设计模式

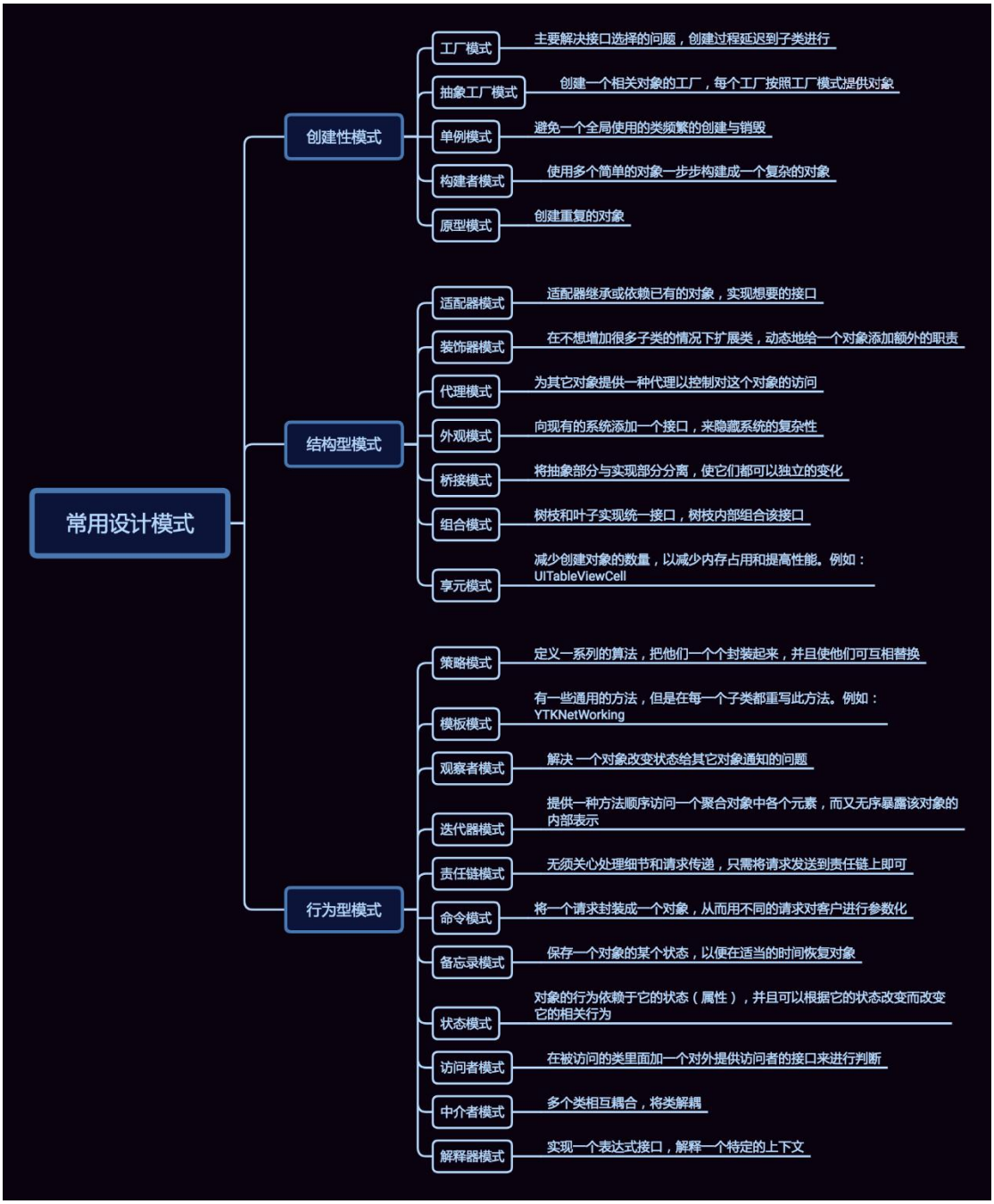

图 1.　　设计模式导图

一.　单例模式

提供一个访问该类的全局访问点，并保证该类仅有一个实例。单例模式不仅可以保证实例的唯一性，还可以严格的控制外部对实例的访问。

1. Unity 泛型单例

1) 泛型参数的约束

在定义泛型类时，可以对客户端代码能够在实例化类时用于类型参数的类型种类施加限制。如果客户端代码尝试使用某个约束所不允许的类型来实例化类，则会产生编译时错误。

这些限制称为约束。约束是使用 where 上下文关键字指定的。下表列出了六种类型的约束：

where T: struct

类型参数必须是值类型。可以指定除 Nullable 以外的任何值类型。有关更多信息，请参见使用可以为 null 的类型（C# 编程指南）。

where T : class

类型参数必须是引用类型；这一点也适用于任何类、接口、委托或数组类型。

where T：new()

类型参数必须具有无参数的公共构造函数。当与其他约束一起使用时，new() 约束必须最后指定。

where T：<基类名>

类型参数必须是指定的基类或派生自指定的基类。

where T：<接口名称>

类型参数必须是指定的接口或实现指定的接口。可以指定多个接口约束。约束接口也可以是泛型的。

where T：U

为 T 提供的类型参数必须是为 U 提供的参数或派生自为 U 提供的参数。

2) 完整代码

PS：

```
using UnityEngine;

public class SingleComponent<T> : MonoBehaviour where T : SingleComponent<T>
{
    //设置为私有字段的变量，在派生类中是无法访问的
    private static T instance;
    public static T Instance//[单例模式]通过关键字 Static 和其他语句之生成对象的一个实例，确保该脚本在场景中的唯一性
    {
        get
        {
            return instance;
        }
    }

    //关键字 Virtual，使此函数变为一个虚函数，让其可以在一个或多个派生类中被重新定义。
    protected virtual void Awake()
    {
        if (instance != null)//场景中的实例不唯一时
        {
            Destroy(instance.gameObject);
        }
        else
        {
            instance = this as T;
```

```
            }
        }
        //设置为 protected 类型的变量，派生类可以访问，非派生类无法直接访问
        protected virtual void OnDestroy()
        {
            Destroy(instance.gameObject);
        }

        //如果遇到报错：Some objects were not cleaned up when closing the scene. (Did you spawn new GameObjects from OnDestroy?)
        //请在 OnDestory 调用单例的地方使用这个判断一下
        /// <summary>
        ///  单例是否已初始化
        /// </summary>
        public static bool IsInitialized
        {
            get
            {
                return instance != null;//返回的是一个 bool 值，即该实例是否为空
            }
        }
    }
```

二. 观察者模式

观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象，这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己的行为.

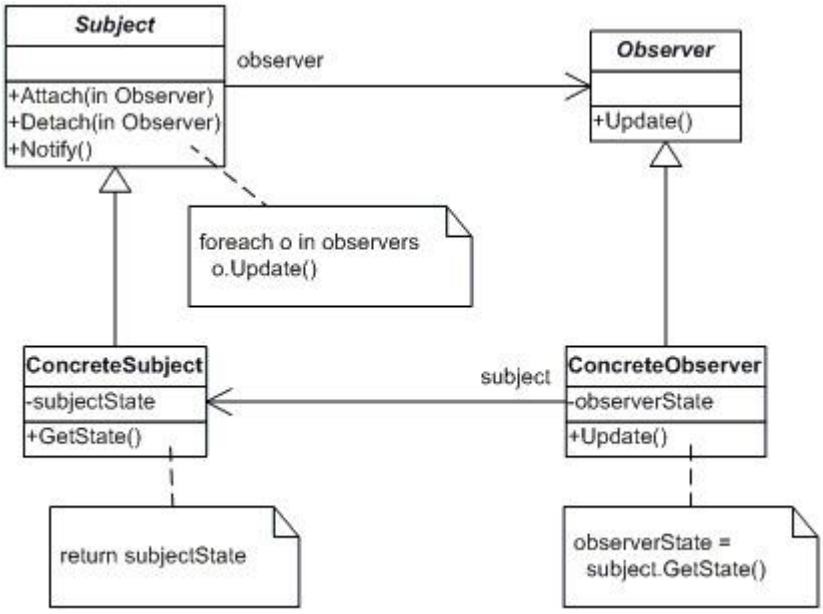

图 2. 图 2.1 观察者模式结构图

在 Unity 中，观察者模式的实现是基于 C#的事件委托机制，要彻底的解除通知者和观察者之间的耦合就可以采用事件委托（Delegate）。委托就是一种引用方法的类型，拥有参数和返回值，作为一种方法的抽象，是特殊的“函数类”，一旦我们为委托分配了方法，委托

将会执行与该方法相同的行为，其实例代表了一个或多个具体的函数。C#中关于回调函数的实现机制便是委托，事件（Event）是基于委托的，事件为委托提供了一种用于实现类之间消息发布与接收的结构。在 C#中订阅与取消事件可以使用“+=”与“-=”，在 Unity 中可以通过添加 Handler 来监听事件消息，进而实现消息响应，可以方便的创建事件与编写响应，通过创建 Handle 来监听事件进而完成游戏逻辑。事件只能被其他的模块监听注册，但是不能由其他的模块触发。要触发事件，只能由模块内的公开方法在其内部进行事件触发，这样可以使得程序更加安全。但是在委托的使用过程中会出现过于依赖反射机制（reflection）来查找消息对应的被调用函数，这会影响部分性能。

1. 优点

第一、观察者模式在被观察者和观察者之间建立一个抽象的耦合。被观察者角色所知道的只是一个具体观察者列表，每一个具体观察者都符合一个抽象观察者的接口。被观察者并不认识任何一个具体观察者，它只知道它们都有一个共同的接口。

由于被观察者和观察者没有紧密地耦合在一起，因此它们可以属于不同的抽象化层次。如果被观察者和观察者都被扔到一起，那么这个对象必然跨越抽象化和具体化层次。

第二、观察者模式支持广播通讯。被观察者会向所有的登记过的观察者发出通知，

观察者模式可以实现表示层和数据逻辑层的分离，并定义了稳定的消息更新传递机制，抽象了更新接口，使得可以有各种各样不同的表示层作为具体观察者角色。

观察者模式在观察目标和观察者之间建立一个抽象的耦合。

观察者模式支持广播通信。

观察者模式符合“开闭原则”的要求。

2. 缺点

第一、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

第二、如果在被观察者之间有循环依赖的话，被观察者会触发它们之间进行循环调用，导致系统崩溃。在使用观察者模式是要特别注意这一点。

第三、如果对观察者的通知是通过另外的线程进行异步投递的话，系统必须保证投递是以自恰的方式进行的。

第四、虽然观察者模式可以随时使观察者知道所观察的对象发生了变化，但是观察者模式没有相应的机制使观察者知道所观察的对象是怎么发生变化的。

如果一个观察目标对象有很多直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。

观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

3. 模式扩展

MVC 模式是一种架构模式，它包含三个角色：模型(Model)，视图(View)和控制器(Controller)。观察者模式可以用来实现 MVC 模式，观察者模式中的观察目标就是 MVC 模式中的模型(Model)，而观察者就是 MVC 中的视图(View)，控制器(Controller)充当两者之间的中介者(Mediator)。当模型层的数据发生改变时，视图层将自动改变其显示内容。

三. 懒汉模式

1. 解决初始化与脚本加载顺序的矛盾

四. UI 框架

1. MVC

M：Model，UI 的数据以及对于 UI 数据的操作函数。数据部分可以理解为 UI 组件的具体渲染数据。以 UGUI 为例，一个 UI 上挂载了一个 Text 组件，一个 Button 组件，那么 Model 里可能就会有一个 string 属性代表 Text 显示的内容，一个 Action 属性表示 Button 被点击时会调用的回调函数。逻辑部分就是我们在游戏逻辑中会对 UI 数据进行的操作，可能还会有一些复杂的变换操作，比如一个人物最大生命值是 100，并且血条本身只显示血量百分比，那么这个 Model 里的 SetHPBarValue()函数就会执行下面的操作，把传进来的当前人物生命值（这里假设是 50）换算到一个百分比的值，即：当前人物生命值/人物最大生命值 ＝0.5。然后会被 Controller 调用，继而把这个值给 View，让它显示 50%。

V：View，UI 的渲染组件，以 UGUI 为例，一个 UI 上挂载了一个 Text，一个 Button，那么这两个 UI 组件都将写到 View 里。也会包含一些操作，例如获取一个 UI 组件，对一个 UI 组件进行操作。

C：Controller，本身 Model 和 View 他们是互不耦合的，所以需要一个中介者把他们联系起来，从而达到数据更新和渲染状态更新同步。它本身是没什么代码量的，代码几乎都在 Model 和 View 中。

然后就是很多人说 MVC 会有很多冗余代码，写起来不方便，应该是他们的使用方法有问题，可能并没有搞清楚 M、V、C 的分工，可能把 M 的工作写到了 C 里，或者把 C 里的工作写到了 V 里，总之概念就搞的一团糟，更不用说写出的代码如何了。

2. MVVM

上面我已经提到了 MVVM，下面来详细介绍一下 MVVM 到底是个什么东西。

M：Model，还是和 MVC 的 M 差不多。

V：View，还是和 MVC 的 V 差不多。

VM：双向耦合（可能这里叫双向绑定更加贴切一点）M 和 V，什么是绑定呢，这里用观察者模式来解释可能更好懂一些。以 M 绑定 V 为例，M 的更新会执行一个委托，而我们此时已经将 V 加入到委托链里了，所以 M 的更新会执行我们指定的一个委托函数，从而达到 V 也更新的目的。V 绑定 M 也是同理。最后，M 更新会导致 V 更新，V 更新也会导致 M 相关数据更新的效果（避免循环应该加一个判重机制）。

我感觉 MVVM 和 MVC 并没有本质上的区别，他只是更加强制制定了更加明显的规范，所以写起来感觉 MVVM 更加厉害一些。

架构

一. OOP 思想

1. 什么是 OO

OO(Object－Oriented)面向对象，OO 方法(Object-Oriented Method，面向对象方法，面向对象的方法)是一种把面向对象的思想应用于软件开发过程中，指导开发活动的系统方法，简称 OO (Object-Oriented)方法，Object Oriented 是建立在“对象”概念基础上的方法学。对象是由数据和容许的操作组成的封装体，与客观实体有直接对应关系，一个对象类定义了具有相似性质的一组对象。而继承性是对具有层次关系的类的属性和操作进行共享的一种方式。所谓面向对象就是基于对象概念，以对象为中心，以类和继承为构造机制，来认识、理解、

刻画客观世界和设计、构建相应的软件系统。

2. 核心思想

封装（Encapsulation）、继承（Inheritance）、多态（Polymorphism）。

二. ECS 架构

在游戏项目开发的过程中，一般会使用 OOP 的设计方式让 GameObject 处理自身的业务，然后框架去管理 GameObject 的集合。但是使用 OOP 的思想进行框架设计的难点在于一开始就要构建出一个清晰类层次结构。而且在开发过程中需要改动类层次结构的可能性非常大，越到开发后期对类层次结构的改动就会越困难。

经过一段时间的开发，总会在某个时间点开始引入多重继承。实现一个又可工作、又易理解、又易维护的多重继承类层次结构的难度通常超过其收益。因此多数游戏工作室禁止或严格限制在类层次结构中使用多重继承。若非要使用多重继承，要求一个类只能多重继承一些 简单且无父类的类

也就是说在大型游戏项目中，OOP 并不适用于框架设计。但是也不用完全抛弃 OOP，只是在很大程度上，代码中的类不再具体地对应现实世界中的具体物件，ECS 中类的语义变得更加抽象了。

ECS 有一个很重要的思想：数据都放在一边，需要的时候就去用，不需要的时候不要动。ECS 的本质就是数据和操作分离。传统 OOP 思想常常会面临一种情况，A 打了 B，那么到底是 A 主动打了 B 还是 B 被 A 打了，这个函数该放在哪里。但是 ECS 不用纠结这个问题，数据存放到 Component 种，逻辑直接由 System 接管。借着这个思想，我们可以大幅度减少函数调用的层次，进而缩短数据流传递的深度。

1. 何为 ECS 架构

ECS，即 Entity-Component-System（实体-组件-系统）的缩写，Entity 由多个 Component 组成，Component 由数据组成，System 由逻辑组成。其模式遵循组合优于继承原则，游戏内的每一个基本单元都是一个实体，每个实体又由一个或多个组件构成，每个组件仅仅包含代表其特性的数据（即在组件中没有任何方法），例如：移动相关的组件 MoveComponent 包含速度、位置、朝向等属性，一旦一个实体拥有了 MoveComponent 组件便可以认为它拥有了移动的能力，系统(System)便是来处理拥有一个或多个相同组件的实体集合的工具，其只拥有行为（即在系统中没有任何数据），在这个例子中，处理移动的系统仅仅关心拥有移动能力的实体，它会遍历所有拥有 MoveComponent 组件的实体，并根据相关的数据（速度、位置、朝向等），更新实体的位置。

实体与组件是一个一对多的关系，实体拥有怎样的能力，完全取决于其拥有哪些组件，通过动态添加或删除组件，可以在（游戏）运行时改变实体的行为。

2. 实体、组件与系统

先有一个 World，它是系统和实体的集合，而实体就是一个 ID，这个 ID 对应了组件的集合。组件用来存储游戏状态并且没有任何行为，系统拥有处理实体的行为但是没有状态。

1) 实体—Entity

实体只是一个概念上的定义，指的是存在你游戏世界中的一个独特物体，是一系列组件的集合。为了方便区分不同的实体，在代码层面上一般用一个 ID 来进行表示。所有组成这个实体的组件将会被这个 ID 标记，从而明确哪些组件属于该实体。由于其是一系列组件的集合，因此完全可以在运行时动态地为实体增加一个新的组件或是将组件从实体中移除。比如，玩家实体因为某些原因（可能陷入昏迷）而丧失了移动能力，只需简单地将移动组件从该实体身上移除，便可以达到无法移动的效果了。

Entity 需要遵循立即创建和延迟销毁原则，销毁放在帧末执行。因为可能会出现这样的

情况：systemA 提出要在 entityA 所在位置创建一个特效，然后 systemB 认为需要销毁 entityA。如果 systemB 直接销毁了 entityA，那么稍后 FxSystem 就会拿不到 entityA 的位置导致特效播放失败（你可能会问为什么不直接把 entityA 的位置记录下来，这样就不会有问题了。这里只是简单举个例子，不要太深究(●'◡'●)）。理想的表现效果应该是，播放特效后消失。

PS：

Player(Position, Sprite, Velocity, Health)

Enemy(Position, Sprite, Velocity, Health, AI)

Tree(Position, Sprite)

注：括号前为实体名，括号内为该实体拥有的组件。

2) 组件—Component（只有变量，没有函数）

一个组件是一堆数据的集合，可以使用 C 语言中的结构体来进行实现。它没有方法，即不存在任何的行为，只用来存储状态。（Component 之间不可以直接通信）一个经典的实现是：每一个组件都继承（或实现）同一个基类（或接口），通过这样的方法，我们能够非常方便地在运行时动态添加、识别、移除组件。每一个组件的意义在于描述实体的某一个特性。例如，PositionComponent（位置组件），其拥有 x、y 两个数据，用来描述实体的位置信息，拥有 PositionComponent 的实体便可以说在游戏世界中拥有了一席之地。当组件们单独存在的时候，实际上是没有什么意义的，但是当多个组件通过系统的方式组织在一起，才能发挥出真正的力量。同时，我们还可以用空组件（不含任何数据的组件）对实体进行标记，从而在运行时动态地识别它。如，EnemyComponent 这个组件可以不含有任何数据，只是拥有该组件的实体会被标记为“敌人”。

根据实际开发需求，这里还会存在一种特殊的组件，名为 Singleton Component （单例组件），顾名思义，单例组件在一个上下文中有且只有一个。

PS：

PositionComponent(x, y)

VelocityComponent(X, y)

HealthComponent(value)

PlayerComponent()

EnemyComponent()

注：括号前为组件名，括号内为该组件拥有的数据

3) 系统—System

理解了实体和组件便会发现，到这里我们还未曾提到过游戏逻辑相关的话题。系统便是 ECS 架构中用来处理游戏逻辑的部分。何为系统，一个系统就是对拥有一个或多个相同组件的实体集合进行操作的工具，它只有行为，没有状态，即不应该存放任何数据。举个例子，游戏中玩家要操作对应的角色进行移动，由上面两部分可知，角色是一个实体，其拥有位置和速度组件，那么怎么根据实体拥有的速度去刷新其位置呢，MoveSystem（移动系统）登场，它可以得到所有拥有位置和速度组件的实体集合，遍历这个集合，根据每一个实体拥有的速度值和物理引擎去计算该实体应该所处的位置，并刷新该实体位置组件的值，至此，完成了玩家操控的角色移动了。

System 之间的执行顺序需要严格制定，且 System 之间不可以直接通信。

注意，我强调了移动系统可以得到所有拥有位置和速度组件的实体集合，因为一个实体同时拥有位置和速度组件，我们便认为该实体拥有移动的能力，因此移动系统可以去刷新每一个符合要求的实体的位置。这样做的好处在于，当我们玩家操控的角色因为某种原因不能移动时，我们只需要将速度组件从该实体中移除，移动系统就得不到角色的引用了，同样的，如果我们希望游戏场景中的某一个物件动起来，只需要为其添加一个速度组件就万事大吉。

一个系统关心实体拥有哪些组件是由我们决定的，通过一些手段，我们可以在系统中很快地得到对应实体的集合。

前面提到过的 Singleton Component（单例组件），明白了系统的概念更容易说明，还是玩家操作角色的例子，该实体速度组件的值从何而来，一般情况下是根据玩家的操作输入去赋予对应的数值。这里就涉及到一个新组件 InputComponent（输入组件）和一个新系统 ChangePlayerVelocitySystem（改变玩家速度系统），改变玩家速度系统会根据输入组件的值去改变玩家速度，假设还有一个系统 FireSystem（开火系统），它会根据玩家是否输入开火键进行开火操作，那么就有 2 个系统同时依赖输入组件，真实游戏情况可能比这还要复杂，有无数个系统都要依赖于输入组件，同时拥有输入组件的实体在游戏中仅仅需要有一个，每帧去刷新它的值就可以了，这时很容易让人想到单例模式（便捷地访问、只有一个引用），同样的，单例组件也是指整个游戏世界中有且只有一个实体拥有该组件，并且希望各系统能够便捷的访问到它，经过一些处理，在任何系统中都能通过类似 world->GetSingletonInput() 的方法来获得该组件引用。

系统这里比较麻烦，还存在一个常见问题：由于代码逻辑分布于各个系统中，各个系统之间为了解耦又不能互相访问，那么如果有多个系统希望运行同样的逻辑，该如何解决，总不能把代码复制 N 份，放到各个系统之中。UtilityFunction（实用函数） 便是用来解决这一问题的，它将被多个系统调用的方法单独提取出来，放到统一的地方，各个系统通过 UtilityFunction 调用想执行的方法，同系统一样， UtilityFunction 中不能存放状态，它应该是拥有各个方法的纯净集合。

PS：

MoveSystem(Position, Velocity)

RenderSystem(Position, Sprite)

注：括号前为系统名，括号内为该系统关心的组件集合