设计模式

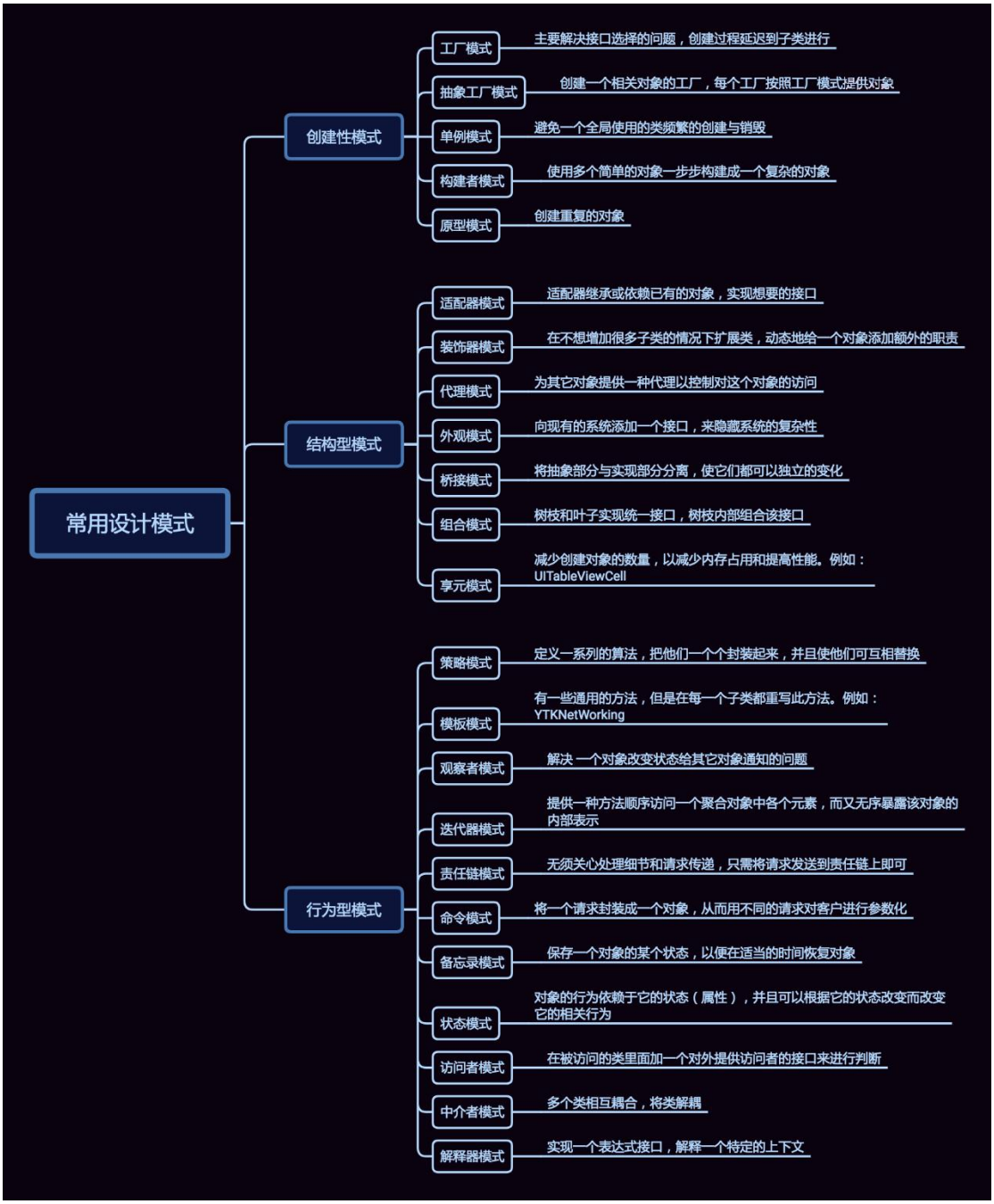

图 1.　　设计模式导图

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.

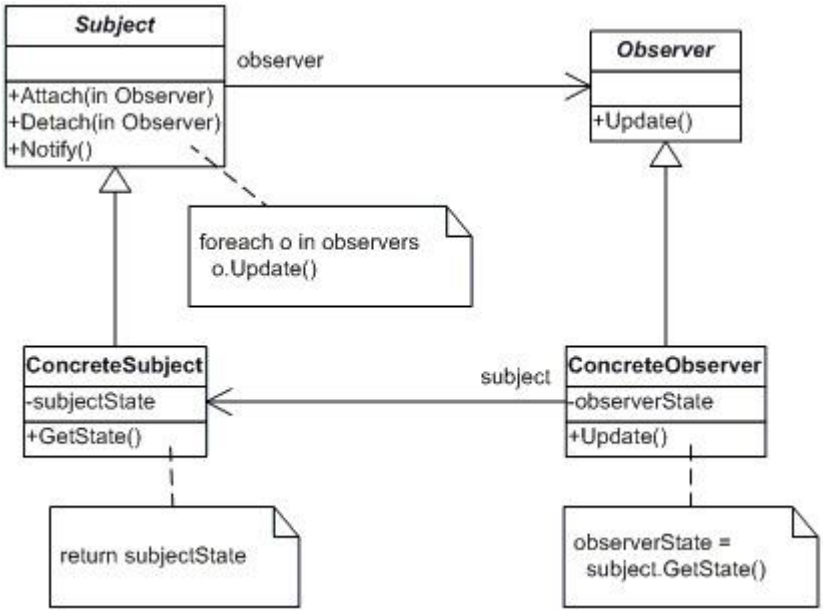

图 2. 图 2.1 观察者模式结构图

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。

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。