前言

> SO主导的游戏架构真正开启了我对架构更深一层的理解，之后学习依赖注入等解耦手段我也经常与SO架构对比，强烈推荐如下的两个介绍SO架构的演讲
>
> - Unite Austin 2017 - Game Architecture with Scriptable Objects - YouTube
> - Unite 2016 - Overthrowing the MonoBehaviour Tyranny in a Glorious Scriptable Object Revolution - YouTube
>
> 同样，这篇文章质量只能算是草稿，因为写得比较早，认识不全面，内容可能现在看来只是很粗浅的理解

SO主导的游戏架构

- 核心思想，利用SO传递数据
  - ScriptableObject继承UnityEngine.Object，属于引用类型，在Inspector面板中
  - SO只存在一份，所以数据可以保持一致性
  - 将事件包装在SO中，可以通过SO传递事件，达成解耦
  - 作为固定的数据，SO恰好可以满足内存中只存在一份；对于运行时动态改变的数据，SO可以充当运行时暂存的容器，持久化可以通过保存到本地文件实现
  - 可以通过在运行时从文件中读取信息改变ScriptableObject的值，虽然它不会被保存，但能保证运行时共用这一份数据
  - SO的特殊性在于，可以通过SO编程，同时因为它是一种资产，保持了在内存中的唯一性，可以像单例一样使用
    - InputSystem也是类似的道理，所以可以将多个action asset在每个脚本中都引用一份，因为只是引用
    - SO和MonoBehaviour不同之处
      - SO实例是资产，使用时一般是使用已经存在的实例，脚本中引用Assets中的SO资产后不会复制，可以实现唯一性
      - MonoBehaviour使用一般需要把脚本挂载到GO上，是创建一份实例，每次创建实例都是复制一份新的脚本，无法做到唯一性
- 缺陷
  - SO动态实例化较难完成解耦，例如不同的敌人有不同的血条，都是使用HealthSO这一类型的SO，但是需要动态实例化，但是让他们的UI血条连接到该动态实例化的SO比较困难（某种程度上，是一个动态实例化的敌人的血条变化事件是动态的）【如果敌人血条是固定的就可以预先创建SO记录信息，就不存在问题，例如SLG游戏中每个关卡敌人数据是记录在地图文件中的，但可能造成SO泛滥】

一些问题

如何解决SO的加载保存

> - 加载
>   - 方案1： 在Inspector面板中指定，SO作为资产导入到场景中，与脚本关联，只要场景中的MB脚本加载就会创建SO实例（导入资源到场景中），多个脚本引用同一个SO不会造成重复导入的问题
> - 方案2：不在Inspector面板中指定，使用代码创建（具体代码？）
> - SO的数据加载不会在SO内部写一个Init函数然后判断是否SO存在然后调用，而是

```
// 加载SO数据的示例
 public IEnumerator LoadSavedInventory()
 {
     // 清空需要载入数据的SO
     _playerInventory.Items.Clear();
     // 遍历存档中记录的item
     foreach (var serializedItemStack in saveData._itemStacks)
     {
         // 加载关联的itemSO，此处的Addressables估计使用了ItemSO的guid作为asset的key，所以itemSO会存储一个guid
```

```
            // 如果是纯值类型SO，直接读取文件信息即可
            var loadItemOperationHandle = Addressables.LoadAssetAsync<ItemSO>
(serializedItemStack.itemGuid);
            yield return loadItemOperationHandle;
            if (loadItemOperationHandle.Status == AsyncOperationStatus.Succeeded)
            {
                var itemSO = loadItemOperationHandle.Result;
                // 读入数据
                _playerInventory.Add(itemSO, serializedItemStack.amount);
            }
        }
    }
```

- 保存
  - 方案：保存至本地文件中

如何运行时产生同一类SO的不同实例？如何让其他需要共享的组件获取动态创建的SO实例？

- 如何让同一个SO有不同的实例，inspector面板不指定SO，我们需要让他动态创建一个SO实例
- 运行时的动态数据SO一般都需要用CreateInstance动态创建（主角的是预先创建的，但NPC的是动态创建的）
- 影响不大，可以通过创建列表等结构来动态管理多个数据，就像背包系统一样，动态创建不是刚需
- 以SLG战斗为例，场上敌人的固有信息可以以SO形式存储（每张map一个文件夹），另外搞一个SO存储列表记录场上敌人的信息，游戏开始时，根据map文件信息先加载敌人的prefab（SO在其中），然后遍历所有敌人获取SO信息记录到列表SO，这样每个个体的SO就构成一个整体，这样就可以获取敌人的数据了，虽然不是动态创建的，但说明动态创建的需求不是刚需

```
private void Awake()
{
    //If the HealthSO hasn't been provided in the Inspector (as it's the case for the player),
    //we create a new SO unique to this instance of the component. This is typical for enemies.
    if (_currentHealthSO == null)
    {
        _currentHealthSO = ScriptableObject.CreateInstance<HealthSO>();
        _currentHealthSO.SetMaxHealth(_healthConfigSO.InitialHealth);
        _currentHealthSO.SetCurrentHealth(_healthConfigSO.InitialHealth);
    }
    // 对于主角来说，可以挂载事件SO让UI更新，但对敌人角色来说就不能这么做了，因为没有办法为每个敌人都设置一个事件
    if (_updateHealthUI != null)
        _updateHealthUI.RaiseEvent();
}
```

SO能做到什么程度，事件的分发是否有限制，性能消耗怎么样，有没有必要用其他的框架搭配

接口搭配SO使用，可以让SO成为类似SingletonManager的存在

SO可以包含其他的SO，事件只是很小的一部分开销，与SingletonManager中放置分发事件没有性能上的差别

在一个开源的大型项目中也使用了SO传递事件，可选择将多个事件装在一个SO里，也可以分开存放，分开存放更方便debug，但需要多次拖拽到inspector多次创建SO实例比较麻烦

可以搭配工厂模式等使用，SO实现实例化的工作，返回的实例通过一个manager保存

可以设置一个Manager处理所有的一类event（通过SO传递数据和事件）

有些时候一些manager之间的互相引用可以使用接口反射实现（融入QFramework，但将反射仅限制在获取依赖时使用，保证只在Start或Awake阶段调用）

一个SO只传递一个事件会不会太浪费

> 不会，当然也可以让多个对同一个数据响应的事件放在一个SO里
>
> 一般来说事件按照参数个数和类型分类为不同的SO类型，然后根据需要的事件创建不同的实例
>
> 事件的消耗，参考UnityEvent或者UnityAction的消耗对比文章
>
> 事件可以使用event Action委托，而不使用UnityEvent(参考InputSystem)

演讲者提出的一些实际开发使用方式

> 解决使用编辑器内使用SO每次都会修改SO的值的办法
>
> > For the games that I have shipped with this, I have a Value to change at edit time and a non-serialized RuntimeValue that is used when the game is running. You can set RuntimeValue to value on first access or using a ISerializationCallbackReceiver.
> >
> > Also, for our float variables in production I have the concept of min and max built int to each one.
>
> SO持久化需要考虑的
>
> > There are 2 levels of persistence worth thinking about here. First is between scenes. We kind of get that for free with Scriptable Objects. The next is between play sessions. That can be easily solved by having a variable save and load through something like player prefs or whatever your serialization model is.
> >
> > It then needs to be up to the game logic to decide when each thing saves and loads, but it can be nice and modular to have each variable track their own save key.
>
> 如何解决动态实例化的SO的解耦（以敌人的血条UI为例）
>
> 其他用户的评论：虽然动态实例化的SO的绑定引入了依赖，不能算是解耦，但也比以前会更好，因为代码的复用性提升了，血条依赖的是SO，而不是具体的玩家/敌人
>
> 演讲者的评论：有些游戏就是不太适合用，要视情况使用SO架构，没有中心化的反馈系统通常就是不需要使用SO架构的原因，比如随处可见的小怪的行为，不需要其他系统介入（玩家则不同，玩家的行为涉及到很多系统进行中心化处理）

SO架构的一些参考代码

> - baratgabor/Unity3D-ReactiveScriptables: ScriptableObject based framework / scaffolding that facilitates loosely coupled communication and automatic update propagation between MonoBehaviour components. (github.com)
> - Unite2017/Assets/Code/Inventory at 5f88315fceaf9104dc09ba062f62daa2cbdafa41 · roboryantron/Unite2017 (github.com)

SO架构的问题

> - 仍需要做一些编辑器方面的工作来显示SO的引用情况
> - SO需要注意生命周期，如果SO不再被引用，会被GC回收，那么未保存的SO数据将丢失（与static不同的地方，static需要显示置null才会回收指向的对象）
> - 对于不变的数据，更适合通过SO存储，这样很容易在编辑器面板看到，而不是修改代码（每个数据提前创建SO实例）；或者通过表格文件导入数据到SO中（动态创建SO实例）
>   - 例如卡牌游戏，每张卡片附带一个SO数据
>   - 例如背包系统，每个物品自带一个SO数据
>   - 至于这个SO是否用于解耦，可以看下UOP中的背包是怎么做的

- 对于运行时变化的数据，static/SO各有优缺点
  - SO相比static的好处是SO可以方便的更改数据，更容易设计，代码风格不是硬编码引用
  - static相比SO的优点是生命周期更容易控制，无需使用inspector面拖拽

Singleton和static

- Singleton就是通过static实例实现的全局访问
- 使用static设置全局可访问的数据不一定使用Singleton（具体分析见SO vs Static）

RuntimeSet用法

- 脚本中引用的SO存储一个列表，在脚本开启关闭时是脚本加入/移除列表的标志，可以用于查看处于活跃状态的引用了某个SO的脚本

SO作为资产，使用AssetRefrence会更好吗

- 如果是静态（运行时不改变）的数据，可能无所谓
- 对于动态的数据，要看需不需要提前在inspector中指定，因为动态创建的SO实例不用asset reference
- 综上，统一不用，或者只对通过inspector面板指定的SO使用

SO的生命周期如何管理

- SO的生命周期和普通资源一样，没有引用时会卸载
- DontUnloadUnusedAsset可以保证不被卸载，
- 以下测试在编辑器内完成
  - 创建SO资产时会调用Awake和OnEnable
  - 刷新asset会触发Disable和Enable，也就是在编辑器内asset处于被load的状态（即使场景没有引用SO资产），每次刷新asset database都会先卸载，然后重新加载
  - 进入playmode，但场景中没有SO资产，会触发Disable和Enable
  - 进入playmode，但场景中有SO资产，会触发Disable和Enable，然后触发Awake
  - playmode中将SO资产置为null，没有触发Disable和Destroy
  - playmode中去往没有SO资产引用的场景，会触发Disable
  - playmode中回到有SO资产引用的场景，会触发Awake+Enable，反复加载到有SO引用的场景，Awake和Enable不会多次触发，说明一直在内存中
  - editor中退出playmode没有触发Disable
  - 如果动态创建一个SO实例，那么在当前场景直接退出程序会触发这个临时SO的Disable和Destroy，但如果切换场景后，这个SO的任何消息都不会被调用
  - 有时候切换场景可能不会调用So的Disable，再次进入也不会调用Awake和enable，这些生命消息函数和内存有关，和GC调用时机有关，不是完全可控
- 以下测试为构建版本完成
  - 脚本中声明SO，但不赋值，不会调用任何消息函数
  - 脚本中挂载了SO资产，启动后自动调用Awake和OnEnable，直接关闭程序窗口会调用OnDisable，但没有调用OnDestroy
  - 将SO资产引用置为null，没有触发Disable和Destroy
  - 切换到没有SO资产引用的场景，将调用Disable，但是没有调用Destroy
- 论坛中关于Destroy的调用说法，结合实验和论坛说法
  - 删除SO脚本才会调用，删除资产也不会调用
  - 运行时动态创建的SO实例在退出程序时会调用Destroy，
- 关于Awake
  - SO被创建时调用（Editor/Runtime），Awake，即CreateInstance
  - 有SO引用的场景加载时调用Awake
- 关于Enable

    - 在Awake之后会立即调用（创建时调用，或者runtime时进入一个有SO引用的场景中）
    - 编辑器内重新加载，在Disable之后调用
    - 编辑器内进入playmode之后会调用，在Disable之后调用
- 关于Disable
    - 进入一个没有SO引用的场景
    - 在Destroy之前调用
    - 编辑器内重新加载
    - 编辑器内进入playmode之后会调用
- 执行顺序，发现SO的消息调用先于引用它的脚本

没有SO如何替代

- 静态数据的SO，相当于全局静态变量，通过文件读取值，静态全局访问，硬编码依赖
- 动态数据的SO，创建普通数据类替代，实例化，通过IoC容器实现依赖注入

event（特指没有固定发布者的事件）是否会更适合用代码写（静态事件）

- 用SO存储事件的好处，内存方面更节约
- 静态事件的好处，稳定，不用担心生命周期
- 认识ECS后的想法：无论是static还是SO都是引入依赖，无论哪种方式只要管理的好实现的效果都差不多，感觉static实例化一个包含事件的类的做法会更好一点，SO太多不方便
- 有待扩展。。。。

如何让SO的引用在编辑器中可见

- 以下脚本可以在编辑器中非运行时找到脚本对应的SO实例，unity自带的功能，点击资产右键FindReferences in Scene可以看到场景中引用SO实例的对象
- ```
  using System;
  using System.Reflection;
  using UnityEditor;
  using UnityEngine;

  namespace Editors {
      class EditorProjectWindowExtensions {
          [MenuItem("Assets/Find ScriptableObject Instances",false, 30)]
  private static void DoSomethingWithVariable() {
      SetSearchString("t:" + Selection.activeObject.name);
  }

  // Note that we pass the same path, and also pass "true" to the second argument.
  [MenuItem("Assets/Find ScriptableObject Instances", true)]
  private static bool NewMenuOptionValidation()
  {
      if (!(Selection.activeObject is MonoScript)) {
          return false;
      }
      MonoScript o = (MonoScript) Selection.activeObject;
      Type ot = o.GetClass();
      return ot.IsSubclassOf(typeof(ScriptableObject));
  }

  private static void SetSearchString(string searchString)
  {
      Type projectBrowserType = Type.GetType("UnityEditor.ProjectBrowser,UnityEditor");
  ```
  ```

```
            if (projectBrowserType == null) {
                return;
            }
            MethodInfo setSearchMethodInfo = projectBrowserType.GetMethod("SetSearch",
BindingFlags.NonPublic | BindingFlags.Public | BindingFlags.Instance, null, new Type[] {
typeof(string) }, null);
            if (setSearchMethodInfo == null) {
                return;
            }

            EditorWindow window = EditorWindow.GetWindow(projectBrowserType);
            setSearchMethodInfo.Invoke(window, new object[] {searchString});
        }
    }
}
```

- 可能需要的其他扩展：找到Assets下引用某个SO实例的所有资产

Unity论坛上关于SO的观点

> - SO存储临时（动态）数据不合适，SO更应该向json文件一样使用，SO的方便之处在于可以直接引用资产资源
> - SO并没有减少依赖，只是将依赖可视化了；SO也是一种Singleton；Singleton也可以是多态的，通过门面模式等
> - SO做的事通过event也能做
> - SO需要在inspector面板指定，非常不好
> - 更喜欢使用代码导向的结构，而非借助SO的架构
> - 一般来说每个敌人的当前血量不会使用SO
> - 更愿意只让SO存储持久化的数据，不让其运行脚本
> - SO作为一种资产不应该动态创建
> - SO的优点在于可编辑，跨场景
> - SO也有很多好评，比较适合小游戏，具体多小，需要实践

演讲者补充

> - In our system, there are three ways to register for an event: GameEventListener MonoBehaviour, interface, and delegate.
>
>   The MB one is the most flexible since it does not require any code changes. It does use a MB per event responded to so we kept the editor simple with an optional "Advanced" dropdown. If not planned, this list can get large, but if you properly modularize your systems, each prefab will only be concerned with a small number of events since it does only one thing. Enemies are only concerned with combat and high level state related events so even if your game gets bigger, they should not observe every new system.
>
>   The interface option allows us to do the response in code. This is for things that we know need to respond to certain events to have any functionality. It is less flexible but faster and cleaner on the editor. A MB implementing the interface takes the events it responds to in the editor and the response is done in code. There is no need to serialize the interface references since they register at runtime.
>
>   The delegate option is similar to the interface one, but makes it easier for a class to respond to multiple events.
>
> - My Unite 2017 Talk - Game Architecture with Scriptable Objects (roboryantron.com)

单例的SO

```
public class SingleScriptableObject<T> : ScriptableObject where T :ScriptableObject
{
 //所有数据资源文件都放在Resources文件夹下加载对应的数据资源文件
 //对需要复用的唯一的数据资源文件名定一个规则：文件名和类名一致
 private static string scriptableObjectPath = "ScriptableObject/"+typeof(T).Name;
 private static T instance;
 public static T Instance
 {
     get
     {
         if (instance == null)
         {
             //如果为空，首先应该去资源路径下加载对应的数据资源文件
             instance = Resources.Load<T>(scriptableObjectPath);
         }
         //如果没有这个文件，直接创建一个数据
         if (instance == null)
         {
             instance = CreateInstance<T>();
         }
         return instance;
     }
 }
}
```

- 不用在inspector面板指定，几乎和static(单例)做法一致，引入了SO的特性便于Debug
- 但是只适用于一个SO脚本对应一个SO资产，意味着失去了软引用的特性

SO和表格的平衡

- 商业游戏不用SO
- scriptable object会直接存储关联asset的guid
- 关系型数据通常都会为每一条数据设置一个索引，例如excel，非关系型数据通常是键值对方式存储，包括json、xml、ScriptableObject等
- excel写表格，然后利用插件自动生成scriptobj，完美利用两者特性

SO的扩展

> 可以用来调试数据
>
> 可以在SO的编辑器界面实现一些OnGUI函数测试数据效果（比如音效）