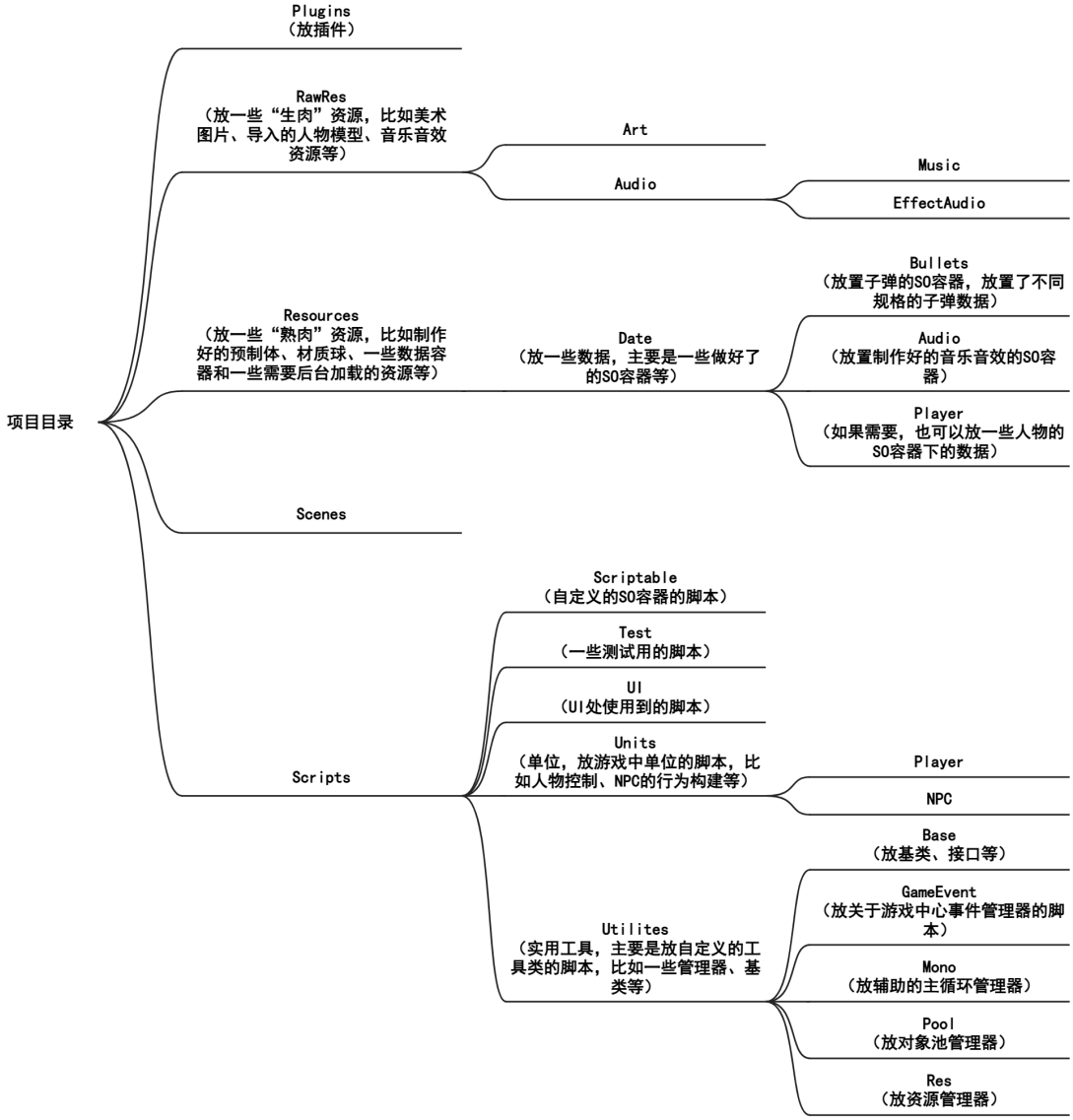
项目层次结构说明

如图：



单例模式管理器模块

管理器基类

封装好了的单例模式管理器基类：ManagerBase<T>。

使用方式一：单例管理器

写需要的单例管理器时，建议继承此类在进行书写。使用时直接继承即可：

```
public class MonoManager : ManagerBase<MonoManager>
{

}
```

主循环辅助模块

主循环管理器（MonoManager）

单例模式不继承MonoBehavior后，可能有些地方需要用到主循环的东西，比如开个协程啥的。

可以将需要开启的协程或者需要加入帧更新的事件交给MonoManager，他可以同一开启。

使用方式如下：

```
MonoManager._Instance.XXX
```

应用1：协助在帧更新中运行、移除函数

案例：Test类不继承MonoBehavior，无法使用unity主线程的Update帧更新等生命周期函数。此时，封装的方法RegisterUpdateEvent()，在MonoManager中注册需要帧更新的函数，即可让函数每帧运行。

```
public class Test
{
    public Test()
    {
        //在Test类被new出来时，注册要加入帧更新的函数。
        MonoManager._Instance.RegisterUpdateEvent(TestFun);

        //在Test类被new出来时，移除之前加入帧更新的函数。
        MonoManager._Instance.RegisterUpdateEvent(TestFun);
    }

    public void TestFun()
    {
        Debug.Log("帧更新测试");
    }
}
```

应用2：协助开启协程

案例：Test类不继承MonoBehavior，无法直接开启协程。此时能够通过MonoManager封装的方法StartCoroutine()，将需要开启的协程传给MonoManager代理开启。

```
public class Test
{
    public Test()
    {
        //在Test类被new出来时，开启协程test()。
        MonoManager._Instance.StartCoroutine(test());
    }

    IEnumerator test()
    {
        Debug.Log("协程测试");
```

```
        yield return new WaitForSeconds(1.0f);
        Debug.Log("协程测试2");
    }
}
```

游戏事件模块

游戏中的某一个事件可以需要协调好几个不同的物体响应，这时可以通过一个“中心”注册事件监听、在“监听列表”中添加委托事件。当需要触发这个事件监听时，只需要告知“中心”，“中心”会将触发事件监听中的所有事件。

游戏事件管理器（GameEventManager)

管理游戏中事件的中心。

使用方式如下：

```
GameEventManager._Instance.XXX
```

应用1：注册事件

```
public void RegisterEventListner(string eventName, UnityAction doSoming);
public void RegisterEventListner<T>(string eventName, UnityAction<T> doSoming);
```

- 注册事件的函数RegisterEventListner()有一个重载。doSoming要么是无参委托，要么是有参委托。
- eventName: 事件监听
- doSoming: 要注册添加的事件

```
//注意：注册监听时，所有的委托需要使用同一种重载方式。如：
//Test、Test2注册事件监听时使用的重载一定要是一样的，即要么都是有参的、要么都是无参的。
public class Test : MonoBehaviour
{
    private void Start()
    {
        GameEventManager._Instance.RegisterEventListner<BulletType>("子弹扣血",
(obj) =>
        {
            Debug.Log("In Test");
        });
    }
}

public class Test2 : MonoBehaviour
{
    private void Start()
    {
        GameEventManager._Instance.RegisterEventListner<BulletType>("子弹扣血",
(obj) =>
        {
            Debug.Log("In Test2");
```

```
        });
    }
}
```

应用2：移除事件

```
public void RemoveEvent(string eventName, UnityAction doSoming);
public void RemoveEvent<T>(string eventName, UnityAction<T> doSoming);
```

- 移除事件的函数RemoveEvent()有一个重载。doSoming要么是无参委托，要么是有参委托。
- eventName: 事件监听的名字
- doSoming: 需要移除的事件

应用样例1：事件无参

比如：有一个“游戏结束”的事件。其触发时，1.需要让玩家消失 2.需要让屏幕变成黑色。

分别在：Player、Lens注册了事件监听触发时的事件后，在Test类中触发测试。

```
public class Player : MonoBehaviour
{
    private void Start()
    {
        //为"游戏结束"注册事件
        GameEventManager._Instance.RegisterEventListner("游戏结束", () =>
        {
            this.gameObject.SetActive(false);
        });
    }
}

public class Lens : MonoBehaviour
{
    public Image bk;

    private void Start()
    {
        //为"游戏结束"注册事件
        GameEventManager._Instance.RegisterEventListner("游戏结束", () =>
        {
            Color color = bk.color;
            color.a = 255;
            bk.color = color;
        });
    }
}

public class Test : MonoBehaviour
{
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
```

```
        {
            //"游戏结束"触发
            GameEventManager._Instance.EventTrigger<BulletType>("游戏结束",);
        }
    }
}
```

应用样例2：事件有参

比如：有一个“子弹扣血”的事件。其触发时，1.玩家减血10（样例方便起见随意定的） 2.需要让屏幕变成红色2秒，随后变回原样。

同样的分别在：Player、Lens注册了事件监听触发时的事件后，在Test类测试

```
public class Player : MonoBehaviour
{
    public int HP = 100;
    private void Start()
    {
        //为"子弹扣血"注册事件
        GameEventManager._Instance.RegisterEventListner("子弹扣血", (obj) =>
        {
            HP -= obj.deHp
        });
    }
}

public class Lens : MonoBehaviour
{
    public Image bk;

    private void Start()
    {
        //为"子弹扣血"注册事件
        GameEventManager._Instance.RegisterEventListner("子弹扣血", (obj) =>
        {
            MonoManager._Instance.StartCoroutine(ZiDanFun());
        });
    }
    IEnumerator ZiDanFun()
    {
        Color color = bk.color;
        color.a = 255;
        color.r = 255;
        bk.color = color;
        yield return new WaitForSeconds(2.0f);
        color.a = 0;
        color.r = 0;
        bk.color = color;
    }
}
```

```
public class Test : MonoBehaviour
{
    //bulletType是提前写好的（某类）子弹的数据资产，存储着子弹的速度、伤害之类的数据。
    public BulletType bulletType;

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            //"子弹扣血"触发
            GameEventManager._Instance.EventTrigger<BulletType>("子弹扣
血",bulletType);
        }
    }
}
```

对象池模块

一些需要生成然后复用的对象可以用这个模块来实现，比如特效、子弹等。

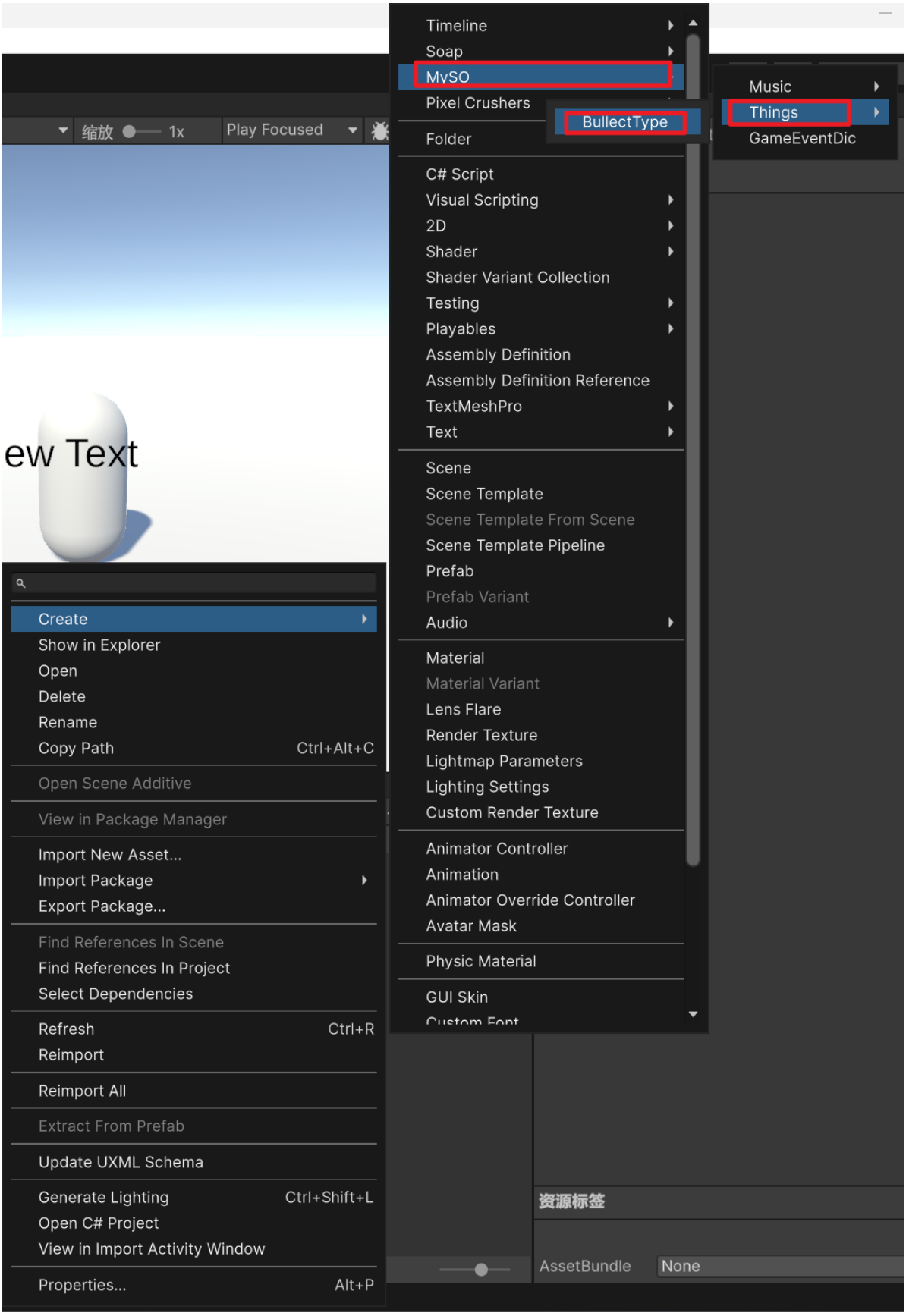
对于需要复用的对象，建议提前写好其脚本和数据容器。

对象容器

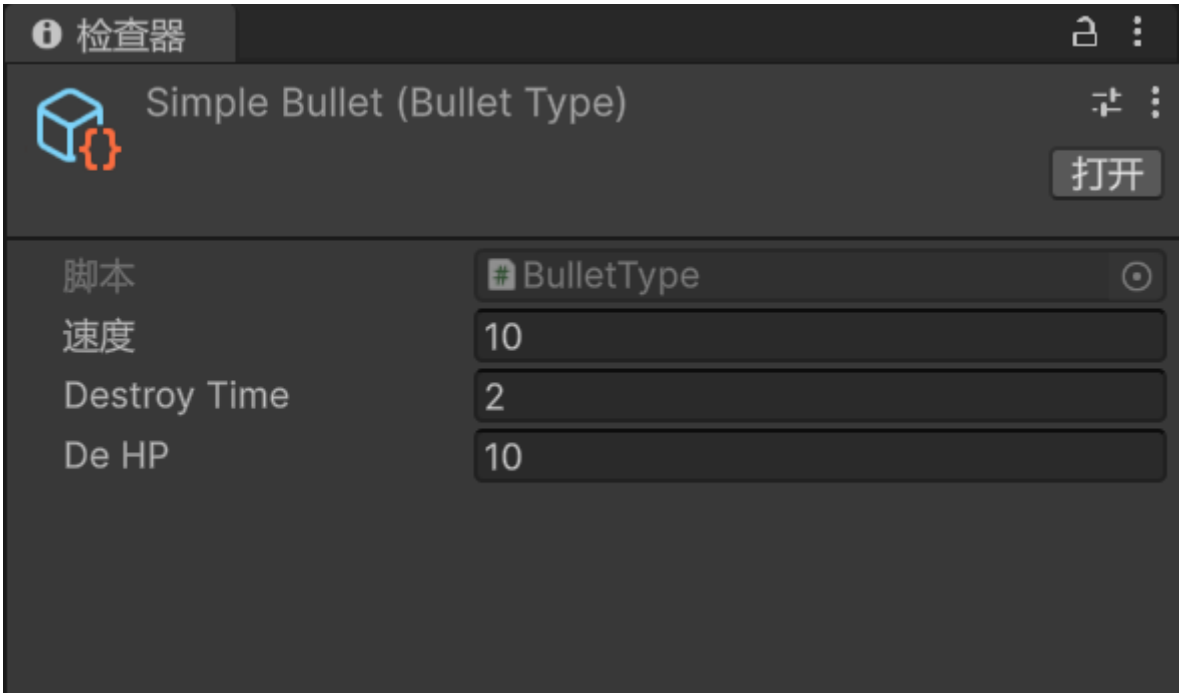
样例1：子弹

以子弹为例，子弹的显示、消失等动态逻辑，可以写入Bullet类中，挂载在预制体上。而其静态的一些属性，例如子弹的运行速度、子弹的伤害值，可以写成数据容器形式。

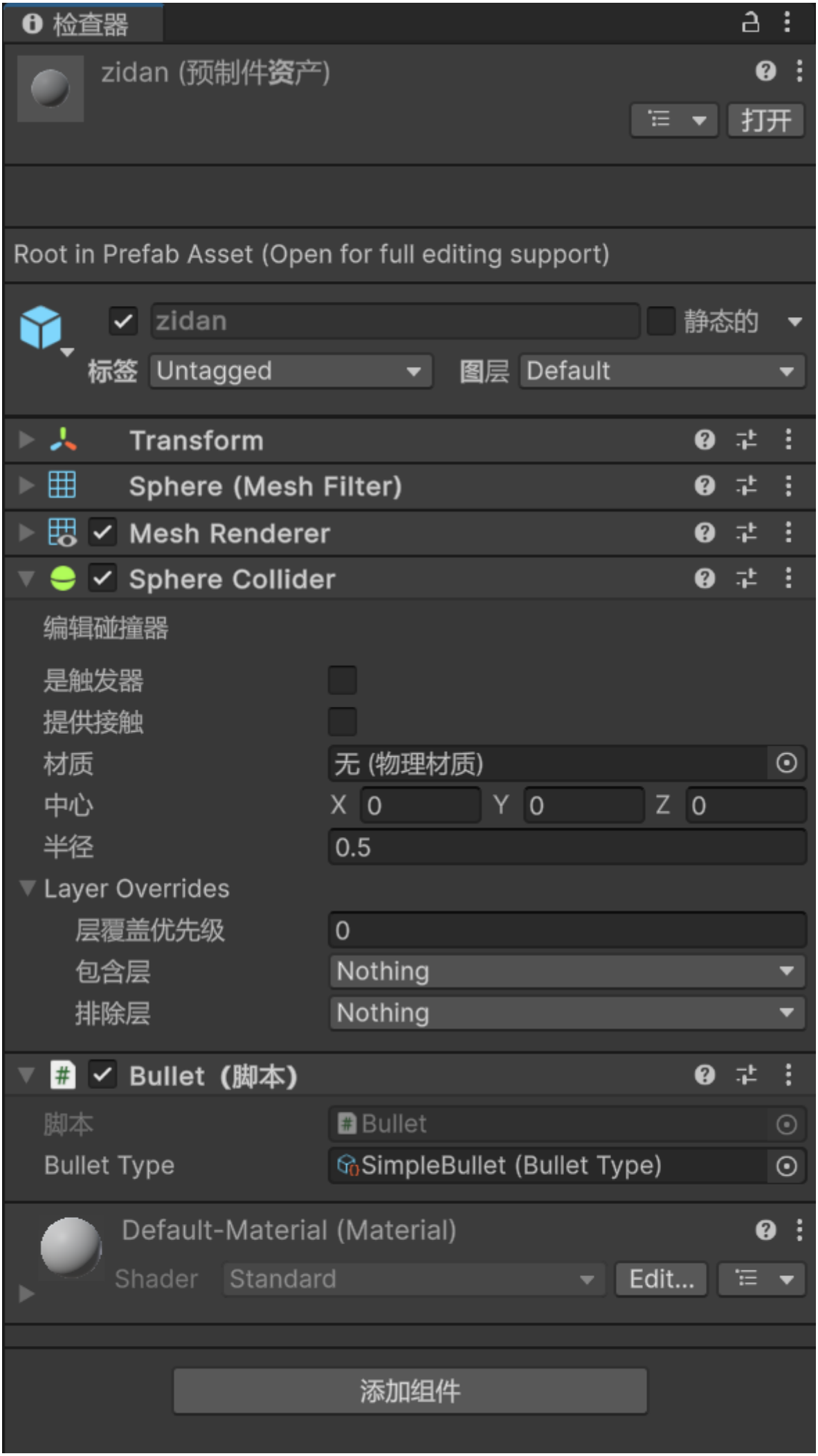
创建子弹容器：



填写“Simple Bullet”这种子弹的数据：



在对应“SimpleBullet”的逻辑脚本中装载：


检查器
zidan (预制件资产)
打开
Root in Prefab Asset (Open for full editing support)
zidan
静态的
标签
Untagged
图层
Default
Transform
Sphere (Mesh Filter)
Mesh Renderer
Sphere Collider
编辑碰撞器
是触发器
提供接触
材质
无 (物理材质)
中心
X 0
Y 0
Z 0
半径
0.5
Layer Overrides
层覆盖优先级
0
包含层
Nothing
排除层
Nothing
Bullet (脚本)
脚本
Bullet
Bullet Type
SimpleBullet (Bullet Type)
Default-Material (Material)
Shader
Standard
Edit...
添加组件

对象池管理器（PoolManager）

PoolManger在某个对象使用完后，可以将对象压入对象池中，等到下次需要再从对象池中压出使用。对象池对象数量不够时，会自动实例化。

使用方式如下：

```
PoolManager._Instance.XXX
```

应用1：生成对象

生成对象，就是将对象从对象池中弹出使用。

```
public void PopObj(string name,UnityAction<GameObject> callBack);
```

- name：对象（池）的名字
- callBack：对这个弹出的对象，需要做什么
- **注意：需要生成的对象一定要放在指定的文件路径下（默认路径是"Prefab/"）**

应用2：销毁对象（让其消失）

销毁只是收回对象池中，并没有从内存中销毁。

```
public void PushObj(string name,GameObject obj);
```

- name：对象（池）的名字
- obj：需要压入对象池的对象

应用1、2脚本例子

生成一颗子弹，1秒钟后消失：

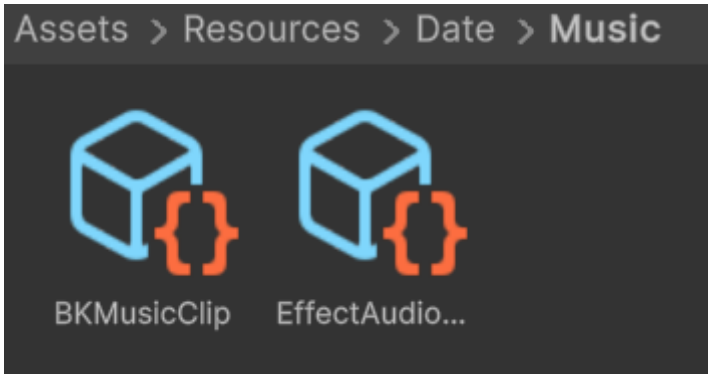
```
public class Test2 : MonoBehaviour
{
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            //MonoManager._Instance.StartCoroutine(test());
            StartCoroutine(test());
        }
    }

    IEnumerator test()
    {
        GameObject Obj = null;
        PoolManager._Instance.PopObj("Bullet/zidan", (obj) =>
```

```
        {
            Obj = obj;
        });
        yield return new WaitForSeconds(1.0f);
        PoolManager._Instance.PushObj("Bullet/zidan", Obj);
    }
}
```

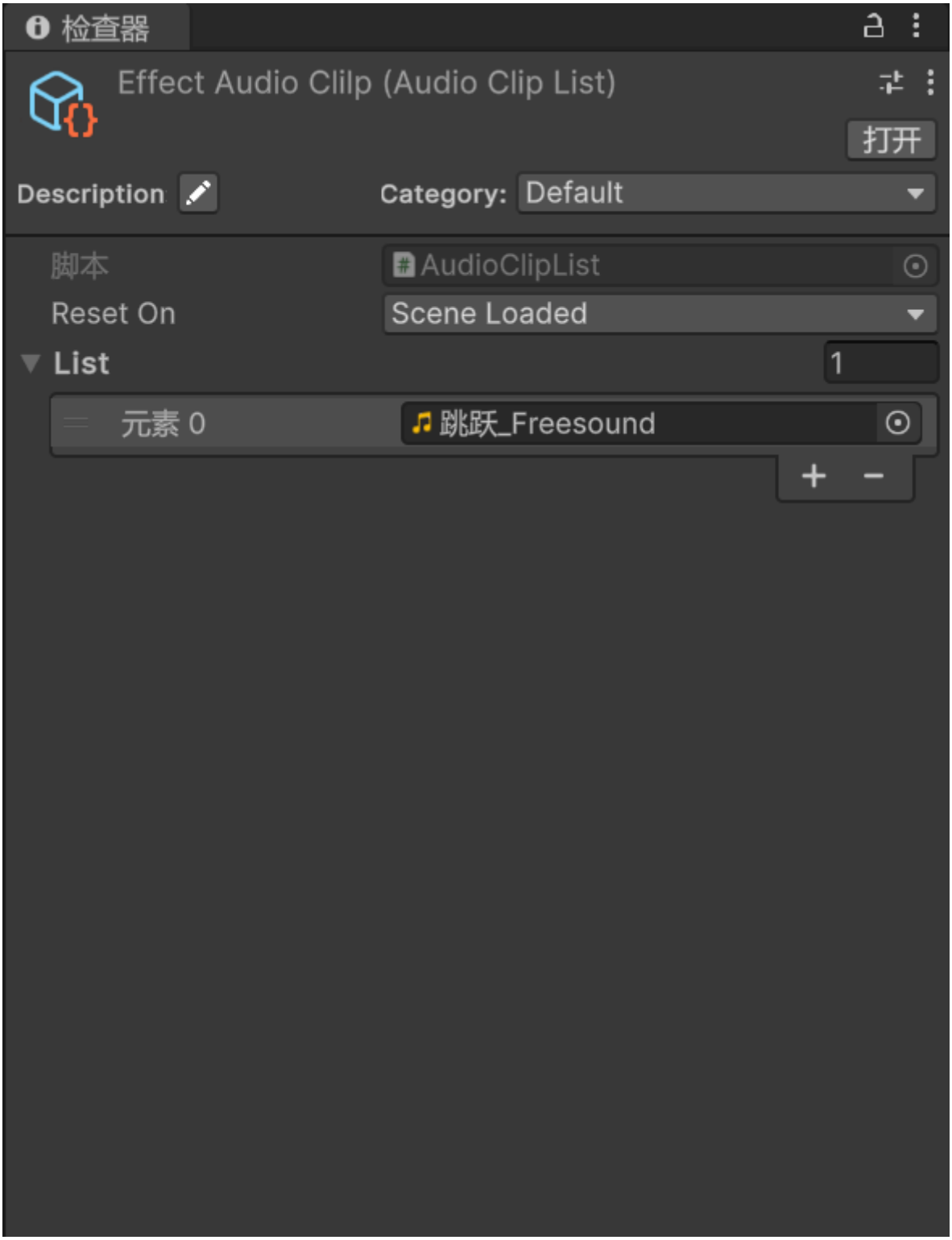
音乐音效模块

音乐音效容器

用来装音乐、音效资源，可从后台直接调用。已经生成放好在项目中，直接添加即可。



添加方式直接添加然后拖入即可：



音乐音效管理器（MusicManager）

没写，挺简单的，建议根据容器来写。

资源管理模块

资源管理器（ResManager）

将Resources.Load<T>这一系列加载资源的函数进行了一些封装，有同步的也有异步的方法。

```
ResManager._Instance.XXX
```

应用1：同步加载资源（位于Resources中的资源路径）

注意：这里只是加载，不提供实例化。

```
public T LoadRes<T>(string resName);
```

- resName: 需要加载的资源路径（比如："Prefab/Bullet/zidan"）
- T: 需要加载的资源的类型（预制体一般是GameObject）

```
public class Test: MonoBehaviour
{
    private void Start()
    {
        GameObject obj = ResManager._Instance.LoadRes<GameObject>
("Prefab/Bullet/zidan");
         Instantiate(obj);
    }
}
```

应用2：异步加载资源（位于Resources中的资源路径）

注意：这里只是加载，不提供实例化。

```
public void LoadResAsync<T>(string resName,UnityAction<T> doSoming );
```

- resName: 同上，需要加载的资源路径
- doSoming: 含参的委托，其实是一个回调（这里用callBack比较准确）。异步加载成功后的这个资源，需要干些什么。
- T: 需要加载的资源的类型

```
public class Test: MonoBehaviour
{
    private void Start()
    {
        ResManager._Instance.LoadResAsync<GameObject>("Prefab/Bullet/zidan",
(obj) =>
        {
            Instantiate(obj);
        });
    }
}
```