MVC模块

M（Model）数据层 用途：1 保存数据 2 发送数据更新信息

V（View）视图层 用途：1 接受用户从界面上的操作 2 根据M层的数据显示相关界面

C (Control) 控制层 用途 1 处理和界面无关的代码逻辑 2 接受和处理网络数据

模型结构：

用户点击->UI响应->调用M更改数据->发送更新界面信息->V接受消息->更新界面

管理者模式

ConfigDataBase 配置文件管理类

SysUIEnv　UI管理类

SysPlayerPrefs 存档类

SysAudioManager 音效管理类

SysSceneManager 场景管理类

这些管理类本身都是基于单例实现的管理者模式结构

但是项目中应该把这些系统模块也进行统一的管理

这样整个项目的轮廓和框架也会更加清晰和简单

```
using UnityEngine;
using System.Collections;

public abstract class SysModule : MonoBehaviour {

    //系统模块初始化
    public virtual bool Initialize() { return true; }

    //释放
    public virtual void Dispose() { }

    //开始执行
    public virtual void Run(object userData) { }
```

```
    //模块更新
    public virtual void OnUpdate() { }
}

------------------------------------------------------

public class SysSceneManager : SysModule {


}


public class SysUIEnv : SysModule {

}

public class ConfigDataBase : SysModule {


}

```

系统模块管理者类

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

public class SysModuleManager{

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

```
    }
    private SysModuleManager() { }

    //绑定了系统模块管理类的游戏物体
    private GameObject rootGameObject;
    public List<SysModule> modules = new List<SysModule>();

    private Dictionary<System.Type, SysModule> type_moduleMap = new
Dictionary<System.Type, SysModule>();

    public void Initialize(GameObject rootGameObject)
    {
        this.rootGameObject = rootGameObject;
    }

    //系统模块更新
    public void OnUpdate()
    {
        for (int i = 0; i < modules.Count; i++)
        {
            var module = modules[i];
            if (module != null)
                module.OnUpdate();
        }
    }

    //给游戏物体添加系统模块管理者
    public T AddSysModule<T>() where T : SysModule
    {
        Type t = typeof(T);

        if (type_moduleMap.ContainsKey(t))
            return type_moduleMap[t] as T;

        SysModule module=rootGameObject.AddComponent<T>();

        module.Initialize();

        modules.Add(module);
        type_moduleMap.Add(t, module);

        return module as T;
```

```
    }


    public T GetSysModule<T>() where T : SysModule
    {

        Type t = typeof(T);

        if (!type_moduleMap.ContainsKey(t))
        {
            for (int i = 0; i < modules.Count; i++)
            {
                var module = modules[i];
                if (module.GetType().IsSubclassOf(t))
                {
                    return module as T;
                }
            }
            return null;

        }
        return type_moduleMap[t] as T;
    }


}

```

程序入口类

```
using UnityEngine;
using System.Collections;

//项目程序入口 Main 负责游戏的主逻辑更新
public class GameMain : MonoBehaviour {

    void Awake()
    {
        DontDestroyOnLoad(gameObject);

```

```
        SysModuleManager.Instance.Initialize(gameObject);
        SysModuleManager.Instance.AddSysModule<SysSceneManager>();
        SysModuleManager.Instance.AddSysModule<SysUIEnv>();
        SysModuleManager.Instance.AddSysModule<ConfigDataBase>();

        SysSceneManager scene=SysModuleManager.Instance.GetSysModule<SysSceneManager>();
        //加载场景
        ConfigDataBase config = SysModuleManager.Instance.GetSysModule<ConfigDataBase>();
        //读取
    }

    void Update () {

        SysModuleManager.Instance.OnUpdate();
    }

    void OnApplicationQuit()
    {

    }
}

```

C#和Unity底层原理

运行库执行环境是.NET FrameWork的核心

运行库叫做： 公共语言运行库(Common Language Runtime) 简称 CLR

CLR的作用

1 微软所有的.NET产品的运行环境都是由CLR提供

2 CLR上运行的语言是一种以字节码形式存在的微软中间语言(MSIL或IL) 并不是常用的编程语言

所以在CLR运行的语言 都是可以编译成这个特定的中间语言的

3 CLR不是都运行于Windows系统 也可以运行在很多其他平台 比如移动平台

所以.NET语言可以很容易的实现"跨平台"

CLR的强大支持和跨平台实现

不同的语言有不同的编译器 编译器的存在目的是审查编程语言的语法是否正确

不同的语言会有很多的特性差异 语法和运行机制都不一定相同

而CLR提供了一个统一的运行平台 屏蔽不同语言之间的差异性

因为 不同的语言都被编译成了IL这个中间语言 而CLR只识别IL 不会理会IL是何种语言编译而来的

这也是Unity选择C#来开发跨平台游戏的原因

语言支持

基于CLR的编译器开发的语言：

1 微软开发 C# C++/CLI VB Python Ruby F# IL

2 其他机构 ： Lua LOGO PHP等

CLR支持的功能

1 基类库支持(.NET部分语言相似性的根源 System.Socket在很多语言中的代码实现和调用过程几乎一样)

2 内存管理

3 线程管理

4 垃圾回收

5 安全性

6 类型检查

7 异常处理

8 即时编译 (JIT编译器 动态编译 在程序运行时才编译代码 即将一条IL语句编译成机器语言然后执行 AOT是静态编译 运行前编译)

C#的底层编译原理

1 把源代码文件用对应的编译器检查语法和进行源代码分析

2 检查分析通过后 生成托管模块（PE32文件或者PE32+文件）

托管模块：

(1)中间语言IL: 编译C#源代码生成的中间代码

(2)元数据：元数据表 主要包含信息一种是源代码定义的类型和成员 一种是源代码中引用的类型和成员

VS编译器通过元数据帮助我们写代码

VS利用智能感知技术解析元数据 可以告诉我们一个类包含哪些方法 属性甚至是方法的参数列表等

3 将托管模块合并成程序集Assembly

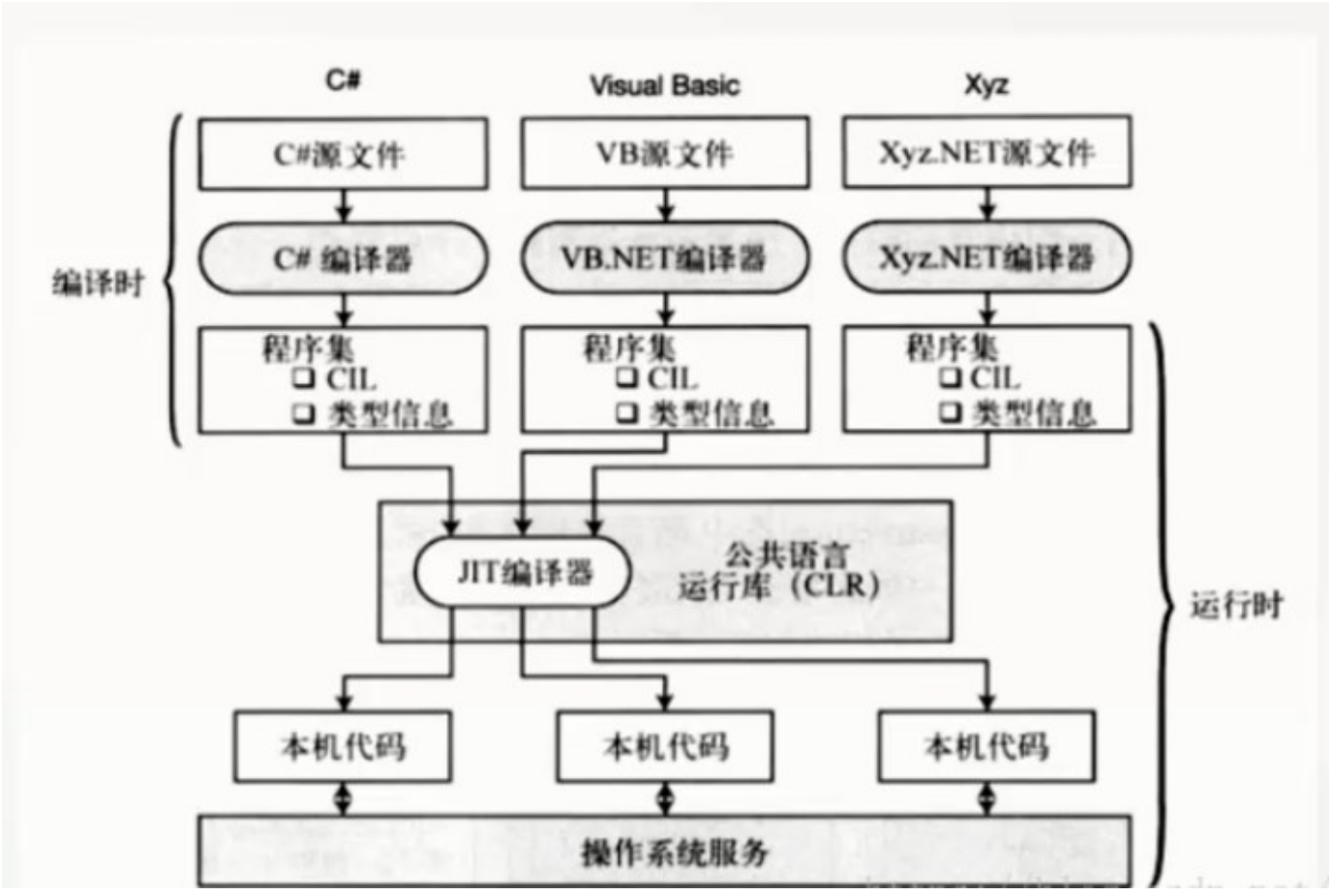
程序集包含.dll文件（扩展名文件）或.exe（可执行文件）

程序集是一个或者多个托管模块 以及一些资源文件的逻辑组合

程序集必须依靠CLR才能顺利运行 也是.NET FrameWork编程的基础组成部分

4 执行程序集代码

程序集中包含IL和数据 JIT编译器将IL转化成本机CPU可以理解的二进制指令



这里JIT编译器 起到了非常重要的作用

在安卓平台

我们的C#代码编译成了IL 然后在启动时再在JIT上进行编译

这也是部分功能我们无法再UnityEditor上测试的原因

IOS平台

因为IOS平台自身的限制问题 不能执行JIT

所以只能通过AOT（静态编译器）运行前就编译成目标代码

这也是我们不能直接通过Untiy进行IOS项目打包 而必须使用XCode的原因

C#的内存管理和垃圾回收

内存基础知识

内存主要分为栈内存和堆内存

内存管理的三个阶段：

内存分配

内存生命周期内管理

内存释放

C#的托管对采用连续内存存储 C/C++是不连续的

新创建一个对象 先判断内存是否足够使用 如果不足够使用就需要扩展堆的长度

堆内存中有一个堆指针 记录下一个对象分配的位置

问题在于 如果一直对象不去析构堆的内存会无限增大

所以管理这一块连续内存的机制 就是垃圾回收机制（GC）

GC的工作就是在需要他得时候释放掉垃圾对象 也可以手动释放System.GC.Collect()

然后把没有释放的对象Copy到堆的端部

他们仍然在一起形成新的连续内存 更新堆指针位置

注意 在GC执行期间 所有的线程暂停工作