回顾SOLID

- S：单一职责原则
- O：开闭原则，表示后续的扩展不能修改原有代码
- L：替换原则，即基类的所有方法子类都应该能用
- I：接口分离原则，即接口不应该掺杂太多东西
- D：依赖倒置原则，高层不依赖低层，两者都依赖抽象（最难理解，个人认为最重要）
- 好莱坞原则：我来找你（高层找低层，但需要通过接口保证高层不依赖低层），你别来找我（低层不能主动去获取高层）

解耦

- static
  - 一种现象，无论是哪种机制，都存在static帮助解耦，这是无法避免的，可以做的是避免处处使用static，导致代码完全没有结构可言
    - SO其实就是一种static的数据，可以全局获取
    - Singleton的全局可访问性也是通过设置静态实例实现
    - IoC容器依赖于静态实例获取
- MVVM
  - ViewModel被多个对象依赖，其在js中的实现是继承链
  - SO类似于ViewModel，比较明显的不同之处是两者获取的方式不一样
- IoC (Inversion of Control)
  - 通过反射获取到类实例
  - 目的是不让被依赖的对象由依赖者直接创建，否则会引发混乱；另外的目的是实现多态（策略模式的体现）
  - 简单的IoC自己就能实现，可以参考QFramework框架
  - 这个网站（Seba's Lab (sebaslab.com)）上的一系列博客介绍了IoC（2012年），随后又阐述了IoC的问题，认为IoC不适合用在电子游戏中（2015年），最后阐述了ECS的解决办法
- 依赖注入
  - 依赖注入和IoC概念其实类似，都是为了解除依赖对象和被依赖对象之间的紧耦合
  - 只是IoC通常和IoC容器概念相关联，而依赖注入更像一个宽泛的概念，个人认为SO也是实现依赖注入的一种手段
  - 依赖注入有比较成熟的框架，例如Zenject（Fork自上面认为IoC不适用于电子游戏的作者的IoC仓库）
- 事件
  - 事件代表着观察者模式
  - 个人划分：事件分为依赖某个具体对象的事件（血条值变化事件）和不依赖具体对象的事件（游戏开始事件）
  - 事件总线机制实现的核心方法就是通过设置全局字典，其他类通过事件总线 注册/取消注册/发送 事件。具体看另一篇事件分析的文章
- 单例
  - 设置全局静态实例，忽视了分层的概念
  - 单例模式最大的缺陷不是违背单一职责（单例的静态实例仅存储数据就可以像SO一样），个人认为最大的缺陷是忽视了分层，所有地方都能访问单例，代码会变得混乱
  - 从解耦的原理讲，类似单例的静态结构是必须存在的，但需要通过接口和其他解耦手段将逻辑分层
- SO
  - 两个系统通过依赖同一个SO资产连接，形成松耦合
  - 涉及非代码层面，容易忽视分层概念，整体和静态单例区别不大，编辑器内可视是突出优点，需要通过面板指定也是一个缺陷
- 接口
  - 主要作用是提供抽象，另外一个重要用途是将程序分层（参考QFramework）
  - 将程序分层可以实现依赖倒置，有助于实现单一职责
- 中介者
  - Unity应用架构设计(2)—使用中介者模式解耦ViewModel之间通信 - 木宛城主 - 博客园 (cnblogs.com)
- GameObject.Find/GetComponent
  - 依赖对象虽然不是由被依赖者管理（生成），但获取依赖对象的方式依赖于某种不变的关系（比如Hierarchy中的层级关系）
  - 如果对场景中的物体使用这种方式获取依赖是不稳定的
  - GameObject.Find/GetComponent只应该在局部内使用，比如一个UI的prefab内部。

- 获得的是整个依赖对象，如果没有分层的概念，代码结构会很乱，可以添加接口，GetComponent指定获取接口
- Unity中Find问题的本质，再到IOC与ECS架构 - 知乎 (zhihu.com)