Unity中的EventSystem

- Unity中的EventSystem除了处理UI的，也可以处理自定义的事件
- EventSystem的主要功能是处理输入，根据射线寻找对应的响应对象
- EventSystem，也可以向场景中的非UI物体传递事件消息，但是基于输入，通过射线实现
  - 但是，为什么不直接挂载InputSystem的某个动作的回调呢？（比如OnTriggerEnter时，自动enable交互动作，挂载回调）
  - 小结：使用射线的事件不靠谱
- 存在不依赖射线的事件系统（基于接口的事件系统）
  - ```
    ExecuteEvents.Execute<ICustomMessageTarget>(target, null, (x,y)=>x.Message1());
    ```
  - 但是需要获取到具体的执行回调的Gameobject，然后再找到具体的执行回调的脚本，感觉不如自己写一套事件系统
  - EventSystem本身在文档中就是处于UI模块下的一个系统
- 总结
  - Unity的EventSystem只是为了UI服务，不适合用来做通用的事件系统

事件分类

- 全局事件和局部事件
- 全局事件
  - 扩散范围较广的事件（涉及多个系统，例如游戏开始）
  - 或者是不能明确事件的具体发起对象（比如成就系统的某项成就数据依赖于多个渠道的数据总和，成就系统不可能依次获取到每个可能发出事件的对象然后监听事件）
- 局部事件
  - 依赖于某个局部对象，比如血条；
  - 又或者是一个子系统内部的通信，在一个系统内部，父级可以直接获取子级组件，子级向父级通信使用事件（能合理持有某个对象的话，可以使用局部事件，是否合理取决于系统如何分层）

自定义的全局事件系统

- 单例类，使用字典存储事件和对应的回调，提供 注册，取消注册，发送事件的功能
- 字典存储事件标识符和事件响应者列表，事件标识符可以是int，enum，Type
  - 基于枚举的事件系统可以看下例，使用枚举比使用int或string更能减少出错概率
    - Unity游戏开发—基于枚举的事件系统 - 知乎 (zhihu.com)
    - 查找效率是高了，但是传递参数比较麻烦
    - 需要解决枚举作为字典的性能问题
  - 基于类型的事件系统可以看QFramework
    - QFramework中的做法
      - 每个事件是一个类，这个类既是事件的标识符，又表示事件的参数
      - 使用时可以传入类实例也可以不传，不传入的话就是默认构造一个类实例作为参数（一般无参事件就不用传，可以选择将事件类设为struct）
      - 传入类实例的话可以在构造类实例时自定义参数，有参事件就是这么用（此时的事件最好设定为class，方便作为引用类型传递）
      - 注意值并不是多个委托的列表，而是一个委托，每种事件仅对应一个委托，因为一个委托可以添加多个回调所以不用列表
      - 所以字典的值可以设置为一个接口（也可以不设置），注册事件时仅对这一个接口实例（字典的值是接口实例）中的委托实例操作
  - 基于接口的事件（灵感来源于Unity的EventSystem）
    - 属于基于类型的事件系统的一种，字典的键是接口类型，值是实现了接口的类实例
    ```

        - 注册时，需要监听某个事件的类必须先继承该事件的接口，然后向全局的事件系统注册自身，需要提供接口类型，注册仍然是泛型函数
        - 取消注册，同注册
        - 触发事件，传递一个委托实例（参数类型为自定义的接口），事件系统将字典中存储的类实例作为参数供给传进来的委托实例使用
        - 好处，通过接口规范了事件接收者的实现方式，事件的参数仍然是由各个类实例决定，事件发起者可以在每个监听者执行方法的前后添加一些自定义操作，事件发起者可以知道有多少类监听
        - 缺点：事件调用比较麻烦，需要调用者自己多定义一层实现的内容，感觉空间消耗略微增加，原本只需要一个委托实例作为回调，现在需要存储监听者列表
        - 其他的事件系统事件发出后就不再归发送者管理【我被偷听了，我不知道他们会干什么】，而基于接口的事件系统的思想是【你想监听我，好，那你准备好在这（字典里）等着，我会来调用你】
        - 基于接口的事件系统较为复杂，对分层有一定帮助，可以满足需要精确控制回调的特殊事件
    - 需要注意因为存储的是类实例的引用（与前面两种方式不同），为了保证不影响类被垃圾回收，需要使用弱引用，注册时创建新的弱引用并缓存弱引用，取消注册时需要根据标识符找到对应类实例的弱引用
        - 实现参考
            - 【Unity教程搬运】事件总线(Event Bus)_inspironx的博客-CSDN博客
            - 需要注意上面的教程有一个误区，即hashCode是会重复的，如果使用hashCode作为查找弱引用的键，可能导致出错。
- 如何处理回调的参数问题
    - 如果事件标识符类型，参数可以设为该类，可以保证事件事件标识符与参数一致
    - 如果事件标识符是其他类型数据，那么事件参数只能设置为泛型类型或者是封装了object数组的自定义类
        - 如果设置为封装了object数组的自定义类，那么所有事件的参数都是该类类型，但监听者无法知晓具体会有几个参数以及这些参数分别是什么
        - 如果设置为泛型类型，回调为一个委托类型，那么委托实例每次添加监听者都需要对比新添加的回调的委托类型是否与之前的一致，同样不能知道具体的委托参数信息
    - 小结：对于有参数的事件，使用基于类型的事件系统是最好的，对于不需要参数的事件使用基于枚举的事件系统

事件总线

> 事件总线（Event Bus）作为中心枢纽管理着一系列可以订阅或发布的事件。它是一种集中式事件处理机制，允许不同的组件之间进行彼此通信而又不需要相互依赖，达到一种解耦的目的。
>
> 因此，事件总线是一种使用发布-订阅模式（publish-subscribe）通过事件连接对象的方法。
>
> 事件总线（Even Bus）模式本质上是发布订阅模式（Publish-Subscribe）。当对象即发布者（Publisher）引发事件时，它给其他对象发送信号。只有订阅该事件的对象即订阅者（Subscriber）才会被通知，并处理该事件。

- 上面介绍的三种类型的自定义事件系统都是事件总线

事件队列

- 异步的观察者模式
- 【游戏编程模式】事件队列模式 - 知乎 (zhihu.com)
- 感觉使用场景有限，上面的文章的做法就是将需要处理的东西集中到一次Update中进行，对于音频类处理可能有点帮助
- 但是游戏中大部分情景都需要及时的反馈，分时的效益不大，而且事件队列主要是防止有性能瓶颈，目前没有实际需求