单例设计模式

适用于项目中某些只允许被实例化一次的对象

例如：玩家信息类 游戏管理类 资源加载类等等

程序设计中最常用的设计模式

很多设计模式都是在单例的基础上进行设计

核心结构中只包含一个成为单例的特殊类

通过单例模式保证系统运行过程中 这个类只有一个对象实例 方便外部调用 实现数据统一

1 C#单例写法

```
using UnityEngine;
using System.Collections;

public class Game{

    private static Game instance;
    public static Game Instance
    {
        get {
            if (instance == null)
                instance = new Game();
            return instance;
        }
    }

    private Game() { }
}
```

2 C#抽象单例模板

适用于项目较大 比较多的单例类的时候使用

```
using UnityEngine;
using System.Collections;
```

```
3
4  //单例模板  抽象父类
5  public abstract class Singleton<T> where T:new(){
6
7      private static T _instance;
8      public static T _Instance
9      {
10         get{
11             if (_instance == null)
12                 _instance = new T();
13             return _instance;
14         }
15     }
16 }
17
18 //子类直接继承 并传递类型 子类直接成为单例类
19 public class Game:Singleton<Game>
20 {
21 }
22 public class Game2 : Singleton<Game2>
23 {
24 }
```

3 Unity中 继MonoBehaviour的单例写法

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class BackGround : MonoBehaviour {
5
6      private static BackGround instance;
7      public static BackGround Instance
8      {
9          get { return instance; }
10     }
11
12     void Awake () {
13         instance = this;
14         DontDestroyOnLoad(gameObject);
15     }
```

```
}

```

4 Unity中单例父类模板

```
using UnityEngine;
using System.Collections;

//1 保证场景中默认没有单例对象 当调用单例时自动创建单例对象
//2 场景中不能出现2个或以上的单例对象
//3 设置报错的机制 当出现超过1个对象时 提示错误
public class UnitySingleton<T> : MonoBehaviour where T:Component {

    private static T _instance;
    public static T _Instance
    {
        get {
            if (_instance == null)
            {
                _instance=FindObjectOfType<T>();
                if (FindObjectsOfType<T>().Length > 1)
                {
                    Debug.LogError("Instance Count >1");
                }

                if (_instance == null)
                {
                    GameObject obj = new GameObject();
                    obj.name = typeof(T).ToString();
                    _instance=(T)obj.AddComponent(typeof(T));
                }

            }
            return _instance;
        }
    }

    public virtual void Awake()
    {
```

```
        DontDestroyOnLoad(gameObject);
        if (_instance == null)
            _instance = this as T;
        else
            Destroy(gameObject);
    }

}

//继承 子类变为单例类
public class AudioManager :UnitySingleton<AudioManager> {
}

```

工厂模式

工厂模式的目的是用来创建不同类型的对象

简单工厂

```
using UnityEngine;
using System.Collections;

//形状工厂的父类
public class ShapeFactory{

    public enum ShapeType
    {
        Cube,
        Sphere,
        Capsule
    }

    public virtual GameObject CreateShape()
    {
        return null;
    }
}
```

```
19
20 public class CubeFactory : ShapeFactory
21 {
22     public override GameObject CreateShape()
23     {
24         return GameObject.CreatePrimitive(PrimitiveType.Cube);
25     }
26 }
27
28 public class SphereFactory : ShapeFactory
29 {
30     public override GameObject CreateShape()
31     {
32         return GameObject.CreatePrimitive(PrimitiveType.Sphere);
33     }
34 }
35 public class CapsuleFactory : ShapeFactory
36 {
37     public override GameObject CreateShape()
38     {
39         return GameObject.CreatePrimitive(PrimitiveType.Capsule);
40     }
41 }
42
43 public class ShapeProducer:Singleton<ShapeProducer>
44 {
45     public GameObject CreateShape(ShapeFactory.ShapeType type)
46     {
47         switch (type)
48         {
49             case ShapeFactory.ShapeType.Cube:
50                 return new CubeFactory().CreateShape();
51             case ShapeFactory.ShapeType.Sphere:
52                 return new SphereFactory().CreateShape();
53             case ShapeFactory.ShapeType.Capsule:
54                 return new CapsuleFactory().CreateShape();
55             default:
56                 return null;
57         }
58     }
59 }
60
```

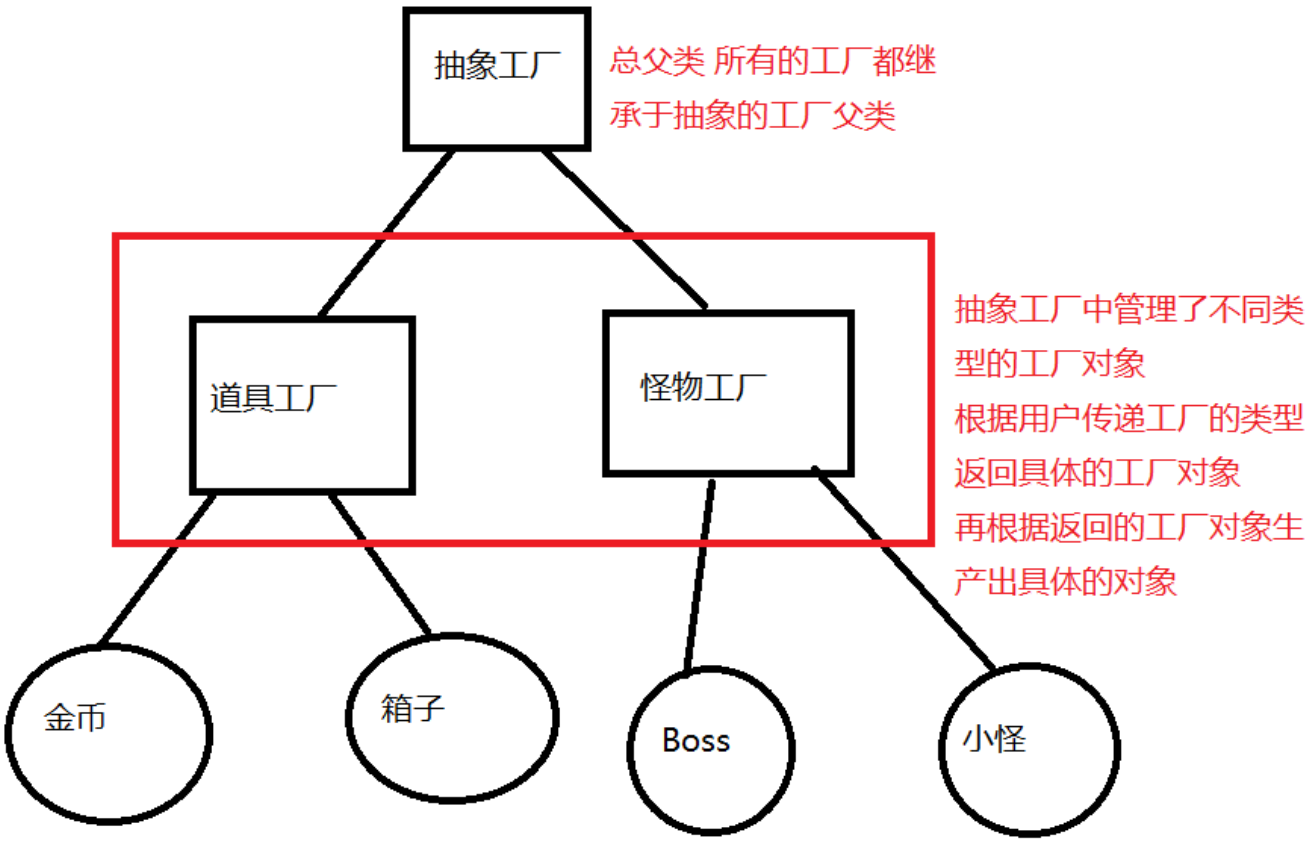
```
61
62
63 void Start () {
64    ShapeProducer._Instance.CreateShape(ShapeFactory.ShapeType.Cube);
65    ShapeProducer._Instance.CreateShape(ShapeFactory.ShapeType.Sphere);
66
   ShapeProducer._Instance.CreateShape(ShapeFactory.ShapeType.Capsule);
67 }
68
69
```

抽象工厂（超级工厂）

工厂的工厂

根据工厂的类型得到具体的工厂

所有的工厂继承于抽象的工厂父类



观察者模式

设定一个观察者对象来观察事件的发生 所有需要响应该事件的对象都要把自己的回调注册到观察者

当事件发生 观察者会通知所有注册了事件的对象 来调用对应回调函数

例 NGUI Button OnClick事件 一对多的依赖关系

定义鼠标事件的委托 再事件管理类中增加Event

添加事件注册 事件移除 触发事件的方法

```
using UnityEngine;
using System.Collections;

public delegate void OnMouseButtonDown();
public class ButtonEventManager:Singleton<ButtonEventManager>{

    public event OnMouseButtonDown ButtonDown;

    //添加事件
    public void AddBtnEvent(OnMouseButtonDown btnDown)
    {
        ButtonDown += btnDown;
    }
    //移除事件
    public void RemoveBtnEvent(OnMouseButtonDown btnDown)
    {
        ButtonDown -= btnDown;
    }

    //触发事件
    public void EventHandler()
    {
        if (ButtonDown != null)
            ButtonDown();
    }
}
```

只要添加事件函数 当鼠标点下时 物体都可以以回调的形式得到通知

C# L1 (default)

```
public class Cube111 : MonoBehaviour {

    void OnEnable () {
        ButtonEventManager._Instance.AddBtnEvent(Mouse);
    }
    void Mouse () {
        Debug.Log("我是Cube111 按钮按下了");
    }
}
```

事件触发

```
public class Txt : MonoBehaviour {


    void Start () {

    }

    // Update is called once per frame
    void Update () {
        if (Input.GetMouseButtonDown(0))
        {
            ButtonEventManager._Instance.EventHandler();
        }
    }
}

```

状态模式

例如人物状态机的实现 行为型的设计模式

1 封装转换规则

2 将所有玉某个状态相关的行为放到一个类中 增加新状态简单化 只需要改变状态对象即可改变对象行为

3 允许状态转换逻辑与状态对象合成一体 而不是通过传统的条件语句块控制

4 可以让多个环境对象共享一个状态对象 从而减少系统中对象的个数