type">组件类型</param>
        /// <param name="selectionRect"></param>
        /// <param name="go">对象</param>
        /// <param name="order"></param>
        /// <returns></returns>
        private static bool Draw(Type type, Rect selectionRect, GameObject go, ref int order)
        {
```

```
            if (!HasComponent(go, type, false))
            {
                return false;
            }
            Rect rect = CreateRect(selectionRect, ref order);
            DrawComponentIcon(rect, type);
            return true;
        }
        #endregion

        #region 区块颜色相关
        /// <summary>
        /// 更新区域颜色列表
        /// </summary>
        private static void UpdateAreaColorList()
        {
            subColorDic.Clear();
            alreadyAdditionBlockName.Clear();
            if (!EnableCustomHierarchyArea)
                return;
            //获取场景中所有的游戏对象，FindObjectsOfType 获取的是无序的，使用
OrderBy 进行排序
            GameObject[] targetObjects = GameObject.FindObjectsOfType<GameObject>()
                .OrderBy(obj => obj.transform.GetSiblingIndex()).ToArray();

            if (targetObjects.Length != 0)
            {
                foreach (GameObject obj in targetObjects)
                {
                    //检测对象的名称是否包含在区块中
                    if (areaColorDic.ContainsKey(obj.name))
                    {
                        //只在 Hierarchy 中查找第一个具有相同名称的对象
                        if (EnableOnlyFindOneSameNameGameObject &&
alreadyAdditionBlockName.Contains(obj.name))
                        {
                            continue;
                        }
                        AddAreaColorChilds(obj, areaColorDic[obj.name]);
                    }
                }
            }
        }
```

```
        /// <summary>
        ///  将对象下的所有子节点都添加进字典中，后续会将所有的子节点都设置为同一颜色
        /// </summary>
        /// <param name="targetObject">目标对象</param>
        /// <param name="color">在 Hierarchy 窗口中显示的颜色</param>
        private static void AddAreaColorChilds(GameObject targetObject, Color color)
        {
            //将对象的实例化 ID 作为 Key 添加到字典中
            subColorDic.Add(targetObject.GetInstanceID(), color);
            if (!alreadyAdditionBlockName.Contains(targetObject.name))
            {
                alreadyAdditionBlockName.Add(targetObject.name);
            }

            //如果区块效果影响的范围涉及子节点
            if (EnableAreaEffectContainChild)
            {
                int childCount = targetObject.transform.childCount;
                if (childCount != 0)
                {
                    for (int i = 0; i < childCount; i++)
                    {

AddAreaColorChilds(targetObject.transform.GetChild(i).gameObject, color);
                    }
                }
            }
        }
        #endregion

        #region  事件处理器
        /// <summary>
        ///  当 Hierarchy 发生更新时，执行的事件处理器
        /// </summary>
        public static void HierarchyChangedHandler()
        {
            if (!EnableCustomHierarchy)
                return;

            //Stopwatch 可以测量一个时间间隔的运行时间，也可以测量多个时间间隔的总运行时间。一般用来测量代码执行所用的时间或者计算性能数据
            Stopwatch stopWatch = new Stopwatch();
            stopWatch.Start();
```

```
            //每次 Hierarchy 发生改变是，我们就必须检查一次是否有新的区块颜色被添加进去
            UpdateAreaColorList();

            stopWatch.Stop();
            long expendTime = stopWatch.ElapsedMilliseconds;
            if (expendTime > 10)
            {
                Debug.LogWarning($"Hierarchy Changed 耗时较长: {stopWatch.ElapsedMilliseconds}ms");
            }
        }

        //绘制自定义的 Hiercrchy
        private static void HierarchyWindowOnGUIHandler(int instanceId, Rect selectionRect)
        {
            if (!EnableCustomHierarchy)
                return;

            Color customFontColor = Color.clear;

            try
            {
                int insID = instanceId;
                //图标排序的索引
                int index = 0;

                //将实例化 id 转换为对象
                GameObject instanceObj = (GameObject)EditorUtility.InstanceIDToObject(instanceId);
                if (instanceObj == null)
                {
                    return;
                }
                float curHalfWidth = EditorGUIUtility.currentViewWidth / 2;

                GUI.backgroundColor = baseColor;
                //显示区块颜色
                if (EnableCustomHierarchyArea)
                {
                    if (subColorDic.ContainsKey(insID))
                    {
                        EditorGUI.DrawRect(selectionRect, subColorDic[insID]);
```

```
            }
        }

        //大于最小宽度时候的显示
        if (minShowWidth < curHalfWidth)
        {
            //Active 勾选框
            bool activeToggle = GUI.Toggle(CreateRect(selectionRect, ref index),
instanceObj.activeSelf, string.Empty);
            if (activeToggle != instanceObj.activeSelf)
            {
                instanceObj.SetActive(activeToggle);
            }

            //静态对象标记显示
            if (instanceObj.isStatic)
            {
                Rect rectIcon = CreateRect(selectionRect, ref index);
                GUI.Label(rectIcon, "S");
            }
        }

        //绘制图标和颜色名称，小于最小图标绘制宽度时只改变名称颜色
        if (curHalfWidth > minIconShowWidth)
        {
            foreach (IconInfoModel model in iconInfoList)
            {
                if (Draw(model.type, selectionRect, instanceObj, ref index))
                {
                    if (model.changeColor)
                    {
                        customFontColor = model.color;
                    }
                }
            }
        }
        else
        {
            foreach (IconInfoModel model in iconInfoList)
            {
                if (HasComponent(instanceObj, model.type, false))
                {
                    if (model.changeColor)
                    {
```

```
                                customFontColor = model.color;
                            }
                        }
                    }
                }

                Rect targetRect = selectionRect;
                targetRect.x += colorFontOffsetX;
                targetRect.y += colorFontOffsetY;

                //开启了增强显示，且在 Hierarchy 窗口中是处于激活状态，就可以执行自定义颜色
                if (EnableCustomHierarchyColor && (customFontColor != Color.clear) && instanceObj.activeInHierarchy)
                {
                    Color originColor = colorLabelStyle.normal.textColor;
                    colorLabelStyle.normal.textColor = customFontColor;
                    GUI.Label(targetRect, instanceObj.name, colorLabelStyle);
                    colorLabelStyle.normal.textColor = originColor;
                }
                else if (EnableCustomHierarchyAreaName && subColorDic.ContainsKey(insID) && instanceObj.activeInHierarchy)
                {
                    Color originColor = colorLabelStyle.normal.textColor;
                    //利用色相设置区块字体的颜色显示
                    Color.RGBToHSV(subColorDic[insID], out H, out S, out V);
                    H -= 0.43f;
                    if (H < 0)
                        H = 1 + H;
                    S = 1;
                    V = 1;
                    colorLabelStyle.normal.textColor = Color.HSVToRGB(H,S,V);
                    GUI.Label(targetRect, instanceObj.name, colorLabelStyle);
                    colorLabelStyle.normal.textColor = originColor;
                }
            }
            catch (Exception e)
            {
                Debug.LogError(e.Message + "\n" + HighlightStackTrace(e.StackTrace));
            }
        }
        #endregion

        #region DebugStack
```

```
        /// <summary>
        ///  调用栈追踪文件名高亮
        /// </summary>
        /// <param name="stackTrace"></param>
        /// <returns></returns>
        public static string HighlightStackTrace(string stackTrace)
        {
            if (stackTrace == null || stackTrace.Length == 0)
            {
                return stackTrace;
            }
            string[] splited = stackTrace.Split('\n');
            StringBuilder strBuilder = new StringBuilder();
            foreach (string str in splited)
            {
                string[] temp = str.Split('\\');
                if (temp.Length > 0)
                {
                    strBuilder.Append(str.Replace(temp[temp.Length - 1],
string.Format("<color=yellow>{0}</color>\n", temp[temp.Length - 1])));
                }
                else
                {
                    strBuilder.Append(str);
                }
            }

            return strBuilder.ToString();
        }
        #endregion

        #region 检测对象中是否含有组件
        /// <summary>
        ///  检测对象上是否含有对应组件
        /// </summary>
        /// <typeparam name="T">泛型组件类型</typeparam>
        /// <param name="go">检测的对象</param>
        /// <param name="checkChildren">是否检测子层级</param>
        /// <returns>是否含有期望组件</returns>
        public static bool HasComponent<T>(GameObject go, bool checkChildren) where T :
Component
        {
            if (!checkChildren)
            {
```

```
                return go.GetComponent<T>();
            }
            else
            {
                return go.GetComponentsInChildren<T>().FirstOrDefault() != null;
            }
        }

        /// <summary>
        ///  检测对象上是否含有对应组件
        /// </summary>
        /// <param name="go">检测的对象</param>
        /// <param name="type">检查的组件类型</param>
        /// <param name="checkChildren">是否检测子层级</param>
        /// <returns>是否含有期望组件</returns>
        public static bool HasComponent(GameObject go, Type type, bool checkChildren)
        {
            if (!checkChildren)
            {
                return go.GetComponent(type) != null;
            }
            else
            {
                return go.GetComponentsInChildren(type).FirstOrDefault() != null;
            }
        }
        #endregion
    }

    /// <summary>
    ///  自定义 Hierarchy 增强显示开关
    /// </summary>
    public class EnableCustomHierarchy : EditorWindow
    {
        private const string ToggleTitle = "游戏工具/GUI 增强/Hierarchy 自定义/开启
Hierarchy 自定义增强显示";
        private const string ColorToggleTitle = "游戏工具/GUI 增强/Hierarchy 自定义/开启
Hierarchy 组件颜色高亮";
        private const string AreaToggleTitle = "游戏工具/GUI 增强/Hierarchy 自定义/开启
Hierarchy 颜色区块";
        private const string AreaNameToggleTitle = "游戏工具/GUI 增强/Hierarchy 自定义/开
启 Hierarchy 颜色区块名称高亮";
        private const string AreaEffectContainChild = "游戏工具/GUI 增强/Hierarchy 自定义
/Hierarchy 颜色区块同时影响其子节点";
```

```
        private const string OnlyFindOneSameNameGameObject = "游戏工具/GUI 增强
/Hierarchy 自定义/Hierarchy 颜色区块只影响第一个同名的对象";
        private const string InfoTitle = "游戏工具/GUI 增强/Hierarchy 自定义/说明";

        #region MenuItem
        //是否开启自定义 Hierarchy 增强视图
        [MenuItem(ToggleTitle, false, 100)]
        public static void IconToggle()
        {
            CustomHierarchy.EnableCustomHierarchy
= !CustomHierarchy.EnableCustomHierarchy;
            if (CustomHierarchy.EnableCustomHierarchy)
                CustomHierarchy.InitConfigAndData();
        }
        [MenuItem(ToggleTitle, true)]
        public static bool SetIconToggle()
        {
            Menu.SetChecked(ToggleTitle, CustomHierarchy.EnableCustomHierarchy);
            return true;
        }

        //自定义 Hierarchy 颜色高亮开关
        [MenuItem(ColorToggleTitle, false, 101)]
        public static void ColorToggle()
        {
            CustomHierarchy.EnableCustomHierarchyColor
= !CustomHierarchy.EnableCustomHierarchyColor;
            CustomHierarchy.HierarchyChangedHandler();
        }
        [MenuItem(ColorToggleTitle, true, 101)]
        public static bool SetColorToggle()
        {
            Menu.SetChecked(ColorToggleTitle,
CustomHierarchy.EnableCustomHierarchyColor);
            return true;
        }

        //自定义 Hierarchy 颜色区块开关
        [MenuItem(AreaToggleTitle, false, 102)]
        public static void AreaToggle()
        {
            CustomHierarchy.EnableCustomHierarchyArea
= !CustomHierarchy.EnableCustomHierarchyArea;
            CustomHierarchy.HierarchyChangedHandler();
```

```
        }
        [MenuItem(AreaToggleTitle, true, 102)]
        public static bool SetAreaToggle()
        {
            Menu.SetChecked(AreaToggleTitle,
CustomHierarchy.EnableCustomHierarchyArea);
            return true;
        }

        //自定义颜色区块名称高亮开关
        [MenuItem(AreaNameToggleTitle, false, 103)]
        public static void AreaNameToggle()
        {
            CustomHierarchy.EnableCustomHierarchyAreaName
= !CustomHierarchy.EnableCustomHierarchyAreaName;
            CustomHierarchy.HierarchyChangedHandler();
        }
        [MenuItem(AreaNameToggleTitle, true, 103)]
        public static bool SetAreaNameToggle()
        {
            Menu.SetChecked(AreaNameToggleTitle,
CustomHierarchy.EnableCustomHierarchyAreaName);
            return true;
        }

        //颜色区块同时影响其子节点开关
        [MenuItem(AreaEffectContainChild, false, 104)]
        public static void AreaEffectContainChildToggle()
        {
            CustomHierarchy.EnableAreaEffectContainChild
= !CustomHierarchy.EnableAreaEffectContainChild;
            CustomHierarchy.HierarchyChangedHandler();
        }
        [MenuItem(AreaEffectContainChild, true, 104)]
        public static bool SetAreaEffectContainChildToggle()
        {
            Menu.SetChecked(AreaEffectContainChild,
CustomHierarchy.EnableAreaEffectContainChild);
            return true;
        }

        //是否只对查找场景中的第一个同名对象应用效果
        [MenuItem(OnlyFindOneSameNameGameObject, false, 105)]
        public static void OnlyFindOneToggle()
```

```
        {
            CustomHierarchy.EnableOnlyFindOneSameNameGameObject
= !CustomHierarchy.EnableOnlyFindOneSameNameGameObject;
            CustomHierarchy.HierarchyChangedHandler();
        }
        [MenuItem(OnlyFindOneSameNameGameObject, true, 105)]
        public static bool SetOnlyFindOneToggle()
        {
            Menu.SetChecked(OnlyFindOneSameNameGameObject,
CustomHierarchy.EnableOnlyFindOneSameNameGameObject);
            return true;
        }

        //说明面板
        [MenuItem(InfoTitle, false, 106)]
        public static void InfoPanel()
        {
            GetWindow<CustomHierarchyInfo>("自定义 Hierarchy 说明");
        }
        #endregion
    }

    #region 自定义 Hierarchy 说明面板
    /// <summary>
    /// 自定义 Hierarchy 说明面板
    /// </summary>
    class CustomHierarchyInfo : EditorWindow
    {
        private const string info = @"
        >该工具可以在 Hierarchy 窗口显示物体上所带有的组件图标以及该物体是否为
static\active 等状态信息，
        且可以根据物体所带有的组件或其名称来改变物体名称的颜色等。

        >如果想显示更多信息，可在 CustomHierarchy.cs 的 HierarchyWindowOnGUI 方法
中添加。

        >组件图标显示和组件颜色高亮可在
<color=#E5E37F>Scripts/Game/Editor/GUI/CustomHierarchyConfig.asset</color>中配置
        例如希望显示 Canvas 的图标并使带有 Canvas 组件的物体名显示为某种颜色，则在
Highlight Models 下程序集为
        UnityEngine 的 Items 中填入 Canvas 并设置对应颜色即可

        >颜色区块可以使某一名称的物体包括其子物体显示为某一颜色
        具体可以在
```

<color=#E5E37F>Scripts/Game/Editor/GUI/CustomHierarchyConfig.asset</color>中配置

例如希望 MainLayer 及其子物体显示为黄色，则在 Models 下填入 MainLayer 然后选中对应颜色即可。

物体名称的颜色变化优先级低于上面组件的颜色，名称高亮可在设置中关闭";

```
        private static GUIStyle style;

        private void Awake()
        {
            style = new GUIStyle("label");
            style.fontSize = 16;
            style.richText = true;
            minSize = new Vector2(45 * style.fontSize, 50);
        }

        private void OnGUI()
        {
            GUILayout.Label(info, style);
        }
    }
    #endregion
```

六. BuildSetting 自动化添加场景

再 Unity3d 开发过程中，如果添加了新的场景，不添加进 BuildingSetting 是无法进入的，而每次手动导入进去会很麻烦。

其次，在正式发布版本中，会使用 Assetbundle 来加载此 Scene，又要将除开第一个场景外的场景移除。所以写了一个脚本来自动化实现。

PS：

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

public class BuildSettingScenesTool
{
    /// <summary>
    ///  自动化操作的场景路径
    /// </summary>
    public static readonly string _SceneDir = Application.dataPath + "/Scenes";
    public const string _AssetSceneDir = "Assets/Scenes";
    public const string _FirstSceneName = "SampleScene.unity";
```

```
        #region MenuItem
        [MenuItem("游戏工具/BuildSettings/更新 BuildSettings 中的 Scene", false,1)]
        public static void AutoAddScenesToBuildForEditor()
        {
            AutoAddScenesToBuild(false);
        }

        //MenuItem 第二个参数是 true 则是给该菜单项添加一个验证，需分别标记两个函数，true 标记的函数作为 false 标记的函数能否启用并执行的验证
        [MenuItem("游戏工具/BuildSettings/更新 BuildSettings 中的 Scene", true,1)]
        public static bool CheckAutoAddScenesToBuildForEditor()
        {
            return !EditorApplication.isPlaying;
        }

        [MenuItem("游戏工具/BuildSettings/更新 BuildSettings 中的 Scene 至打包状态", false, 2)]
        public static void AutoAddScenesToBuildForPackage()
        {
            AutoAddScenesToBuild(true);
        }

        [MenuItem("游戏工具/BuildSettings/更新 BuildSettings 中的 Scene 至打包状态", true, 2)]
        public static bool CheckAutoAddScenesToBuildForPackage()
        {
            return !EditorApplication.isPlaying;
        }

        [MenuItem("游戏工具/BuildSettings/清空 BuildSettings 中的所有场景", false, 3)]
        public static void RemoveScenesToBuildForPackage()
        {
            RemoveAllScenesToBuild();
        }

        [MenuItem("游戏工具/BuildSettings/清空 BuildSettings 中的所有场景", true, 3)]
        public static bool CheckRemoveScenesToBuildForPackage()
        {
            return !EditorApplication.isPlaying;
        }
        #endregion

        /// <summary>
        ///  将场景添加到 BuildSetting 中
```

```
        /// </summary>
        /// <param name="isPackage">是否是打包状态</param>
        public static void AutoAddScenesToBuild(bool isPackage)
        {
            //匹配目录下的所有.unity 文件
            string pattern = "*.unity";
            string[] files = FileUtility.GetFilesPaths(_SceneDir, pattern);
            EditorBuildSettingsScene[] buildSettingScene = new
EditorBuildSettingsScene[files.Length];
            for (int i = 0; i < files.Length; i++)
            {
                string name = FileUtility.GetFileName(files[i], false);
                buildSettingScene[i] = GetBuidSettingScene(name, isPackage);
            }

            //设置场景
            EditorBuildSettings.scenes = buildSettingScene;
            Debug.Log("加载 BuidSettingScene 完成");
        }

        /// <summary>
        ///  清空 BuildSetting 中的所有场景
        /// </summary>
        public static void RemoveAllScenesToBuild()
        {
            EditorBuildSettings.scenes = null;
        }

        /// <summary>
        ///  获取 BuildSettingScene
        /// </summary>
        /// <param name="sceneName">场景名称</param>
        /// <param name="isPackage">是否是打包状态</param>
        /// <returns></returns>
        public static EditorBuildSettingsScene GetBuidSettingScene(string sceneName, bool
isPackage)
        {
            //是否在  Build Settings  窗口中启用此场景
            bool enableScene = true;
            if (isPackage)
            {
                enableScene = (sceneName == _FirstSceneName);
            }
```

```
            string sceneAssetPath = _AssetSceneDir + "/" + sceneName;
            //EditorBuildSettingsScene 的资源路径，必须包含扩展名
            return new EditorBuildSettingsScene(sceneAssetPath, enableScene);
        }
    }
```

七. 双击按钮与长按按钮

1. 双击按钮

PS：

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;
using UnityEngine.UI;
using UnityEngine.Events;
using UnityEngine.EventSystems;

//双击按钮
[AddComponentMenu("UI/ExpendButton/DubleClickButton")]
public class DoubleClickButton : Button
{
    [Serializable]
    public class DoubleClickedEvent : UnityEvent { }

    [SerializeField]
    private DoubleClickedEvent onDoubleClickEvent = new DoubleClickedEvent();

    //这个是双击成功后激活的事件
    public DoubleClickedEvent OnDoubleClick
    {
        get { return onDoubleClickEvent; }
        set { onDoubleClickEvent = value; }
    }

    private DateTime firstClickTime;
    private DateTime secondClickTime;

    /// <summary>
    ///  执行 DoubleClick
    /// </summary>
    private void DoDoubleClick()
    {
        if (OnDoubleClick != null)
            OnDoubleClick.Invoke();
        resetTime();
```

```
        }

        public override void OnPointerDown(PointerEventData eventData)
        {
            base.OnPointerDown(eventData);
            // 按下按钮时对两次的时间进行记录
            if (firstClickTime.Equals(default(DateTime)))
                firstClickTime = DateTime.Now;
            else
                secondClickTime = DateTime.Now;
        }

        public override void OnPointerUp(PointerEventData eventData)
        {
            base.OnPointerUp(eventData);

            // 在第二次鼠标抬起的时候进行时间的触发,时差小于 400ms 触发
            if (!firstClickTime.Equals(default(DateTime))
&& !secondClickTime.Equals(default(DateTime)))
            {
                TimeSpan intervalTime = secondClickTime - firstClickTime;
                //float milliSeconds = intervalTime.Seconds * 1000 +
intervalTime.Milliseconds; 1s=1000ms

                //总毫秒数是否小于 400 毫秒
                if (intervalTime.TotalMilliseconds < 400)
                    DoDoubleClick();
                else
                    resetTime();
            }
        }

        public override void OnPointerExit(PointerEventData eventData)
        {
            base.OnPointerExit(eventData);
            resetTime();
        }

        /// <summary>
        /// 重置计时
        /// </summary>
        private void resetTime()
        {
            firstClickTime = default(DateTime);
```

```
            secondClickTime = default(DateTime);
        }
    }
```

2. 长按按钮

PS：

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;
using UnityEngine.UI;
using UnityEngine.Events;
using UnityEngine.EventSystems;

//长按按钮
[AddComponentMenu("UI/ExpendButton/LongClickButton")]
public class LongClickButton : Button
{
    /// <summary>
    /// 长按时间（ms）
    /// </summary>
    public int longPressTime = 400;

    [Serializable]
    public class LongClickEvent : UnityEvent {}

    private LongClickEvent onLongClickUpEvent = new LongClickEvent();
    private LongClickEvent onLongPressEvent = new LongClickEvent();

    /// <summary>
    /// 倒计时协程
    /// </summary>
    private Coroutine countDownCoroutine = null;

    /// <summary>
    /// 长按后抬起时响应。（使用属性封装一次，控制外部对内部私有字段的访问）
    /// </summary>
    public LongClickEvent OnLongClickUp
    {
        get { return onLongClickUpEvent; }
        set { onLongClickUpEvent = value; }
    }

    /// <summary>
    /// 长按事件
```

```
/// </summary>
public LongClickEvent OnLongPressEvent
{
    get { return onLongPressEvent; }
    set { onLongPressEvent = value; }
}

private DateTime firstClickTime = default(DateTime);
private DateTime firstClickTime_Up = default(DateTime);

/// <summary>
/// 长按抬起的事件处理器
/// </summary>
private void LongClickUpHandler()
{
    if (OnLongClickUp != null)
        OnLongClickUp.Invoke();
    resetTime();
}

/// <summary>
/// 长按事件处理器
/// </summary>
private void LongPressHandler()
{
    if (OnLongPressEvent != null)
        OnLongPressEvent.Invoke();
    if (countDownCoroutine != null)
        StopCoroutine(countDownCoroutine);
}

/// <summary>
/// 鼠标按下时执行
/// </summary>
/// <param name="eventData"></param>
public override void OnPointerDown(PointerEventData eventData)
{
    base.OnPointerDown(eventData);
    if (firstClickTime.Equals(default(DateTime)))
    {
        firstClickTime = DateTime.Now;
        countDownCoroutine = StartCoroutine(CountDown());
    }
}
```

```
        /// <summary>
        ///  鼠标抬起时执行
        /// </summary>
        /// <param name="eventData"></param>
        public override void OnPointerUp(PointerEventData eventData)
        {
            base.OnPointerUp(eventData);
            // 在鼠标抬起的时候进行事件触发，时差大于 600ms 触发
            if (!firstClickTime.Equals(default(DateTime)))
                firstClickTime_Up = DateTime.Now;
            if (!firstClickTime.Equals(default(DateTime))
&& !firstClickTime_Up.Equals(default(DateTime)))
            {
                //计算两次时间戳的间隔时间（毫秒）
                TimeSpan intervalTime = firstClickTime_Up - firstClickTime;
                if (intervalTime.TotalMilliseconds > longPressTime)
                    LongClickUpHandler();
                else
                    resetTime();
            }
        }

        /// <summary>
        ///  鼠标移出
        /// </summary>
        /// <param name="eventData"></param>
        public override void OnPointerExit(PointerEventData eventData)
        {
            base.OnPointerExit(eventData);
            resetTime();
        }

        /// <summary>
        ///  重置计时
        /// </summary>
        private void resetTime()
        {
            firstClickTime = default(DateTime);
            firstClickTime_Up = default(DateTime);
        }

        /// <summary>
        ///  协程倒计时
```

```
        /// </summary>
        /// <returns></returns>
        private IEnumerator CountDown()
        {
            TimeSpan temporary = firstClickTime - DateTime.Now;
            while (temporary.TotalMilliseconds < longPressTime)
            {
                temporary = DateTime.Now - firstClickTime;
                //yield return 后面的返回值永远只有一帧，不论是 yield return 1 还是 yield return 100
                yield return null;
            }
            LongPressHandler();
        }
    }
```

3. 自定义长按按钮 Inspector 界面

我们在前面的长按按钮中公开了一个变量，但是这一变量是无法直接显示的，我们需要重新自定义该组件的 Inspector 界面。

PS：

```
using UnityEditor;
using UnityEditor.UI;

[CustomEditor(typeof(LongClickButton))]
public class LongClickButtonEditor : ButtonEditor
{
    private LongClickButton theTarget;

    public void Awake()
    {
        theTarget = (LongClickButton)this.target;
    }

    public override void OnInspectorGUI()
    {
        int temporary = EditorGUILayout.IntField("设置长按时间（ms）", theTarget.longPressTime);
        if (temporary != theTarget.longPressTime)
            theTarget.longPressTime = temporary;
        base.OnInspectorGUI();
    }
}
```

4. ExtralEditorUtility

1) 创建 EventSystem

2) 完整代码

PS：

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEditor;
using UnityEngine.UI;
using UnityEngine.EventSystems;

namespace GameUtil
{
    public static class ExtralEditorUtility
    {
        /// <summary>
        /// UI 对象的 Layer 层级名称
        /// </summary>
        private const string UI_LAYER_NAME = "UI";

        /// <summary>
        ///  设置 UI 元素的父节点
        /// </summary>
        /// <param name="element"></param>
        /// <param name="menuCommand">执行该菜单命令</param>
        public static void SetUIElementRoot(GameObject element, MenuCommand
menuCommand)
        {
            GameObject parent = menuCommand.context as GameObject;
            //父节点为空或者父节点（包括爷爷辈）无 Canvas
            if (parent == null || parent.GetComponentInParent<Canvas>() == null)
                parent = GetOrCreateCanvasGameObject();
            //确保新的子游戏对象与层级视图中的同级相比具有唯一的名称
            string uniqueName =
GameObjectUtility.GetUniqueNameForSibling(parent.transform, element.name);
            element.name = uniqueName;

            Undo.RegisterCreatedObjectUndo(element, "Create " + element.name);
            Undo.SetTransformParent(element.transform, parent.transform, "Parent " +
element.name);

            GameObjectUtility.SetParentAndAlign(element, parent);
            //判断父节点是否是执行该菜单命令的对象，不是就将其在 SceneView 中
```

```
居中
                if (parent != menuCommand.context)
                    SetPositionInSceneView(parent.GetComponent<RectTransform>(),
element.GetComponent<RectTransform>());
                Selection.activeGameObject = element;
            }

            /// <summary>
            ///  设置在场景视图中的位置
            /// </summary>
            /// <param name="canvasRTransform"></param>
            /// <param name="itemTransform"></param>
            public static void SetPositionInSceneView(RectTransform canvasRTransform,
RectTransform itemTransform)
            {
                //获得最近激活的场景
                SceneView sceneView = SceneView.lastActiveSceneView;
                if (sceneView == null && SceneView.sceneViews.Count > 0)
                    sceneView = SceneView.sceneViews[0] as SceneView;

                //  找不到场景视图就不设置位置
                if (sceneView == null || sceneView.camera == null)
                    return;

                //  在画布位置创建位于世界空间的 Plane
                Vector2 localPlanePosition;
                Camera camera = sceneView.camera;
                Vector3 position = Vector3.zero;
                //将一个屏幕空间点转换为 RectTransform 的本地空间，
Vector2(camera.pixelWidth / 2, camera.pixelHeight / 2)是计算当前分辨率下的屏幕中心点
                if
(RectTransformUtility.ScreenPointToLocalPointInRectangle(canvasRTransform, new
Vector2(camera.pixelWidth / 2, camera.pixelHeight / 2), camera, out localPlanePosition))
                {
                    //调整画布的 pivot
                    localPlanePosition.x = localPlanePosition.x +
canvasRTransform.sizeDelta.x * canvasRTransform.pivot.x;
                    localPlanePosition.y = localPlanePosition.y +
canvasRTransform.sizeDelta.y * canvasRTransform.pivot.y;

                    localPlanePosition.x = Mathf.Clamp(localPlanePosition.x, 0,
canvasRTransform.sizeDelta.x);
                    localPlanePosition.y = Mathf.Clamp(localPlanePosition.y, 0,
canvasRTransform.sizeDelta.y);
```

```
                    //调整画布的 anchor
                    position.x = localPlanePosition.x - canvasRTransform.sizeDelta.x *
itemTransform.anchorMin.x;
                    position.y = localPlanePosition.y - canvasRTransform.sizeDelta.y *
itemTransform.anchorMin.y;

                    Vector3 minLocalPosition;
                    minLocalPosition.x = canvasRTransform.sizeDelta.x * (0 -
canvasRTransform.pivot.x) + itemTransform.sizeDelta.x * itemTransform.pivot.x;
                    minLocalPosition.y = canvasRTransform.sizeDelta.y * (0 -
canvasRTransform.pivot.y) + itemTransform.sizeDelta.y * itemTransform.pivot.y;

                    Vector3 maxLocalPosition;
                    maxLocalPosition.x = canvasRTransform.sizeDelta.x * (1 -
canvasRTransform.pivot.x) - itemTransform.sizeDelta.x * itemTransform.pivot.x;
                    maxLocalPosition.y = canvasRTransform.sizeDelta.y * (1 -
canvasRTransform.pivot.y) - itemTransform.sizeDelta.y * itemTransform.pivot.y;

                    position.x = Mathf.Clamp(position.x, minLocalPosition.x,
maxLocalPosition.x);
                    position.y = Mathf.Clamp(position.y, minLocalPosition.y,
maxLocalPosition.y);
                }

                itemTransform.anchoredPosition = position;
                itemTransform.localRotation = Quaternion.identity;
                itemTransform.localScale = Vector3.one;
            }

            /// <summary>
            ///  获取或者创建 Canvas（有 Canvas 则是获取，无就是创建）
            /// </summary>
            /// <param name="globalFind">当前组件及其父物体上没有 Canvas 时，是否进
行全局查找</param>
            /// <returns></returns>
            public static GameObject GetOrCreateCanvasGameObject(bool globalFind = true)
            {
                GameObject selectedGo = Selection.activeGameObject;
                Canvas canvas = (selectedGo != null) ?
selectedGo.GetComponentInParent<Canvas>() : null;
                if (canvas != null && canvas.gameObject.activeInHierarchy)
                    return canvas.gameObject;
                //全局查找一次
```

```
            if (globalFind)
            {
                canvas = Object.FindObjectOfType(typeof(Canvas)) as Canvas;
                if (canvas != null && canvas.gameObject.activeInHierarchy)
                    return canvas.gameObject;
            }
            return CreateNewUI();
        }

        /// <summary>
        ///  创建新的 UI 画布
        /// </summary>
        /// <returns></returns>
        public static GameObject CreateNewUI()
        {
            GameObject canvas = new GameObject();
            //确保新的子  GameObject  与其在层次结构中的兄弟姐妹相比具有唯一
的名称。
            canvas.name = GameObjectUtility.GetUniqueNameForSibling(null,
"Canvas");
            canvas.layer = LayerMask.NameToLayer(UI_LAYER_NAME);
            canvas.AddComponent<Canvas>();
            canvas.GetComponent<Canvas>().renderMode =
RenderMode.ScreenSpaceOverlay;
            canvas.AddComponent<CanvasScaler>();
            canvas.AddComponent<GraphicRaycaster>();
            //创建新对象时需要将其注册到撤销堆栈中
            Undo.RegisterCreatedObjectUndo(canvas, "Create" + canvas.name);
            CreateEventSystem();
            return canvas;
        }

        /// <summary>
        ///  创建一个 EventSystem
        /// </summary>
        /// <param name="select">是否选中对象</param>
        /// <param name="parent">父节点</param>
        public static void CreateEventSystem(bool select = false, GameObject parent =
null)
        {
            //全局查找是否有 EventSystem 存在
            EventSystem globalEventSystem =
Object.FindObjectOfType<EventSystem>();
            if (globalEventSystem == null)
```

```
                {
                    GameObject eventSystem = new GameObject("EventSystem");
                    //将子对象设置到父节点下，并确保子对象的 Layer 个 Position 是一
致的
                    GameObjectUtility.SetParentAndAlign(eventSystem, parent);
                    globalEventSystem = eventSystem.AddComponent<EventSystem>();
                    eventSystem.AddComponent<StandaloneInputModule>();

                    Undo.RegisterCreatedObjectUndo(eventSystem, "Create " +
eventSystem.name);
                }

                if (select && globalEventSystem != null)
                {
                    //设置选中对象为当前创建的对象
                    Selection.activeGameObject = globalEventSystem.gameObject;
                }
            }
        }
    }
```

5. 额外组件控制

PS：

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEditor;
using GameUtil;

public class ExtralCustomComponentControl
{
    #region 创建组件
    /// <summary>
    /// 创建双击组件
    /// </summary>
    public static GameObject CreateDoubleClickButton()
    {
        GameObject dualClickButton = DefaultControls.CreateButton(new
DefaultControls.Resources());
        dualClickButton.name = "DoubleClickButton";
        dualClickButton.transform.Find("Text").GetComponent<Text>().text =
"DoubleClickButton";
        Object.DestroyImmediate(dualClickButton.GetComponent<Button>());
        dualClickButton.AddComponent<DoubleClickButton>();
```

```
            return dualClickButton;
        }

        /// <summary>
        ///  创建长按组件
        /// </summary>
        public static GameObject CreateLongClickButton()
        {
            GameObject longPressButton = DefaultControls.CreateButton(new
DefaultControls.Resources());
            longPressButton.name = "LongClickButton";
            longPressButton.transform.Find("Text").GetComponent<Text>().text =
"LongClickButton";
            Object.DestroyImmediate(longPressButton.GetComponent<Button>());
            longPressButton.AddComponent<LongClickButton>();
            return longPressButton;
        }
        #endregion

        #region  添加组件
        [MenuItem("GameObject/UI/Buttons/DoubleClickButton", false, 1)]
        public static void AddDoubleClickButton(MenuCommand menuCommand)
        {
            GameObject theButton = CreateDoubleClickButton();
            ExtralEditorUtility.SetUIElementRoot(theButton, menuCommand);
        }


        [MenuItem("GameObject/UI/Buttons/LongClickButton", false, 2)]
        public static void AddLongClickButton(MenuCommand menuCommand)
        {
            GameObject theButton = CreateLongClickButton();
            ExtralEditorUtility.SetUIElementRoot(theButton,menuCommand);
        }
        #endregion
    }
```

八. 自定义扩展 Transform 组件

简单讲解一下 TransformPlus 这个 Editor 脚本。

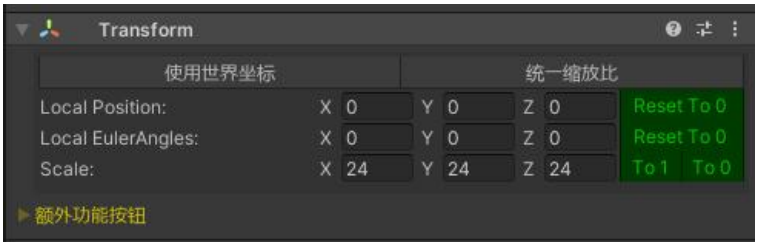

图 10.　TransformPlus 效果图

最开始的位置是所需要的各种变量，这些变量的值不同，在 Inspector 面板上显示的内容也将会发生变化。

PS：

```
/// <summary>
/// 显示的是否是世界坐标
/// </summary>
private static bool useWorldSystem = false;
/// <summary>
/// 是否使用统一的缩放比
/// </summary>
private bool uniformScale = true;

private Transform theTarget;
//是否显示额外功能按钮
private static bool showExtraButton = false;
```

接着在 OnInspectorGUI 函数中，先绘制出上方的显示调整按钮。然后根据调整按钮此刻所处的状态，去分别设置局部坐标和世界坐标情况下的布局。同时，为了能够快速归零位置和角度，在其最右侧新增一个归零按钮。

对于 Scale，因为其三个分量大多数情况下都是一致的。我们就让其在分量值一致的情况下，只显示一个公共值，同时也可以手动切换回原本的三分量的显示模式。其最右侧依旧和前面的位置和旋转一样，我们为其添加一些常用的按钮，我这里写的就是值归零和归一两个按钮。

最后，我们绘制最下方的扩展按钮组。通过扩展按钮，我们就可以非常快速的复制当前对象的位置、角度、路径等。

1) 完整代码

PS：

```
using UnityEngine;
using UnityEditor;
using GameUtil;

//CustomEditor 特性表明这是一个自定义的 Editor 脚本
[CustomEditor(typeof(Transform))]
public class TransformInspectorPlus : Editor
{
    /// <summary>
    /// 显示的是否是世界坐标
    /// </summary>
    private static bool useWorldSystem = false;
    /// <summary>
    /// 是否使用统一的缩放比
    /// </summary>
    private bool uniformScale = true;
```

```
        private Transform theTarget;
        //是否显示额外功能按钮
        private static bool showExtraButton = false;

        private void Awake()
        {
            theTarget = (Transform)this.target;
            uniformScale = theTarget.localScale.x == theTarget.localScale.y &&
theTarget.localScale.y == theTarget.localScale.z;
        }

        public override void OnInspectorGUI()
        {
            GUILayout.BeginVertical("box");
            {
                #region 上方的功能选项
                GUILayout.BeginHorizontal("Toolbar");
                {
                    useWorldSystem = GUILayout.Toggle(useWorldSystem, "使用世界坐标
", "toolbarbutton");
                    uniformScale = GUILayout.Toggle(uniformScale, "统一缩放比",
"toolbarbutton");
                }
                GUILayout.EndHorizontal();
                #endregion

                #region 根据使用的坐标系绘制位置和旋转值
                Vector3 temporaryVector;
                if (useWorldSystem)
                {
                    #region 世界坐标情况
                    GUILayout.BeginHorizontal();
                    {
                        temporaryVector = EditorGUILayout.Vector3Field("Position:",
theTarget.position);
                        if (temporaryVector != theTarget.position)
                        {
                            //将修改加入 Undo 堆栈
                            Undo.RecordObject(theTarget, "Modify Position");
                            theTarget.position = temporaryVector;
                        }

                        GUI.color = Color.green;
                        if (GUILayout.Button("Reset To 0", "toolbarbutton",
```

```
GUILayout.MaxWidth(80)))
                    {
                        Undo.RecordObject(theTarget, "Modify Position");
                        theTarget.position = Vector3.zero;
                    }
                    GUI.color = Color.white;
                }
                GUILayout.EndHorizontal();

                GUILayout.BeginHorizontal();
                {
                    temporaryVector = EditorGUILayout.Vector3Field("EulerAngles:",
theTarget.eulerAngles);
                    if (temporaryVector != theTarget.eulerAngles)
                    {
                        Undo.RecordObject(target, "Modify Rotation");
                        theTarget.eulerAngles = temporaryVector;
                    }

                    GUI.color = Color.green;
                    if (GUILayout.Button("Reset To 0", "toolbarbutton",
GUILayout.MaxWidth(80)))
                    {
                        Undo.RecordObject(theTarget, "Modify Rotation");
                        theTarget.rotation = Quaternion.identity;
                    }
                    GUI.color = Color.white;
                }
                GUILayout.EndHorizontal();
                #endregion
            }
            else
            {
                #region 局部坐标情况
                GUILayout.BeginHorizontal();
                {
                    temporaryVector = EditorGUILayout.Vector3Field("Local Position:",
theTarget.localPosition);
                    if (temporaryVector != theTarget.localPosition)
                    {
                        Undo.RecordObject(target, "Modify Position");
                        theTarget.localPosition = temporaryVector;
                    }
```

```
                GUI.color = Color.green;
                if (GUILayout.Button("Reset To 0", "toolbarbutton",
GUILayout.MaxWidth(80)))
                {
                    Undo.RecordObject(theTarget, "Modify Position");
                    theTarget.localPosition = Vector3.zero;
                }
                GUI.color = Color.white;
            }
            GUILayout.EndHorizontal();

            GUILayout.BeginHorizontal();
            {
                temporaryVector = EditorGUILayout.Vector3Field("Local
EulerAngles:", theTarget.localEulerAngles);
                if (temporaryVector != theTarget.localEulerAngles)
                {
                    Undo.RecordObject(theTarget, "Modify Rotation");
                    theTarget.localEulerAngles = temporaryVector;
                }

                GUI.color = Color.green;
                if (GUILayout.Button("Reset To 0", "toolbarbutton",
GUILayout.MaxWidth(80)))
                {
                    Undo.RecordObject(theTarget, "Modify Rotation");
                    theTarget.localRotation = Quaternion.identity;
                }
                GUI.color = Color.white;
            }
            GUILayout.EndHorizontal();
            #endregion
        }
        #endregion

        #region 绘制自定义 Scale
        GUILayout.BeginHorizontal();
        {
            if (uniformScale)
            {
                //统一缩放比时，就以 x 分量为基准设置其他分量
                float value = EditorGUILayout.FloatField("Scale:",
theTarget.localScale.x);
                if (theTarget.localScale.x != value)
```

```
                    {
                        Undo.RecordObject(theTarget, "ScaleIsUniform");
                        theTarget.localScale = new Vector3(value, value, value);
                    }
                }
                else
                {
                    temporaryVector = EditorGUILayout.Vector3Field("Scale:",
theTarget.localScale);
                    if (temporaryVector != theTarget.localScale)
                    {
                        Undo.RecordObject(theTarget, "ScaleIsUniform");
                        theTarget.localScale = temporaryVector;
                    }
                }
                #region 绘制 Scale 右侧的按钮
                GUI.color = Color.green;
                if (GUILayout.Button("To 1", "toolbarbutton",
GUILayout.MaxWidth(40)))
                {
                    Undo.RecordObject(theTarget, "scaleIsUniform to one");
                    theTarget.localScale = Vector3.one;
                }
                if (GUILayout.Button("To 0", "toolbarbutton",
GUILayout.MaxWidth(40)))
                {
                    Undo.RecordObject(theTarget, "scaleIsUniform to zero");
                    theTarget.localScale = Vector3.zero;
                }
                GUI.color = Color.white;
                #endregion
            }
            GUILayout.EndHorizontal();
            #endregion

            GUILayout.Space(5);

        }
        GUILayout.EndVertical();

        #region 下方的按钮
        GUI.color = Color.yellow;
        //设置按钮中的字体对齐方式为居中对齐。
        GUI.skin.button.alignment = TextAnchor.MiddleCenter;
```

```
            showExtraButton = EditorGUILayout.Foldout(showExtraButton, "额外功能按钮
");
            GUILayout.BeginHorizontal();
            GUILayout.FlexibleSpace();
            GUILayout.BeginVertical();
            GUILayout.FlexibleSpace();
            if (showExtraButton)
            {
                if (GUILayout.Button("复制世界坐标", GUILayout.MaxWidth(300)))
                {
                    //使用 GUIUtility.systemCopyBuffer 获取系统粘贴板的权限，实现“复
制和粘贴”功能
                    GUIUtility.systemCopyBuffer = theTarget.transform.position.ToString();
                }
                if (GUILayout.Button("复制世界空间下的欧拉角",
GUILayout.MaxWidth(300)))
                {
                    GUIUtility.systemCopyBuffer =
theTarget.transform.eulerAngles.ToString();
                }
                if (GUILayout.Button("复制局部坐标", GUILayout.MaxWidth(300)))
                {
                    GUIUtility.systemCopyBuffer =
theTarget.transform.localPosition.ToString();
                }
                if (GUILayout.Button("复制局部空间下的欧拉角",
GUILayout.MaxWidth(300)))
                {
                    GUIUtility.systemCopyBuffer =
theTarget.transform.localEulerAngles.ToString();
                }
                if (GUILayout.Button("复制对象的 Hierarchy 层级路径",
GUILayout.MaxWidth(300)))
                {
                    ExtralEditorUtility.GetGameObjectHierarchyPath(theTarget);
                }
            }
            GUILayout.FlexibleSpace();
            GUILayout.EndVertical();
            GUILayout.FlexibleSpace();
            GUILayout.EndHorizontal();
            GUI.color = Color.white;
            #endregion
        }
```

}

九. RectTransformPlus

对于 RectTransform 组件，我们不能直接继承 Editor。因为 RectTransform 组件的布局是写在 RectTransformEditor 里的，我们为了保持原有的布局不发生改变，就不能在这里直接继承 Editor。

Unity 官方的每一个组件，原则上都有其对应的布局 Editor，而这些 Editor 多数是未公开的。这里我是写了一个单独的 DecoratorEditor，让我可以通过反射去获取到那些未公开的 Editor。后续的按钮组方法就不再像 Transform 组件扩展一样，每一个按钮的名称和调用方法都单独在按钮绘制时编写。

我这里新增了一个 ButtonHandler 类，用这个类存储按钮的显示名称和回调方法。接着在 OnInspectorGUI 函数中使用 for 循环，去遍历 ButtonHandler 的数组，进而在 Inspector 面板上创建出对应的按钮。

最后，每一个按钮的响应，如果是会修改到对象数据的，最好都将操作存储到 Undo 堆栈中，不然使用该工具的小伙伴可能会想打人。

1. 代码

1) RectTransformInspectorPlus

PS：

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

[CustomEditor(typeof(RectTransform))]
public class RectTransformInspectorPlus : DecoratorEditor
{
    //RectTransformEditor 这个类不是一个对外公开的类。所以不能继承它，我们也就无法调用它的 OnInspectorGUI()方法
    public RectTransformInspectorPlus() : base("RectTransformEditor") { }
    private static RectTransform theTarget;
    //静态的标记可以让我们存储上次是要展开还是折叠按钮组
    private static bool showExtraButton = false;
    //按钮组
    private ButtonHandler[] buttonHandlerArray = {
        new ButtonHandler("Reset Position To Zero",ResetPosition),
        new ButtonHandler("Reset RotationTo Zero",ResetRotation),
        new ButtonHandler("Reset Scale To One",ResetScale),
        new ButtonHandler("Reset Scale To Zero",ResetScaleToZero),
        new ButtonHandler("Copy Hierarchy Path",CopyHierarchyPath),
    };



    private void Awake()
```

```
        {
            theTarget = (RectTransform)this.target;
        }

        public override void OnInspectorGUI()
        {
            base.OnInspectorGUI();
            GUI.color = Color.yellow;
            showExtraButton = EditorGUILayout.Foldout(showExtraButton, "额外功能按钮
");
            if (showExtraButton)
            {
                EditorGUILayout.BeginHorizontal();
                for (int i = 0; i < buttonHandlerArray.Length; i++)
                {
                    ButtonHandler temporaryButtonHandler = buttonHandlerArray[i];
                    if (GUILayout.Button(temporaryButtonHandler.showDescription,
"toolbarbutton"))//, GUILayout.MaxWidth(150)
                    {
                        temporaryButtonHandler.onClickCallBack();
                    }
                    GUILayout.Space(5);
                    //一行最多显示两个按钮
                    if ((i + 1) % 2 == 0 || i + 1 == buttonHandlerArray.Length)
                    {
                        EditorGUILayout.EndHorizontal();
                        if (i + 1 < buttonHandlerArray.Length)
                        {
                            GUILayout.Space(5);
                            EditorGUILayout.BeginHorizontal();
                        }
                    }
                }
            }
            GUI.color = Color.white;
        }

        #region  按钮的点击响应
        private static void ResetPosition()
        {
            Undo.RecordObject(theTarget, "ResetPosition To Zero");
            theTarget.localPosition = Vector3.zero;
        }
```

```
        private static void ResetRotation()
        {
            Undo.RecordObject(theTarget, "ResetRotation To Zero");
            theTarget.localRotation = Quaternion.identity;
        }

        private static void ResetScale()
        {
            Undo.RecordObject(theTarget, "ResetScale To One");
            theTarget.localScale = Vector3.one;
        }
        private static void ResetScaleToZero()
        {
            Undo.RecordObject(theTarget, "ResetScale To Zero");
            theTarget.localScale = Vector3.zero;
        }

        private static void CopyHierarchyPath()
        {
            ExtralEditorUtility.GetGameObjectHierarchyPath(theTarget);
        }
        #endregion
    }
```

2) DecoratorEditor

PS：

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

/// <summary>
///一个用于装饰 Unity 内置编辑器类型的编辑器基类，利用反射获取原始布局函数。
/// </summary>
public abstract class DecoratorEditor : Editor
{
    // 用于使用反射调用方法的空数组
    private static readonly object[] EMPTY_ARRAY = new object[0];

    #region Editor Fields

    /// <summary>
    /// 装饰 Unity 的基类类型
    /// </summary>
```

```
        private System.Type decoratedEditorType;

        /// <summary>
        ///  编辑器类型所对应的对象
        /// </summary>
        private System.Type editedObjectType;

        private Editor editorInstance;

        #endregion

        private static Dictionary<string, MethodInfo> decoratedMethods = new
Dictionary<string, MethodInfo>();

        private static Assembly editorAssembly = Assembly.GetAssembly(typeof(Editor));

        protected Editor EditorInstance
        {
            get
            {
                if (editorInstance == null && targets != null && targets.Length > 0)
                {
                    editorInstance = Editor.CreateEditor(targets, decoratedEditorType);
                }

                if (editorInstance == null)
                {
                    Debug.LogError("Could not create editor !");
                }

                return editorInstance;
            }
        }

        public DecoratorEditor(string editorTypeName)
        {
            this.decoratedEditorType = editorAssembly.GetTypes().Where(t => t.Name ==
editorTypeName).FirstOrDefault();

            Init();

            //检查自定义编辑器类型
            System.Type originalEditedType = GetCustomEditorType(decoratedEditorType);
```

```
            if (originalEditedType != editedObjectType)
            {
                throw new System.ArgumentException(
                    string.Format("Type {0} does not match the editor {1} type {2}",
                                  editedObjectType, editorTypeName, originalEditedType));
            }
        }

        /// <summary>
        /// 获取自定义编辑器类型
        /// </summary>
        /// <param name="type">类型</param>
        /// <returns></returns>
        private System.Type GetCustomEditorType(System.Type type)
        {
            BindingFlags flags = BindingFlags.NonPublic | BindingFlags.Instance;

            CustomEditor[] attributes = type.GetCustomAttributes(typeof(CustomEditor), true)
as CustomEditor[];
            FieldInfo field = attributes.Select(editor =>
editor.GetType().GetField("m_InspectedType", flags)).First();

            return field.GetValue(attributes[0]) as System.Type;
        }

        /// <summary>
        /// 初始化
        /// </summary>
        private void Init()
        {
            BindingFlags flags = BindingFlags.NonPublic | BindingFlags.Instance;

            CustomEditor[] attributes =
this.GetType().GetCustomAttributes(typeof(CustomEditor), true) as CustomEditor[];
            FieldInfo field = attributes.Select(editor =>
editor.GetType().GetField("m_InspectedType", flags)).First();

            editedObjectType = field.GetValue(attributes[0]) as System.Type;
        }

        void OnDisable()
        {
            if (editorInstance != null)
            {
```

```
                DestroyImmediate(editorInstance);
            }
        }

        /// <summary>
        /// 根据方法名称去获取对应的方法
        /// </summary>
        protected void CallInspectorMethod(string methodName)
        {
            MethodInfo method = null;

            // 将 MethodInfo 添加到数据字典中
            if (!decoratedMethods.ContainsKey(methodName))
            {
                BindingFlags flags = BindingFlags.Instance | BindingFlags.Static |
BindingFlags.NonPublic | BindingFlags.Public;

                method = decoratedEditorType.GetMethod(methodName, flags);

                if (method != null)
                {
                    decoratedMethods[methodName] = method;
                }
                else
                {
                    Debug.LogError(string.Format("Could not find method {0}", method));
                }
            }
            else
            {
                method = decoratedMethods[methodName];
            }

            if (method != null)
            {
                method.Invoke(EditorInstance, EMPTY_ARRAY);
            }
        }

        public void OnSceneGUI()
        {
            CallInspectorMethod("OnSceneGUI");
        }
```

```
protected override void OnHeaderGUI()
{
    CallInspectorMethod("OnHeaderGUI");
}

public override void OnInspectorGUI()
{
    EditorInstance.OnInspectorGUI();
}

public override void DrawPreview(Rect previewArea)
{
    EditorInstance.DrawPreview(previewArea);
}

public override string GetInfoString()
{
    return EditorInstance.GetInfoString();
}

public override GUIContent GetPreviewTitle()
{
    return EditorInstance.GetPreviewTitle();
}

public override bool HasPreviewGUI()
{
    return EditorInstance.HasPreviewGUI();
}

public override void OnInteractivePreviewGUI(Rect r, GUIStyle background)
{
    EditorInstance.OnInteractivePreviewGUI(r, background);
}

public override void OnPreviewGUI(Rect r, GUIStyle background)
{
    EditorInstance.OnPreviewGUI(r, background);
}

public override void OnPreviewSettings()
{
    EditorInstance.OnPreviewSettings();
}
```

```
        public override void ReloadPreviewInstances()
        {
            EditorInstance.ReloadPreviewInstances();
        }

        public override Texture2D RenderStaticPreview(string assetPath, UnityEngine.Object[]
subAssets, int width, int height)
        {
            return EditorInstance.RenderStaticPreview(assetPath, subAssets, width, height);
        }

        public override bool RequiresConstantRepaint()
        {
            return EditorInstance.RequiresConstantRepaint();
        }

        public override bool UseDefaultMargins()
        {
            return EditorInstance.UseDefaultMargins();
        }
    }
```

3) ButtonHandler

PS：

```
using System;

public class ButtonHandler
{
    public string showDescription = "";
    public Action onClickCallBack;

    /// <summary>
    ///  两参构造函数
    /// </summary>
    /// <param name="Description">文本描述</param>
    /// <param name="callBack">回调方法</param>
    public ButtonHandler(string Description, Action callBack)
    {
        showDescription = Description;
        onClickCallBack = callBack;
    }
}
```

十. 一键替换组件

1. Text 和 Text Mesh Pro 相互替换

PS：

```
using System;
using TMPro;
using UnityEditor;
using UnityEngine;
using UnityEngine.UI;

public class SwitchComponentEditor : Editor
{
    #region 替换 Text 或 TextMeshPro 组件
    #region Property
    private const string MainFontPath = "Assets/Res/Fonts/Main.ttf";
    private const string ArtFontPath = "Assets/Res/Fonts/Art.ttf";
    //Text Mesh Pro 的字体资源
    private const string TMPMainFontPath = "Assets/Res/Fonts/TMPMain/MainSDF.asset";
    private const string TMPArtFontPath = "Assets/Res/Fonts/TMPArt/ArtSDF.asset";
    #endregion

    //[MenuItem("Tools/UI/选中的预制将 TMP 转 Text")]
    [MenuItem("GameObject/UI/SwitchComponent/TMP 转 Text", priority = 3)]
    public static void TMPToText()
    {
        Font mainFont = AssetDatabase.LoadAssetAtPath<Font>(MainFontPath);
        Font artFont = AssetDatabase.LoadAssetAtPath<Font>(ArtFontPath);
        GameObject[] selectObjects = Selection.gameObjects;
        foreach (GameObject theObject in selectObjects)
        {
            TextMeshProUGUI[] allTMP =
theObject.GetComponentsInChildren<TextMeshProUGUI>(true);
            if (allTMP.Length == 0)
            {
                Debug.LogError(theObject.name + "预制件下没有 TMP 组件");
                return;
            }
            else
            {
                foreach (TextMeshProUGUI tmp in allTMP)
                {
                    GameObject tmpObj = tmp.gameObject;
                    string tmpObjName = tmpObj.name;
                    if (tmpObjName.Contains("enhanceTMP"))
                    {
```

```
                string replaceName = tmpObjName.Replace("enhanceTMP",
"enhanceText");
                tmpObj.name = replaceName;
            }

            TMP_FontAsset tmpFontAsset = tmp.font;
            string tmpText = tmp.text;
            float tmpFontSize = tmp.fontSize;
            //string tmpLangKey = tmp.languageKey;
            Color tmpColor = tmp.color;
            TextAlignmentOptions tmpAlignment = tmp.alignment;

            TextAnchor textAnchor = TextAnchor.MiddleCenter;
            switch (tmpAlignment)
            {
                case TextAlignmentOptions.Bottom:
                    textAnchor = TextAnchor.MiddleRight;
                    break;
                case TextAlignmentOptions.BottomLeft:
                    textAnchor = TextAnchor.LowerLeft;
                    break;
                case TextAlignmentOptions.BottomRight:
                    textAnchor = TextAnchor.LowerRight;
                    break;
                case TextAlignmentOptions.Top:
                    textAnchor = TextAnchor.MiddleLeft;
                    break;
                case TextAlignmentOptions.TopLeft:
                    textAnchor = TextAnchor.UpperLeft;
                    break;
                case TextAlignmentOptions.TopRight:
                    textAnchor = TextAnchor.UpperRight;
                    break;
            }

            DestroyImmediate(tmp, true);

            Text temporaryText = tmpObj.AddComponent<Text>();
            if (tmpFontAsset.name == "MainSDF")
                temporaryText.font = mainFont;
            else if (tmpFontAsset.name == "ArtSDF")
                temporaryText.font = artFont;
            temporaryText.text = tmpText;
            temporaryText.fontSize = Convert.ToInt32(tmpFontSize);
```

```
                        //temporaryText.languageKey = tmpLangKey;
                        temporaryText.color = tmpColor;
                        temporaryText.alignment = textAnchor;
                        temporaryText.raycastTarget = false;

                        EditorUtility.SetDirty(theObject);
                    }
                }
            }
            //将所有未保存的资源更改写入磁盘
            AssetDatabase.SaveAssets();
            EditorUtility.DisplayDialog("提示", "转换成功", "确定");
        }

        //[MenuItem("Tools/UI/选中的预制将 Text 转 TMP")]
        [MenuItem("GameObject/UI/SwitchComponent/Text 转 TMP", priority = 4)]
        public static void TextToTMP()
        {
            var mainFont =
AssetDatabase.LoadAssetAtPath<TMP_FontAsset>(TMPMainFontPath);
            var artFont =
AssetDatabase.LoadAssetAtPath<TMP_FontAsset>(TMPArtFontPath);
            GameObject[] selectObjects = Selection.gameObjects;
            foreach (GameObject theObject in selectObjects)
            {
                Text[] allText = theObject.GetComponentsInChildren<Text>(true);
                if (allText.Length == 0)
                {
                    Debug.LogError(theObject.name + "预制件下没有 Text 组件");
                    return;
                }
                else
                {
                    foreach (Text text in allText)
                    {
                        GameObject textObj = text.gameObject;
                        string textObjName = textObj.name;
                        if (textObjName.Contains("enhanceText"))
                        {
                            string replaceName = textObjName.Replace("enhanceText",
"enhanceTMP");
                            textObj.name = replaceName;
                        }
```

```
Font textFont = text.font;
string textText = text.text;
int tmpFontSize = text.fontSize;
//string tmpLangKey = text.languageKey;
Color tmpColor = text.color;
TextAnchor textAlignment = text.alignment;
TextAlignmentOptions tmpAnchor;
switch (textAlignment)
{
    case TextAnchor.UpperLeft:
        tmpAnchor = TextAlignmentOptions.TopLeft;
        break;
    case TextAnchor.UpperCenter:
        tmpAnchor = TextAlignmentOptions.Top;
        break;
    case TextAnchor.UpperRight:
        tmpAnchor = TextAlignmentOptions.TopRight;
        break;
    case TextAnchor.MiddleLeft:
        tmpAnchor = TextAlignmentOptions.Left;
        break;
    case TextAnchor.MiddleCenter:
        tmpAnchor = TextAlignmentOptions.Center;
        break;
    case TextAnchor.MiddleRight:
        tmpAnchor = TextAlignmentOptions.Right;
        break;
    case TextAnchor.LowerLeft:
        tmpAnchor = TextAlignmentOptions.BottomLeft;
        break;
    case TextAnchor.LowerCenter:
        tmpAnchor = TextAlignmentOptions.Bottom;
        break;
    case TextAnchor.LowerRight:
        tmpAnchor = TextAlignmentOptions.BottomRight;
        break;
    default:
        tmpAnchor = TextAlignmentOptions.Center;
        break;
}

DestroyImmediate(text, true);

TextMeshProUGUI tmp =
```

```
textObj.GetComponent<TextMeshProUGUI>();
                            if (tmp == null)
                            {
                                tmp = textObj.AddComponent<TextMeshProUGUI>();
                            }

                            if (textFont.name == "Art")
                                tmp.font = artFont;
                            else
                                tmp.font = mainFont;

                            tmp.text = textText;
                            tmp.fontSize = tmpFontSize;
                            //tmp.languageKey = tmpLangKey;
                            tmp.color = tmpColor;
                            tmp.alignment = tmpAnchor;
                            tmp.raycastTarget = false;

                            EditorUtility.SetDirty(theObject);
                        }
                    }
                }
                //将所有未保存的资源更改写入磁盘
                AssetDatabase.SaveAssets();
                EditorUtility.DisplayDialog("提示", "转换成功", "确定");
            }

            [MenuItem("GameObject/UI/SwitchComponent/TMP 转 TMP_Plus", priority = 1)]
            public static void TMPToTMPPlus()
            {
                //TODO TMPPlus 组件还没有写
            }

            [MenuItem("GameObject/UI/SwitchComponent/Text 转 TMP_Plus", priority = 2)]
            public static void TextToTMPPlus()
            {
                //TODO TMPPlus 组件还没有写
            }
            #endregion
        }
```

十一. 检查 UI 中的 RaycastTarget

重写 UI 组件创建

十二. PropertyAttribute 和 PropertyDrawer 完成编辑器绘制复用

在自定义 Inspector 面板显示时会经常性的出现重复的内容，对于这部分重复而简单的内容，我们可以使用 PropertyAttribute 和 PropertyDrawer 来完成绘制代码的复用。

我这里将 PropertyAttribute 放置在 Sccripts 文件夹下，而 PropertyDrawer 是绘制 GUI 的，则放置在 Editor 文件夹下。

1. ReadOnly

有时候我们仅仅是想让一些属性和字段显示在 Inspector 面板上，但是不希望用户能修改其值。这时就需要在 PropertyDrawer 的派生类中使用 EditorGUI 里的 BeginDisabledGroup 和 EndDisabledGroup，它们会禁用组内的控件交互。

考虑到 ReadOnly 的应用场景，我们可以添加一个枚举去判断何种状态下才执行这一限制。

而在 PropertyAttribute 的派生类中，我们需要添加一些构造函数，来便于外部的使用。对于这一派生类，我们还可以使用 AttributeUsage 来限定其使用范围。

1) 代码

a. PropertyAttribute

PS：

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

public enum ReadOnlyType
{
    Common,//所有情况都是 ReadOnly
    Prefab,//只有在预制体模式下是 ReadOnly
    Play,//只在游戏运行时是 ReadOnly
}

[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field, AllowMultiple = false, Inherited = true)]
public class ReadOnlyAttribute : PropertyAttribute
{
    public ReadOnlyType TheReadOnlyType { get; private set; }
    /// <summary>
    /// 无参构造函数默认为 Common
    /// </summary>
    public ReadOnlyAttribute()
    {
        this.TheReadOnlyType = ReadOnlyType.Common;
    }
```

```
        public ReadOnlyAttribute(ReadOnlyType theType)
        {
            this.TheReadOnlyType = theType;
        }

    }
```

b. PropertyDrawer

PS：

```
    using UnityEngine;
    using UnityEditor;

    [CustomPropertyDrawer(typeof(ReadOnlyAttribute))]
    public class ReadOnlyDrawer : PropertyDrawer
    {
        public override void OnGUI(Rect position, SerializedProperty property, GUIContent
label)
        {
            ReadOnlyAttribute targetAttribute = (ReadOnlyAttribute)attribute;
            bool readOnly = true;
            switch (targetAttribute.TheReadOnlyType)
            {
                case ReadOnlyType.Prefab:
                    //处于预制体查看状态禁用组内的控件
                    readOnly =
UnityEditor.SceneManagement.PrefabStageUtility.GetCurrentPrefabStage() != null;
                    break;
                case ReadOnlyType.Play:
                    //处于游戏运行状态时禁用组内的控件
                    readOnly = Application.isPlaying;
                    break;
            }
            EditorGUI.BeginDisabledGroup(readOnly);//设置是否禁用组内的控件
            EditorGUI.PropertyField(position, property, label);
            if (readOnly)
                EditorGUI.EndDisabledGroup();
        }
    }
```