CTFramework

时间：2022年4月11日12:51:12

框架版本：v0.2

- 由于框架和文档都是我一人纯手工编写，因此文档和实际代码可能会有所出入，一切以实际代码为准！
- 不足之处，请多包涵！如果有任何建议，可以发送到我的邮箱：1322080797@qq.com，让我们一起进步！

更新内容

v0.2版本较v0.1没有多大变化，甚至删减了网络模块（对旧网络框架不太满意）

主要修改了多处小bug，改动幅度较大的部分记载如下：

> 文档

本文档定位是教程文档，将移除之前绝大部分教程无关内容

> 系统模块

- 新增**框架辅助模块**
  - 将一部分管理类交付给单例类`CTHelper.cs`管理
- 完善全局配置文件的工作机制
  - 提供重定向方案
  - 提供对整个框架的全局配置
- 优化工厂
  - 所有模块按需加载，不会创建不需要的模块内容以节省内存
  - 统一核心模块内部的生产方式
- 完善玩家数据类，由`CTHelper`管理
  - 分为**运行时数据（Key以~开头）**和**存档数据**

> UI模块

- 每次Push时可传参
- 提供预加载选项
  - 需预先使用PushToCachePool将面板推入缓存池（注意缓存池的大小）
- 重新设计IController
  - 彻底避免拆装箱
  - 基于链式编程思想
- 缓存的UI面板可以手动卸载

> 音乐模块

- 音乐音效新增重载路径，方便资产复用
- ~~背景音乐的淡入淡出~~（拒接接入，需要引入DOTween插件）
  - 改进，暴露获取系统音量和配置文件音量的方式，由用户自己去相乘控制音量
- 音效预加载可以不进行缓存，在播放一次后自动删除缓存（应对诸如一次性对话的场景）

简介

开发环境：Unity2019.3.x LTS、Visual Studio 2017

特点：简单与轻量，高可读，高可用，高可拓展

设计思想：数据，行为，初始化三者分离，面向接口编程

- 数据：由配置文件驱动（目前均直接使用Unity自带的ScriptableObject）
- 行为：由加载器驱动（目前均使用Loader命名加载器）
- 初始化：由工厂类驱动

插件

协助优化工作流，部分非必须

> 所有dll必须直接放在Plugins下，Plugins的位置可以任意调整

- Odin：第三方插件，必须。用于编辑器拓展，提供可序列化的ScritableObject
  - 下载路径：Asset Store购买（等我赚够了，我就不用某宝的盗版）
  - 存放路径：`Assets/CTFramework/Dependencies`
- AssetBundleBrowser：官方插件，非必须。用于打包AssetBundle包
  - 可以使用脚本替代（详见宣雨松《Unity3d游戏开发》第二版P309）
  - 下载路径：
    - PackageManager：有的话直接装（Unity 2020版本我好像没找到）
    - Github下载：https://github.com/Unity-Technologies/AssetBundles-Browser
  - 存放路径：`Packages`
- TextMeshPro：官方插件，非必须。用于文字的显示与特效
  - 下载路径：PackageManager
  - 存放路径：`Assets/CTFramework/Dependencies`
- InputSystem：官方插件，非必须。用于输入系统，缺少该插件可能导致案例报错，此时删除案例可以解决
  - 存放路径：`Packages`
- Newtonsoft.Json：第三方插件，必须。提供复杂Json的解析
  - 教程URL：https://blog.csdn.net/qq471263589/article/details/95318118
  - 序列化性能比较：https://www.cnblogs.com/landeanfen/p/4627383.html
  - 命名空间：`using Newtonsoft.Json;`
  - Unity提供的JsonUtility只能解析对象，无法直接解析List，数组，Dictionary等类型。因此需要使用Newtonsoft.Json来应对某些场景。能否在项目打包后运行，有待考证
  - 存放路径：`Assets/CTFramework/Plugins`

> 服务端

- MySql.Data.dll：
  - 教程：https://dev.mysql.com/doc/connector-net/en/

---------- 框架快速上手 ----------

> 站在使用者角度学习框架内容，只想使用该框架开发的同学可以只看该章内容

> - 如遇不清楚的概念可以尝试在答疑解惑部分找到回答
> - 所有案例资源存放在：`Assets/CTFramework/Example`中
> - 代码块中的`...`代表做某些事情，不该成为读者重点关注的部分

文件夹介绍

- 绝大多数情况下（特殊情况会在此处说明），框架文件应置于`Assets/CTFramework`文件夹下，根据模块命名
  - 例如：`1.ResourceSystem`
- 编辑器拓展类脚本，按照要求，全部放在`Assets/Editor/CTFramework`下
- CTFramework下的重要文件夹：
  - `0.Core`：必须，框架核心文件
  - `Examples`：可选，示例文件夹
  - `Tools`：必须，工具集
    - 位于`CT.Tools`命名空间下
  - `Plugins`：必须，插件集
  - `Resources`：必须，除Examples外其他谨慎删除，框架内动态加载文件时可能需要

框架辅助模块

介绍：框架辅助模块的重要程度介于核心模块和工具类之间，它们是框架中重要但非必要的辅助工具。

所有辅助模块均位于`CTHelper`单例类中

CTHelper路径：`Assets\CTFramework\0.Core\CoreSystem\Modules\CTHelper.cs`

1) 场景管理类

概述：Unity场景切换的增强版，在场景切换的几个重要结点设置相应回调函数，方便用户控制

> 场景生命周期：设A为初始场景，B为下一个场景
> 场景A：所有物体的Awake(Start)->LoadBefore(Pop UI)->所有物体的OnDestory->Unloaded
> 场景B：loaded->所有物体的Awake(Start)->afterLoaded
> 所有回调均为一次性回调，每次触发后都会清空

SceneMgr.cs 重要代码

```
1  public interface ISceneMgr
2  {
3    //同步加载场景
4    //参数①：场景名称
5    void LoadScene(string name);
6
7    //异步加载场景
8    //参数①：场景名称
9    //参数②：加载时回调函数
10   //参数③：切换场景条件
11   void LoadSceneAsync(string name, Action<float> loading = null, Func<bool>
   loaded = null);
12
13   //场景生命周期：设A为初始场景，B为下一个场景
```

```
  //场景A：所有物体的Awake(Start)->LoadBefore(Pop UI)->所有物体的OnDestory->Unloaded
  //场景B：loaded->所有物体的Awake(Start)->LoadAfter
  //所有回调均为一次性回调，每次触发后都会清空

  void SetLoadBefore(Action callback);

  void SetUnLoaded(UnityAction<Scene> callback);

  void SetLoaded(UnityAction<Scene> callback);

  void SetLoadAfter(Action callback);
}
```

使用举例

同步加载

```
//参数①：场景名
CTHelper.sceneMgr.LoadScene("xxx");
```

异步加载

```
 //异步加载，完成后自动切换场景
CTHelper.sceneMgr.LoadSceneAsync("xxx", true);
//异步加载，完成后手动切换场景
//参数①：场景名
//参数②：是否自动切换场景
//参数③：加载时回调函数，传入参数float
//参数④：可以切换场景的判断函数，返回值为bool
CTHelper.sceneMgr.LoadSceneAsync("xxx", false, progress=>{...}, ()=>{...return bool值}); //异步加载，不自动切换场景
```

2) 定时回调系统

概述：Unity定时回调系统，用于延迟执行某一函数，该系统会受到 `timeScale` 的影响。如果使用了Wait函数，最好每次加载场景时调用Reset方法重置TimerMgr

TimerMgr.cs 重要代码

```
public interface ITimerMgr
{
  //定时回调函数(受Timescale的影响)
  //返回值：TaskID
  //参数①：延迟时间(单位:秒)
  //参数②：回调函数
  //参数③：触发次数（默认1次，-1代表无限循环）
  //参数④：触发间隔（默认-1，-1代表以延迟时间为间隔）
  int Wait(float delay, UnityAction callback, int count = 1, float interval = -1);

  //设置组名
  //参数①：Task ID
  //参数②：组名
```

```
    void SetGroupName(int ID, string groupName);

    //根据ID取消任务
    void Cancel(int id);

    //取消一组任务
    void CancelGroup(string groupName);

    //取消所有任务
    void CancelAll();
}
```

使用举例

```
CTHelper.timerMgr.Wait(2f, ()=>Debug.Log("2s后我会被执行一次"));
CTHelper.timerMgr.Wait(2f, ()=>Debug.Log("2s后我会被执行三次"), 3);
CTHelper.timerMgr.Wait(2f, ()=>Debug.Log("2s后我会被执行一次，间隔1s再次执行"), 2, 1);
CTHelper.timerMgr.Wait(2f, ()=>Debug.Log("我会间隔2s，且无限执行"), -1, 2);
```

框架核心模块

核心配置文件

概述：可以对整个框架进行全局配置（目前需一次性配置所有框架内容，随着框架增大，将采用SpringBoot的动态配置思想，方便用户）

CTConfig 路径：`Assets\CTFramework\0.Core\Dynamic\CTConfig.cs`

1.资源系统

资源系统概述

说明：资源系统模块是**可选**的，负责资源的同步和加载工作。目前支持AB包加载。Resources加载请直接参考Unity API，这里就不画蛇添足了（最多拓展一下异步对Resources加载方式）

适用场景：

- 需要进行**资源热更新**的**网络游戏**
- 需要使用带回调的Resources异步加载

资源配置文件

概述：资源模块的配置文件，存放初始化配置信息

所在类名：ResourceProfile

范围：全局有效

字段	说明
isABLoad	是否启用AB包加载，只有启用下面的属性才生效（v0.2新增！）
localPath	AB包本地路径，如果从StreamingAssets加载，直接写子目录；否则写完整路径
mainABName	主包文件名
isStreamingAssets	是否从StreamingAssets加载本地资源
abVersionList	AB包清单列表文件，用于本地与服务器端版本校验
isNetUpdate	是否更新资源
netPath	网络路径
retryCount	更新失败的重试次数

> 有个印象就好，配合案例食用效果更佳

资源系统接口

IRes 接口

概述：可供用户使用操作资源的核心方法**（重点掌握）**

```
public interface IRes : IRelease
{
  //资源同步
  //参数①：更新时回调
  //参数②：更新成功回调
  //参数③：更新失败回调
  void UpdateAssetBundle(
    UnityAction<float> loading,
    UnityAction successLoaded,
    UnityAction failLoaded);

  #region AB包同步加载
    //泛型同步加载
    //参数①：ab包名
    //参数②：资源名
    T LoadRes<T>(string abName, string resName) where T : Object;

  //类型同步加载
  //参数①：ab包名
  //参数②：资源名
  //参数③：类型
  Object LoadRes(string abName, string resName, System.Type type);
```

```
  #endregion

    #region AB包异步加载
    //泛型异步加载
    //参数①：ab包名
    //参数②：资源名
    //参数③：回调函数
    void LoadResAsync<T>(string abName, string resName, UnityAction<T>
callback) where T : Object;
  //类型异步加载
  //参数①：ab包名
  //参数②：资源名
  //参数③：类型名
  //参数④：回调函数
  void LoadResAsync(string abName, string resName, System.Type type,
UnityAction<Object> callback);
  #endregion

    #region AB包卸载
    //单个包卸载
    void UnLoad(string abName);
  //所有包卸载
  void ClearAll();
  #endregion
}
```

拓展：异步Resources加载

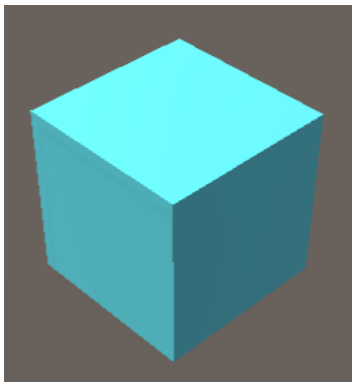
```
public static class ExtendResMgr
{
  //Resources异步加载增强
  public static void LoadResAsync<T>(this IRes mgr, string path,
UnityAction<T> callback) where T : UnityEngine.Object
  {
    _CT.Instance.StartCoroutine(LoadEnumerator(path, callback));
  }

  //异步加载协程
  private static IEnumerator LoadEnumerator<T>(string path, UnityAction<T>
callback) where T : UnityEngine.Object
  {
    ResourceRequest request = Resources.LoadAsync<T>(path);

    while (!request.isDone)
    {
      //Debug.Log("Loading progress: " + resourcesRequest.progress);
      //资源没有加载完成  返回空，加载完以后进行后面的模块
      yield return null;
    }
    T asset = request.asset as T;
    callback.Invoke(asset);
  }
}
```

示例1.资源打包

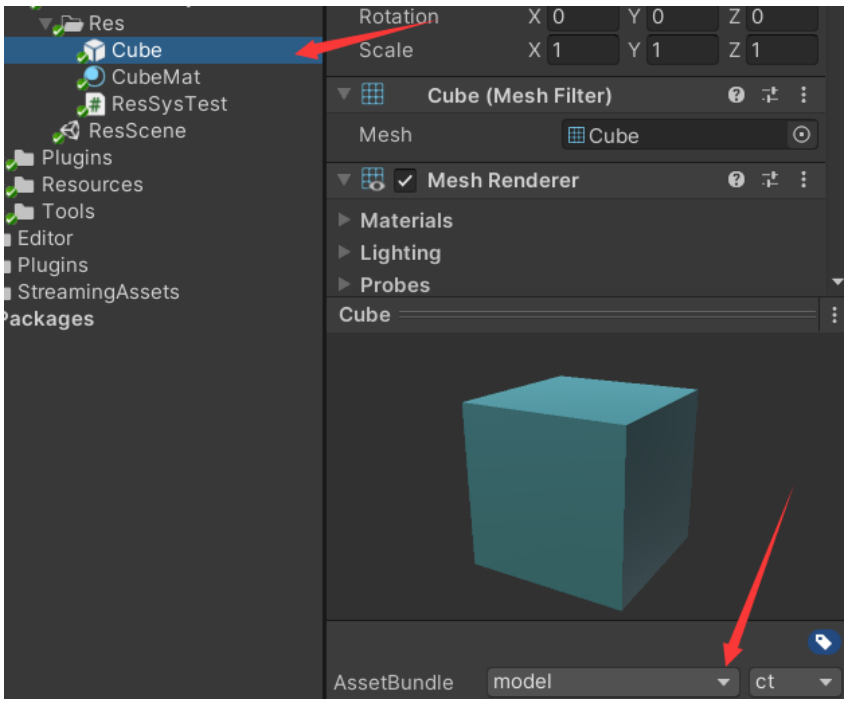
介绍：本案例会从最基础的开始，手把手教你创建AssetBundle，为之后资源的加载做好准备。这里提一点建议，就是把美术素材等单独打成一个AssetBundle包，避免美术资源的重复依赖，节省ab包大小。

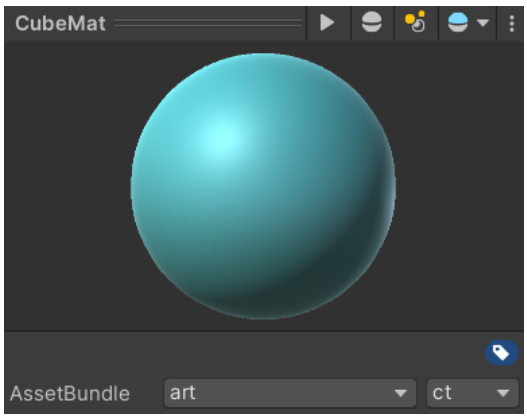
- 学习目标：掌握打包AssetBundle资源

1）在场景中创建一个立方体，并赋予材质。并生成其Prefab



2）在Inspector的Preview窗口下，指定资源的AssetBundle的名称和后缀

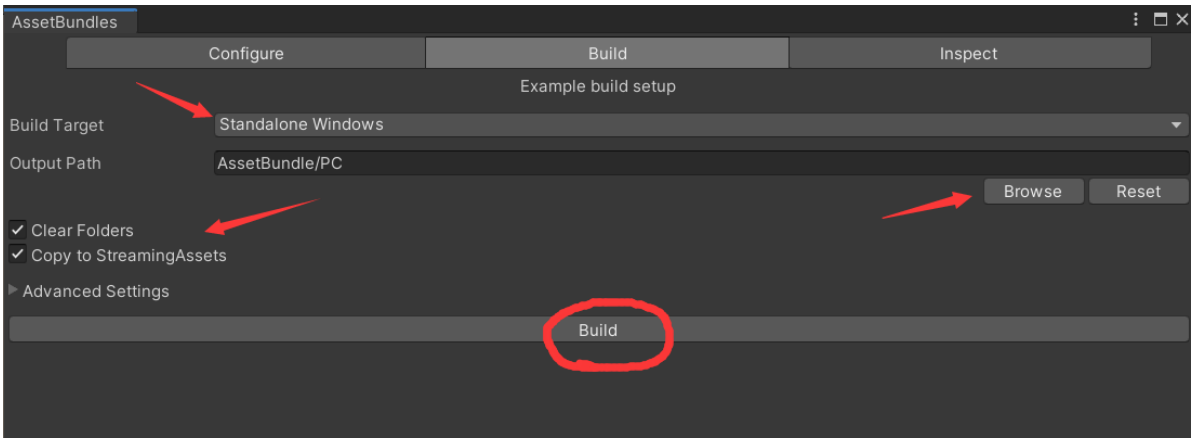




> 这里将立方体Prefab打包进model.ct，材质打包进art.ct

3）打开菜单栏选项卡`Window->AssetBundleBrowser`，配置相应参数后点击Build

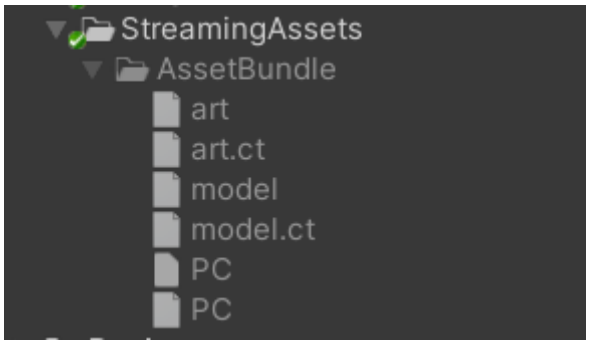
- 这里拷贝一份到StreamingAssets是为了下一步的本地测试



> 如果不了解AssetBundle的话，请参考：
>
> - AssetBundle个人整理笔记：https://www.yuque.com/congtou-l6afc/unity3d/ab

4）资源打包成功！

> StreamingAssets的副本



- 在下个案例中，我们会利用 `StreamingAssets文件夹` 的内容加载本地AssetBundle资源，准备好就开始吧！

> 注意：在实际开发中，不一定要使用StreamingAssets文件夹，可能会有读写问题，可以考虑其他文件夹或放到持久化文件夹下

示例2.本地资源加载

介绍：本示例会教你如何使用AssetBundle实现本地资源的加载，生成一个之前打包的立方体对象。

- 学习目标：从本地加载AssetBundle资源

1）修改默认配置文件：

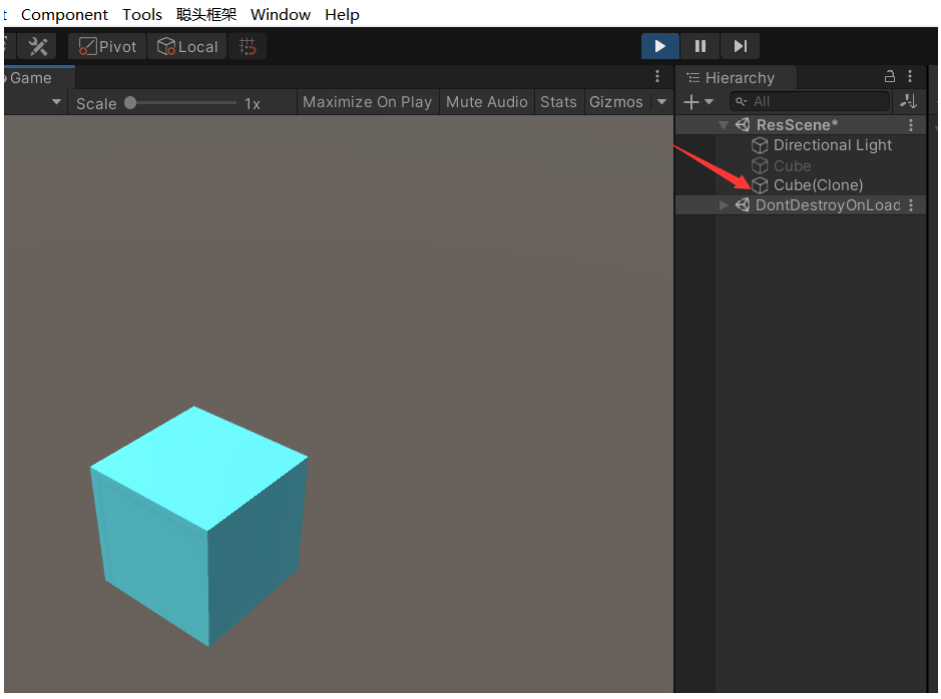
- 位于：`Resources/CTFramework/Default/` 下
- 每个参数说明详见 资源配置文件

2）挂载 `_CT.cs` 核心脚本类到任意已激活对象上

3）新建 `ResSysTest.cs` 脚本，挂载到任意激活的对象上

```
public class ResSysTest : MonoBehaviour
{
  private void Start()
  {
    //示例2
    //参数①：ab包名称，带后缀
    //参数②：资源名称
    Instantiate(_CT.ResMgr.LoadRes<GameObject>("model.ct", "Cube"));
  }
}
```

4）运行即可看到结果



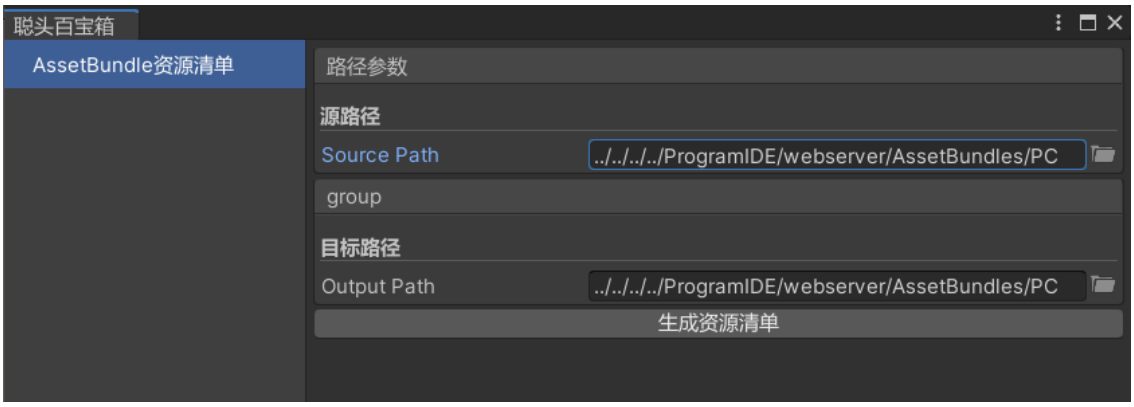
> 怎么样是不是感觉超级简单？！好吧，可能和Resources.Load比较，没啥优势，又要打包又要配置，挺麻烦的。别急，下面才是重头戏，我们将引入网络对资源进行下载和加载

示例3.网络资源加载

介绍：本示例会手把手教你如何进行网络资源的更新，并且加载资源。目前暂不支持资源的单独下载与校验，仅支持一次性更新并校验所有服务器资源。后期视具体情况，再考虑是否单独暴露出具体方法。

- 学习目标：从网络同步AssetBundle资源并加载

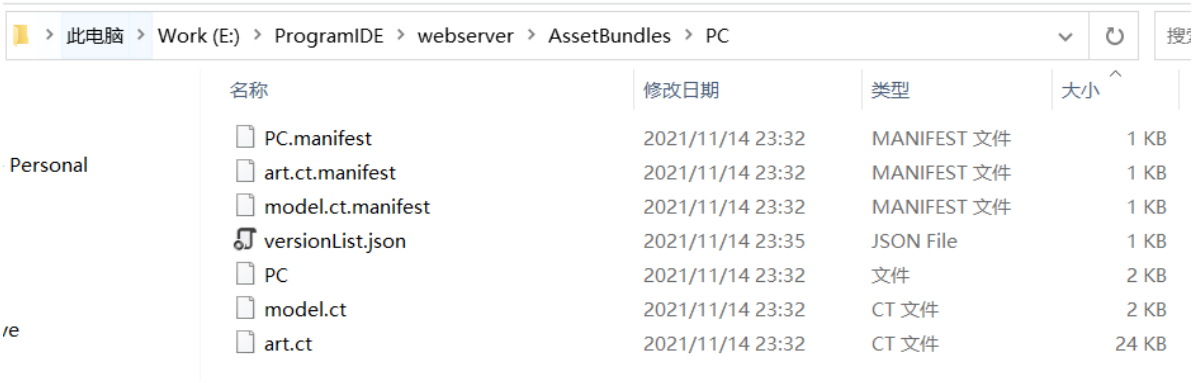
1）先删除之前StreamingAssets内拷贝的ab包资源(没有内容请忽略该句)，此时运行示例2脚本必定报错。接着，选择菜单栏 `聪头框架->工具箱->AssetBundle资源清单`，选择AssetBundle所在路径，生成一份资源清单



2）配置web服务器，这里使用最轻量的 `NetBox2`。演示在本地进行，如果需要真正从网络下载资源，只需要购买云服务器，把NetBox2运行在云端即可，这里不进行演示。后面可能会更换WebServer，NetBox2有点老，似乎端口号啥的也没法配置，再说吧

> Netbox2自取：
>
> - 就一个.exe，扔哪个文件夹，哪个文件夹下就是web根目录（神奇！）
> - 百度网盘下载链接：https://pan.baidu.com/s/1O_lU4MY2gJWhIsLlWwQU1Q
>   提取码：y9be
>
> 检测是否安装成功：
>
> - 安装后，在同级目录下创建index.html，随便输入一些内容
> - 运行服务后，在浏览器输入：http://localhost:57223/，如果可以正常显示，则运行成功。
>   (PS: URL的端口号要根据实际运行时的参数修改)
>
> 
>

3）在NetBox2同级目录下创建 `AssetBundle/PC` 文件夹，将所有AssetBundle资源及清单文件拷贝到该目录下



4）修改默认配置文件(如下图)

- 主要修改的参数是网络路径，修改成资源所在目录的URL

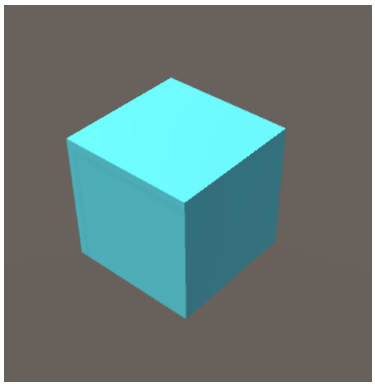
5）挂载 `_CT.cs` 核心脚本类到任意已激活对象上

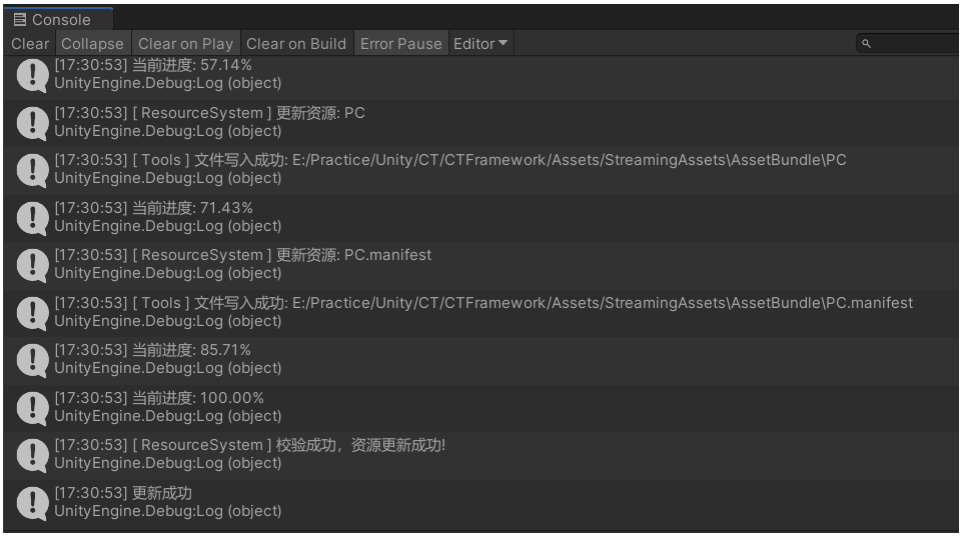
6）新建 `ResSysTest.cs` 脚本，挂载到任意激活的对象上，输入以下内容

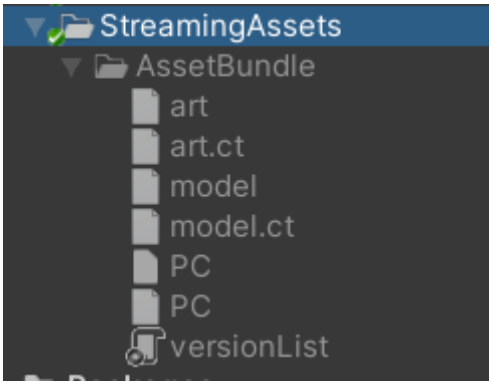
```
public class ResSysTest : MonoBehaviour
{
  private void Start()
  {
    //示例3
    //参数①：更新中回调
    //参数②：更新成功回调
    //参数③：更新失败回调
    _CT.ResMgr.UpdateAssetBundle(
      x => Debug.Log($"当前进度: {(100 * x):F2}%"),
      () =>
      {
        Debug.Log("更新成功");
        Instantiate(_CT.ResMgr.LoadRes<GameObject>("model.ct", "Cube"));
      },
      () => Debug.Log("更新失败"));
  }
}
```

> 通常情况下，更新成功就切换场景，然后正常调用LoadRes方法即可。这里在成功回调里面生成游戏对象，主要是因为更新操作在协程中进行，必须要等所有资源更新成功并校验后，才可以生成对象

7）运行结果







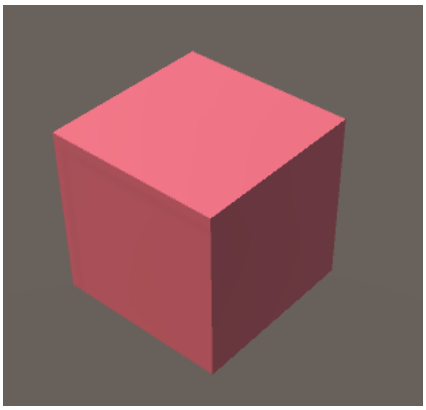
> 此时会发现本地的StreamingAssets路径下多了一些文件，说明资源也已经同步更新了。再次运行，会让本地清单和服务器清单作比较，如果一致，则无需下载资源，加载会更快；如果不一致，也只会下载不一致的部分。

示例4.素材替换

介绍：本示例会展现AssetBundle加载的魅力之处，在不修改任何工程代码与资源的情况下，替换美术资源。当然并不仅限于美术素材，一切能打包成AssetBundle的资源我们都可以动态替换。本示例中我们会把之前加载的蓝色立方体，颜色换成红色

- 学习目标：不修改何旧资源的前提下，完成素材的替换

参考 示例3.网络资源加载，我们把修改后的CubeMat重新打包AssetBundle，重新生成资源清单（VersionList.json），上传至服务器即可



恭喜！至此，所有的资源加载模块的内容就已经顺利学完了！当然，笔者时间有限，并没有演示全部内容，但是参照接口提供的方法，掌握剩下的内容，应该不难叭。就这样喽，下章见，谢谢观看！

> 资源系统文档完结于：2021年11月16日00:04:05

2.UI系统

UI系统概述

说明：UI是游戏中非常重要的环节，如何让UI更高效地与游戏内容协作也是一直困扰我的问题。这次，经过长时间思考和大量实践，我自己总结了一套非常完善的UI解决方案，它兼顾效率与实用性。下面作简要介绍：

- **支持不同的面板加载方式**
  - 目前支持 `Resources` 和 `AssetBundle` 加载
  - 之前的面板加载，全部都是Resources提供的，虽然简单，但是不够灵活，适合硬编码的单机游戏。这次引入了额外的AssetBundle加载，只需简单配置，即可根据需求加载不同风格的UI
- **支持面板缓存**

    - 有效缓解了反复加载面板导致的略微延迟和卡顿的现象
    - 面板缓存仅适用于**非破坏性面板**（非破坏性面板：在整个生命周期没有动态增删内容的面板。不是什么高大上的词，实际上就是控制面板的激活与关闭。如果需要破坏面板，在配置文件中就不要选择缓存了）
    - 之前的面板，都是Push即New，Pop即删。不仅可能导致加载延迟，频繁地入栈出栈也容易造成内存碎片，影响游戏性能。所以，对于需要频繁入栈出栈的面板对象，这次提供了缓存技术，牺牲空间换时间，我看行！
- **基于"MVVM"的设计**
    - ~~什么是MVVM呢？好像是什么双向绑定？其实我也不太了解。。所以打了个引号（后面出现的MVVM，我就不打了，如果理解有误，那就按我的来）。因为我没来得及学完Vue，就因考研放弃了前后端的学习，甚是遗憾（先留着，以后再找它算账）。但是学过MVC都知道，把模型、控制、视图分离，有利于项目的开发。于是我猜测MVVM就是MVC的改进，反正基于此思想设计，使用起来非常方便！亲测很棒！~~
    - 前后端交互的载体是ModelView，其本身是基于 `Key-Value` 的。这样做的好处是，每次修改数据（后端），你根本不需要知道它是哪个UI控件，修改 `Key` 和 `Value` 就能在UI面板上体现（前端）。当然需要面板初始化时在前端进行绑定
    - 支持动态绑定监听。你可以利用 `Key` 方便地为其绑定回调函数，不仅限于绑定UI（例如，可以绑定玩家的血量，当血量低于多少时触发回调函数）。~~同时为了避免无脑调用回调函数，我对浮点类型和整型引入阈值控制，仅当变换幅度大于阈值时，才会更新值，触发回调函数。~~
- **支持一键生成脚本和配置文件**
    - 这是UI技术的一大亮点，极大地解放生产力。按照规范设计的UI面板对象，利用我编写的编辑器工具，就能一键生成所需脚本和配置文件，并且完成一部分初始化工作。
    - 较之前相比，我对底层的生成算法进行了进一步优化，能够以最低的性能代价找到UI控件。加之这次引入的面板缓存技术，性能较之前版本有了大幅提升。（当然，最省性能的方式还是拖拽赋值，弊端懂得都懂）
- **合理使用泛型，避免拆装箱消耗**
    - 暴露的接口使用泛型，方便调用者使用
    - 分享一下之前落后的设计。早期设计所有的 `Key-Value` 都是string，这样做有两个弊端：
        - 一是许多场景使用时要进行类型转换，麻烦又低效
        - 二是转换失败可能会有安全隐患，如果多加判断又白白浪费性能。
    - 凡事具有两面性。这样设计唯一的好处就是开发期间特别爽，大家都是string，一套逻辑直接搞定，容易非常多。但再三思索下，我还是放弃了之前的设计方案。设计了一套全新的底层交互机制，使用起来不仅比之前方便，性能也好得多。（希望）努力没有白费！

> 一个UI面板想要顺利运行在框架上需要用户提供三个文件：
>
> 1. UI面板的Prefab
> 2. 配置文件（PanelProfile）
> 3. ViewComponent的实现类（xxView.cs，最好提前挂载在面板根GameObject）

面板生命周期函数

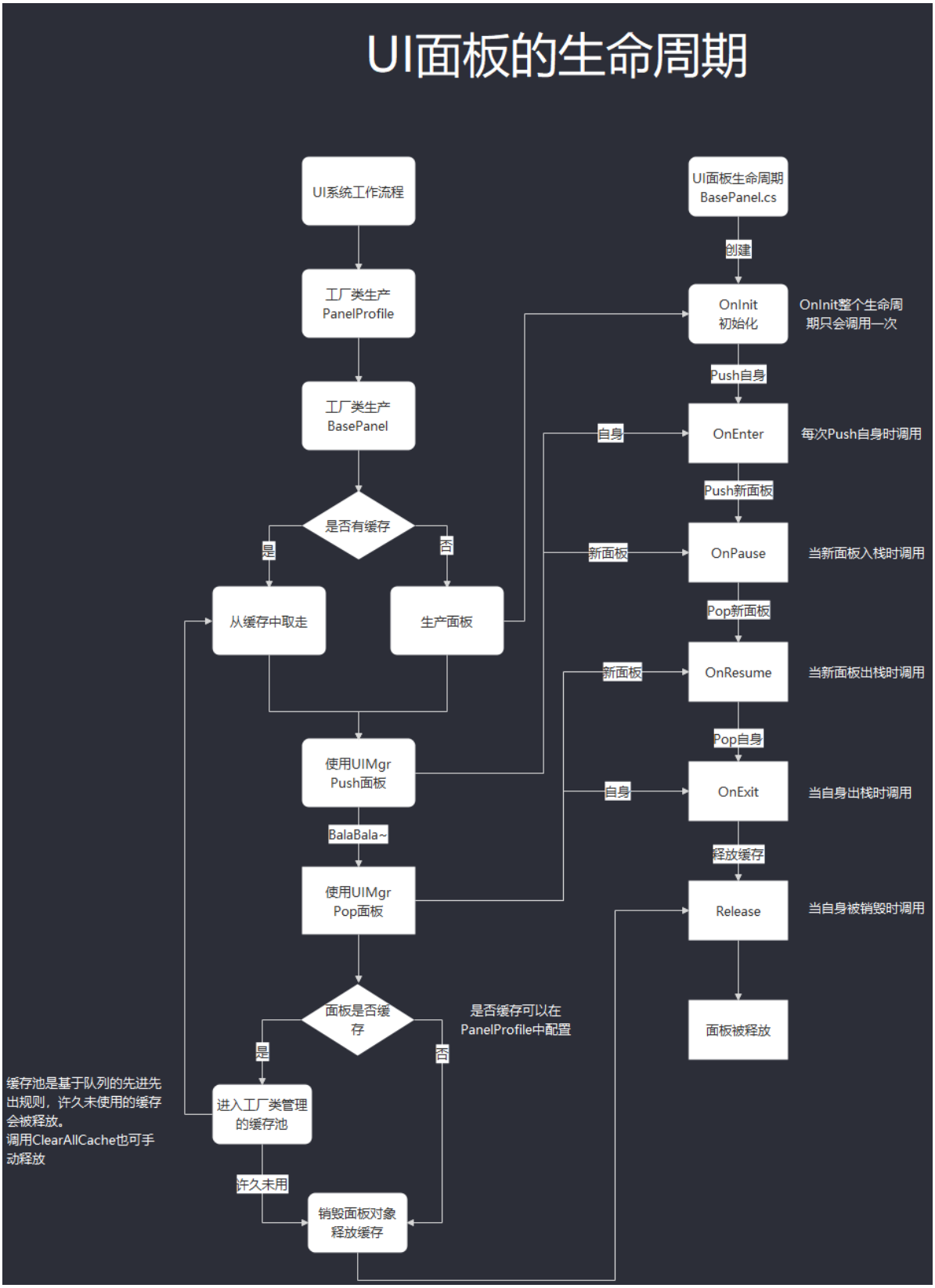
完整生命周期：Awake -> OnInit -> OnEnter(可能多次) -> Start -> OnExit(可能多次) -> OnRelease

概述：了解面板的生命周期函数，有利于理解UI系统的工作流，同时也方便用于拓展自己的需求

- OnInit：初始化函数，整个生命周期中**只执行一次**
    - 底层：在创建GameObject之后，调用OnInit
    - 任务：查找所有UI组件 -> 手动初始化全局Key-Values -> 添加监听
- OnEnter：当前面板Push时调用，如果启用了面板缓存，则可能多次调用

    - 底层：在Push面板时，调用OnEnter（Push的面板可能是全新的，也可能来自缓存池）
    - 任务：从配置文件初始化Key-Values -> 手动初始化Key-Values 和 面板数据（在外界看来是全新的，但是底层仅仅是激活与关闭了该对象）
- OnPause：新面板Push时旧面板调用
    - 任务：冻结当前面板所有UI
- OnResume：新面板Pop时旧面板调用
    - 任务：解冻当前面板所有UI
- OnExit：面板自身Pop时调用，如果启用了面板缓存，则可能多次调用
    - 底层：关闭面板GameObject显示，根据缓存属性选择进入缓存池还是释放数据
- Release：释放UI面板

UI面板生命周期底层实现流程图：



UI配置文件

UI配置文件分两类：UI全局配置文件、UI面板配置文件

UI全局配置文件

概述：描述UI工厂的缓存策略和默认加载面板配置文件的方式

范围：全局有效

所在类名：UIProfile

字段	说明
IsSingleCache	启用单缓存还是缓存池技术
MaxCache	UI面板缓存池上限
isABLoad	是否启用AB包加载该配置文件
profileDir	Resources加载时的路径
abName	AB包加载时的包名

面板配置文件

概述：面板初始化时的配置文件

所在类名：PanelProfile

范围：局部有效（面板配置文件是针对具体某一面板进行参数配置）

字段	说明
panelName	面板预制体名称
isCache	是否启用面板缓存
isPreLoad	是否启用预加载，Push前必须先调用AddToCachePool函数（v0.2新增！）
isAssetBundle	是否使用AssetBundle加载预制体
resourcesDir	Resources加载时有效，预制体所在目录
abName	AssetBundle加载时有效，预制体所在包名
isCacheProfile	是否启用预定义ModelView（v0.2新增！） 启用后游戏运行时，如果面板需要缓存，则会缓存该配置文件用于初始化ModelView
keyValues	预定义键值对，用于UIModelView的初始化

UI系统接口

IUI 接口

概述：提供管理面板的方法，底层是基于栈，所以管理方法类似栈的调用 **（重点掌握）**

```
/***************************************************
  文件：IUI.cs
  作者：聪头
  邮箱：  1322080797@qq.com
  日期：2021/11/15 20:22:48
  功能：
***************************************************/
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

namespace CT.UISys
{
  public interface IUI : IRelease, IUIController
  {
    #region 栈顶操作
      //UI入栈操作（UIProfile中默认配置为Resources）
      //参数①：加载目录或者AB包
      //参数②：加载PanelProfile名称（可以省略后缀）
      //参数③：传递给面板的参数，在OnEnter时访问
      void Push(string resDir, string panelName, object obj = null);

    //UI入栈操作（需要在UIProfile中配置默认加载路径）
    //参数①：加载PanelProfile名称
    //参数②：传递给面板的参数，在OnEnter时访问
    void Push(string panelName, object obj = null);

    //UI缓存操作（需要在UIProfile中配置默认加载路径）
    //参数①：加载PanelProfile名称（可以省略后缀）
    //默认将面板设为可缓存类型
    void AddToCachePool(string panelName);

    //从缓存（池）中移除
    //参数①：加载PanelProfile名称
    void RemoveFromCachePool(string panelName);

    //出栈
    void Pop();

    //清空栈
    void AllPop();

    //获得栈顶UI面板
    BasePanel PeekPanel();

    //通过泛型获得栈顶UI面板的ViewComponent具体实现类
    T PeekViewComponent<T>() where T : ViewComponent;

    //获得栈顶UI面板的游戏对象
```

```
    GameObject PeekObj();

    //获得栈顶UI面板的Controller
    IUIController PeekController();

    //根据名称返回栈顶面板是否匹配
    bool isPeek(string panelName);
    #endregion

      #region 非栈顶操作
      //根据面板名称获得UI面板（包括缓存面板）
      BasePanel GetPanel(string panelName);

    //根据面板名称获得UI面板游戏对象（包括缓存面板）
    GameObject GetObj(string panelName);
    #endregion
  }
}
```

IController 接口

概述：Controller层提供的接口，提供前后端数据交互，绑定监听的功能**（重点掌握）**

```
/***************************************************
  文件：IController.cs
  作者：聪头
  邮箱：  1322080797@qq.com
  日期：2021/11/17 16:26:54
  功能：
***************************************************/
using CT.UISys;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;

namespace CT
{
  public interface IUIController
  {
    //得到ModelView
    UIModelView GetModelView();

    //设置Key-Value值，有则覆盖，无则添加
    //参数①：键
    //参数②：值
    //参数③：是否触发回调事件
    //参数④：是否在第一帧调用After监听，渲染数据
    UIBool SetBool(string key, bool value, bool isTrigger = true, bool
isFirstRender = true);
    UIInt SetInt(string key, int value, bool isTrigger = true, bool
isFirstRender = true);
    UIFloat SetFloat(string key, float value, bool isTrigger = true, bool
isFirstRender = true);
```

```
    UIString SetString(string key, string value, bool isTrigger = true, bool isFirstRender = true);

    //根据Key，得到UI对象，访问data可以取出值
    UIBool GetUIBool(string key);
    UIInt GetUIInt(string key);
    UIFloat GetUIFloat(string key);
    UIString GetUIString(string key);

    //根据Key，得到值
    bool GetBoolValue(string key);
    int GetIntValue(string key);
    float GetFloatValue(string key);
    string GetStringValue(string key);

    //根据Key，删除Property
    void RemoveUIBool(string key);
    void RemoveUIInt(string key);
    void RemoveUIFloat(string key);
    void RemoveUIString(string key);

    //修改条件函数，每当根据Key赋值时会调用，返回true则更新值，false则不更新
    //参数①：键
    //参数②：二元谓词
    void ModifyBoolCondition(string key, System.Func<bool, bool, bool> condition);
    void ModifyIntCondition(string key, System.Func<int, int, bool> condition);
    void ModifyFloatCondition(string key, System.Func<float, float, bool> condition);
    void ModifyStringCondition(string key, System.Func<string, string, bool> condition);

    //添加before监听，数据改变前触发
    //参数①：键
    //参数②：回调函数，第一个参数代表旧值；第二个参数代表新值
    UIBool AddBoolBeforeListener(string key, UnityAction<bool, bool> callback);
    UIInt AddIntBeforeListener(string key, UnityAction<int, int> callback);
    UIFloat AddFloatBeforeListener(string key, UnityAction<float, float> callback);
    UIString AddStringBeforeListener(string key, UnityAction<string, string> callback);

    //添加after监听，数据改变后触发
    //参数①：键
    //参数②：回调函数，参数代表新值
    UIBool AddBoolAfterListener(string key, UnityAction<bool> callback);
    UIInt AddIntAfterListener(string key, UnityAction<int> callback);
    UIFloat AddFloatAfterListener(string key, UnityAction<float> callback);
    UIString AddStringAfterListener(string key, UnityAction<string> callback);

    //删除before监听
```

```
    void RemoveBoolBeforeListener(string key, UnityAction<bool, bool>
callback);
    void RemoveIntBeforeListener(string key, UnityAction<int, int>
callback);
    void RemoveFloatBeforeListener(string key, UnityAction<float, float>
callback);
    void RemoveStringBeforeListener(string key, UnityAction<string, string>
callback);

    //删除after监听
    void RemoveBoolAfterListener(string key, UnityAction<bool> callback);
    void RemoveIntAfterListener(string key, UnityAction<int> callback);
    void RemoveFloatAfterListener(string key, UnityAction<float> callback);
    void RemoveStringAfterListener(string key, UnityAction<string>
callback);

    //删除所有监听
    void RemoveAllListener();

    //强制触发所有监听，一般用于初始化双向绑定后的数据渲染
    void InvokeAll();
  }
}
```

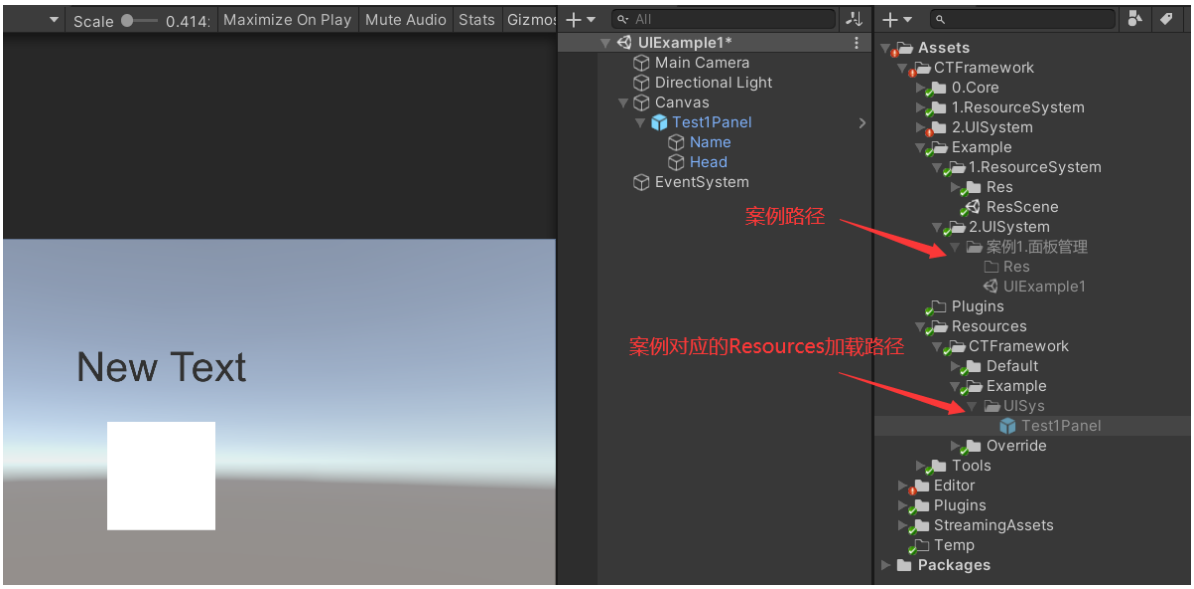
示例1.管理面板

介绍：本案例会教你如何创建符合规范的“最最基础”也是“最最自由”的Panel，并且会手把手教你如何在运行时管理这些面板。掌握本节内容后，你甚至可以就使用本节内容进行项目开发，而不使用MVVM技术。以下动态加载方式均使用常用的Resources加载。

- 学习目标：掌握面板的管理

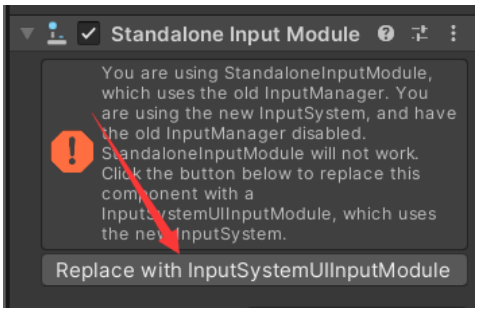
1）先准备一个Panel Prefab，这里我们简单创建一个Text和Image，分别命名为Name和Head

- 由于使用了Unity新的InputSystem，因此Hierarchy的EventSystem要替换成新的，否则运行会报错

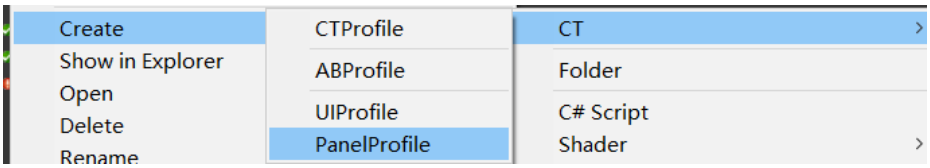


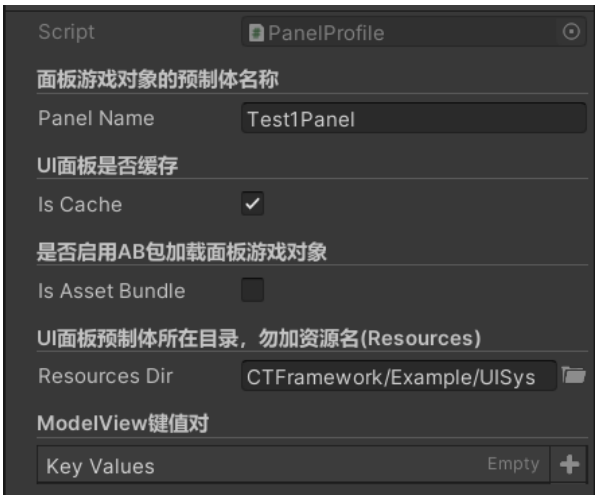
> 1.图片指出的两个路径很重要，一个是存放案例静态资源的（以后叫做静态路径），一个是用于动态加载资源的（以后叫做静态路径）。后面几个案例同理，不再单独指出

> - 案例的静态路径为：`Assets/CTFramework/Example/2.UISystem/`
> - 案例的动态路径为(从Resources算起)：`CTFramework/Example/UISys`
>
> 2.修改EventSystem，防止报错



2）在**动态路径**创建**面板配置文件**，修改名称为Test1PanelProfile





3）在**静态路径**创建脚本 Test1View.cs（最好按规范命名 xxxView.cs），将其挂载到Prefab根对象上（本例的Test1Panel）

```
public class Test1View : ViewComponent
{
  public const string PANEL_NAME = "Test1Panel"; //面板名称

  #region 案例1:管理面板
  private void Start() {}

  private void Update() {}
  #endregion
}
```

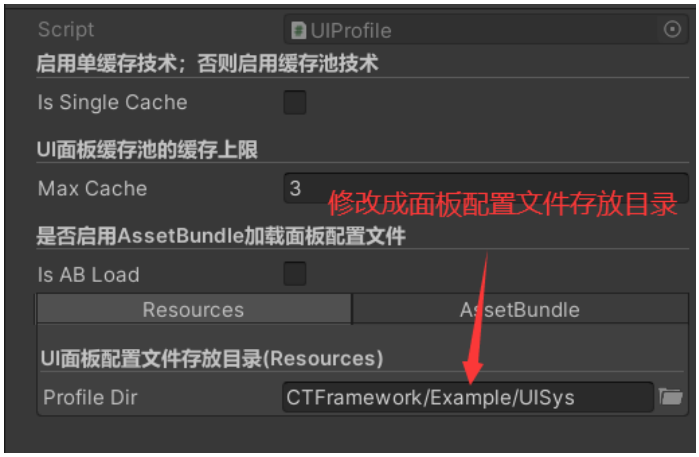
> 在UI系统的概述部分我曾说过，用户创建一个UI面板需要提供3个文件，至此3个文件全部准备完毕!

4）在**静态路径**编写测试脚本 UITest1.cs，挂载在任意已激活对象上

```
public class UITest1 : MonoBehaviour
{
```

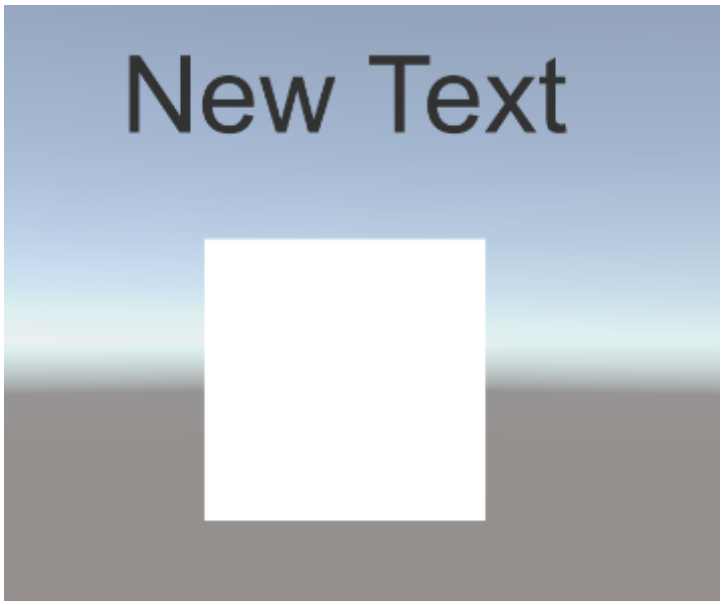
```
  private void Update()
  {
    //示例1
    Keyboard board = Keyboard.current;
    if (board.eKey.wasPressedThisFrame) //按下e，push一个面板
    {
      Debug.Log("按下e");
      //加载面板
      //_CT.UIMgr.Push("CTFramework/Example/UISys/", Test1View.PANEL_NAME);
      //如果设置了UIProfile配置文件的默认加载方式，可以直接传入面板名称
      _CT.UIMgr.Push(Test1View.PANEL_NAME);
    }
    else if (board.qKey.wasPressedThisFrame) //按下q，pop一个面板
    {
      Debug.Log("按下q");
      _CT.UIMgr.Pop();
    }
  }
}
```

> 修改UIProfile，优化配置文件的加载方式：
>
> - 目前仅支持单路径加载，这对于中小型游戏而言足够了



5）运行结果

- 按下E键就能压入一个新面板，Q键弹出顶层面板



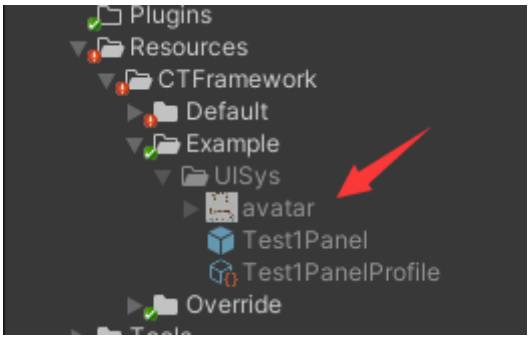
> 本案例没有使用到MVVM的技术，也没有写任何UI交互逻辑。你可以自行在xxxView.cs中编写UI的交互逻辑，ViewComponent本身继承自MonoBehaviour，因此所有操作和平常编写UI脚本时没有任何区别，这仅仅是加入了一个管理类，协助你管理UI面板。
>
> 注意：如果引入面板缓存且面板属于破坏性面板（即在整个生命周期中有增删面板内容），需要在OnEnter和OnExit中做数据的初始化和释放工作（框架底层仅仅对面板进行了激活和关闭GameObject的操作）

示例2.使用MVVM

介绍：本案例会教你如何利用 `Key` 绑定数据和UI组件。我们会在运行时修改文字内容和图片信息。使用起来真的非常简单，多说无益，让我们开始吧！

学习目标：掌握MVVM

1）随便找几张图片素材（比较懒，找一张），放到Resources下（随便放，记住路径即可）



1）修改案例1的 Test1View.cs 中的Test1View类

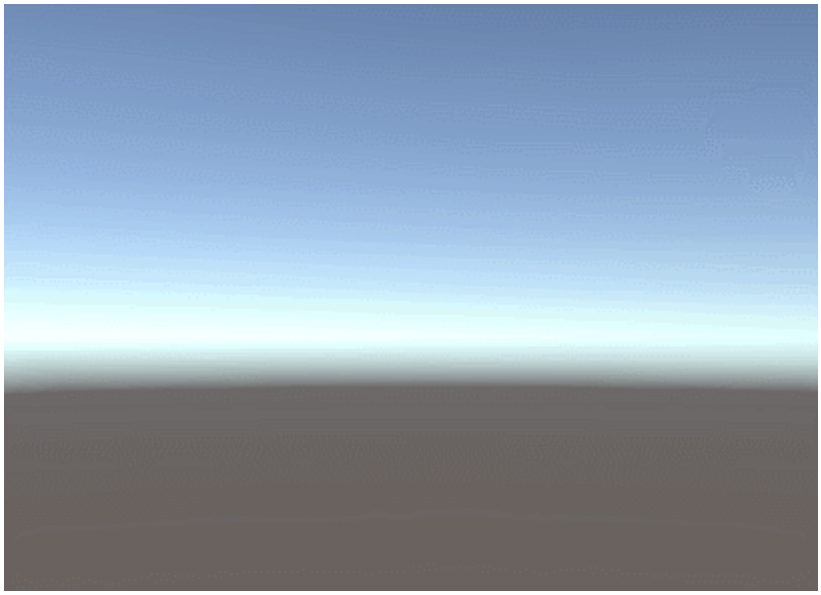
```
/*****************************************************
  文件：Test1View.cs
  作者：聪头
  邮箱：1322080797@qq.com
  日期：2022/4/18 12:19:41
  功能：
*****************************************************/
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using CT;
using CT.UISys;

public class Test1View : ViewComponent
{
  public const string PANEL_NAME = "Test1Panel"; //面板名称

  public Text txt;
  public Image image;

  #region 案例2:使用MVVM

  public override void OnInit()
  {
    base.OnInit();
    txt = transform.Find("Name").GetComponent<Text>();
    image = transform.Find("Head").GetComponent<Image>();

    //绑定监听
    controller.AddStringAfterListener("Name", v => txt.text = v);
    controller.AddStringAfterListener("Image", v => image.sprite =
Resources.Load<Sprite>(v));
  }

  public override void OnEnter(object obj)
  {
    //设置Key-Values(避免此时触发事件)
    controller.SetString("Name", "聪头", false).isTrigger = true;
    controller.SetString("Image",
"CTFramework/Example/UISys/avatar",false).isTrigger = true;

    base.OnEnter(obj); //渲染所有数据(最后调用)
  }

  #endregion

}

```

3）在**静态路径**编写测试脚本 UITest2.cs，挂载在任意已激活对象上。如果是新场景，记得挂_CT.cs

```
public class UITest2 : MonoBehaviour
{
  private void Update()
```

```
    {
        //示例1
        Keyboard board = Keyboard.current;
        if (board.eKey.wasPressedThisFrame) //按下e，push一个面板
        {
            //加载面板
            _CT.UIMgr.Push(Test1View.PANEL_NAME);
        }
        else if (board.qKey.wasPressedThisFrame) //按下q，pop一个面板
        {
            _CT.UIMgr.Pop();
        }
        //示例2新增代码！！！
        else if (board.aKey.wasPressedThisFrame) //按下a
        {
            _CT.UIMgr.SetString("Name", "我按下了a"); //修改文本内容
            _CT.UIMgr.SetString("Image", "CTFramework/Example/UISys/avatar");
        }
        else if (board.dKey.wasPressedThisFrame) //按下d
        {
            _CT.UIMgr.SetString("Name", "我按下了d");
            _CT.UIMgr.SetString("Image", ""); //直接给空路径，加载为null
        }
    }
}
```

4）测试结果



> 至此，MVVM技术就介绍完毕了，不知道你懂了没？反正只需要一个Key，就可以完成所有的一切！当然，这里还有个小的细节问题。Key目前每次访问都是手动编写字符串，一不小心很容易出错。这里给出的建议是，单独用一个文件或在xxxView.cs中，以常量的形式存放所有的Key，这样在访问时就不容易出错啦！lll
>
> 不过，还是感觉好麻烦啊，每次创建一个面板，又要创建配置文件，又要创建脚本，绑定UI。虽然自由度已经很高了，但是还有不少规则要遵守。如果你有这种感觉，那就对了，我也有！因此，我将使用一个工具帮各位自动完成大部分工作，我们要做的就是写写Key-Value，写写监听，其他都交给编辑器帮我们完成。怎么样，如果你心动的话，请不要走开，下一节更精彩！

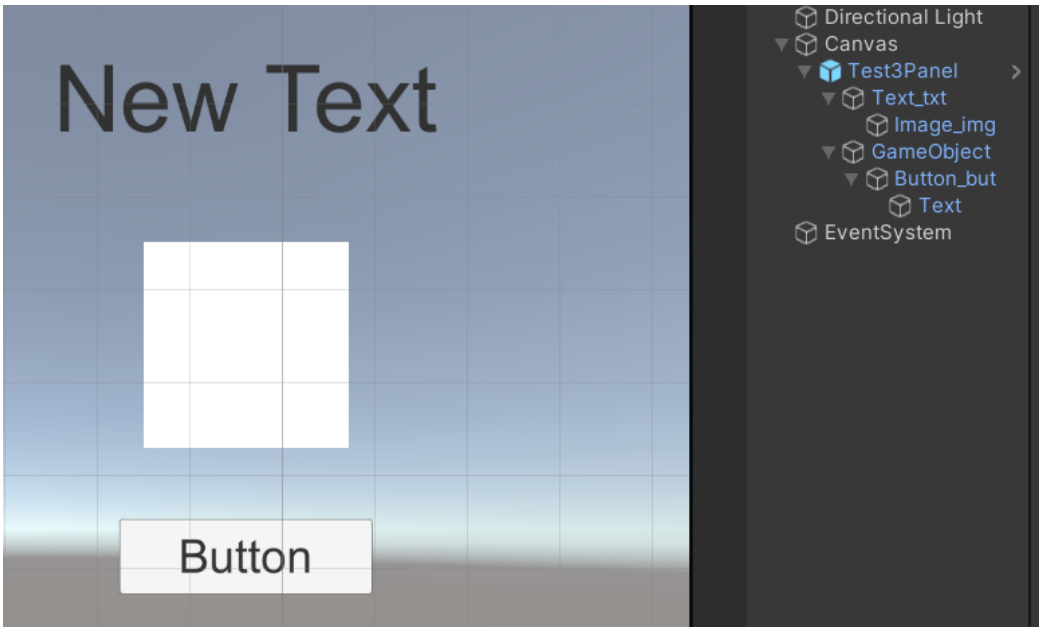
示例3.UI自动化

介绍：本案例会教你如何使用UI自动化工具，只需进行简单配置，即可生成所需文件，助你的UI开发一臂之力！

学习目标：掌握UI自动化工具的使用

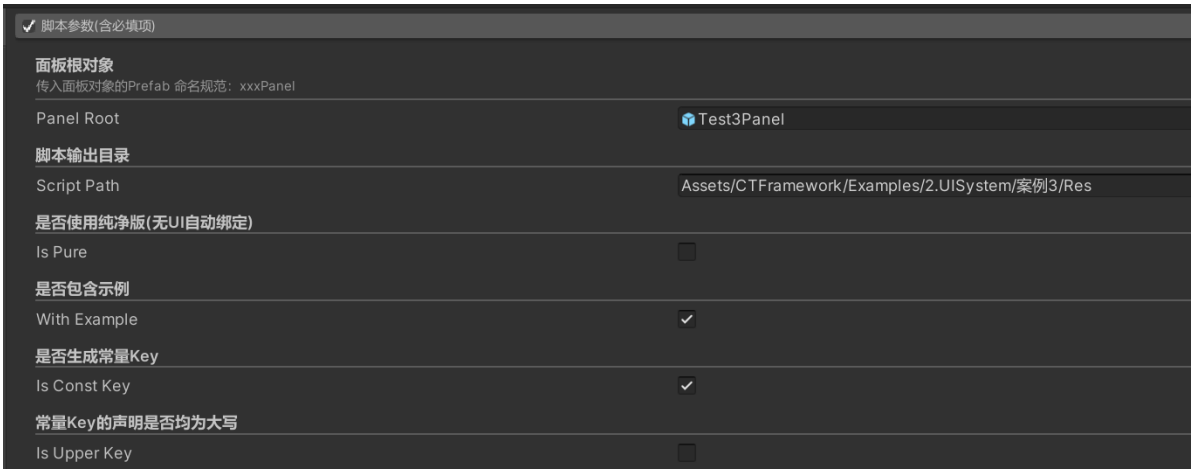
1）创建一个新场景，设计如下Panel，拖拽到**动态路径**中生成Panel Prefab

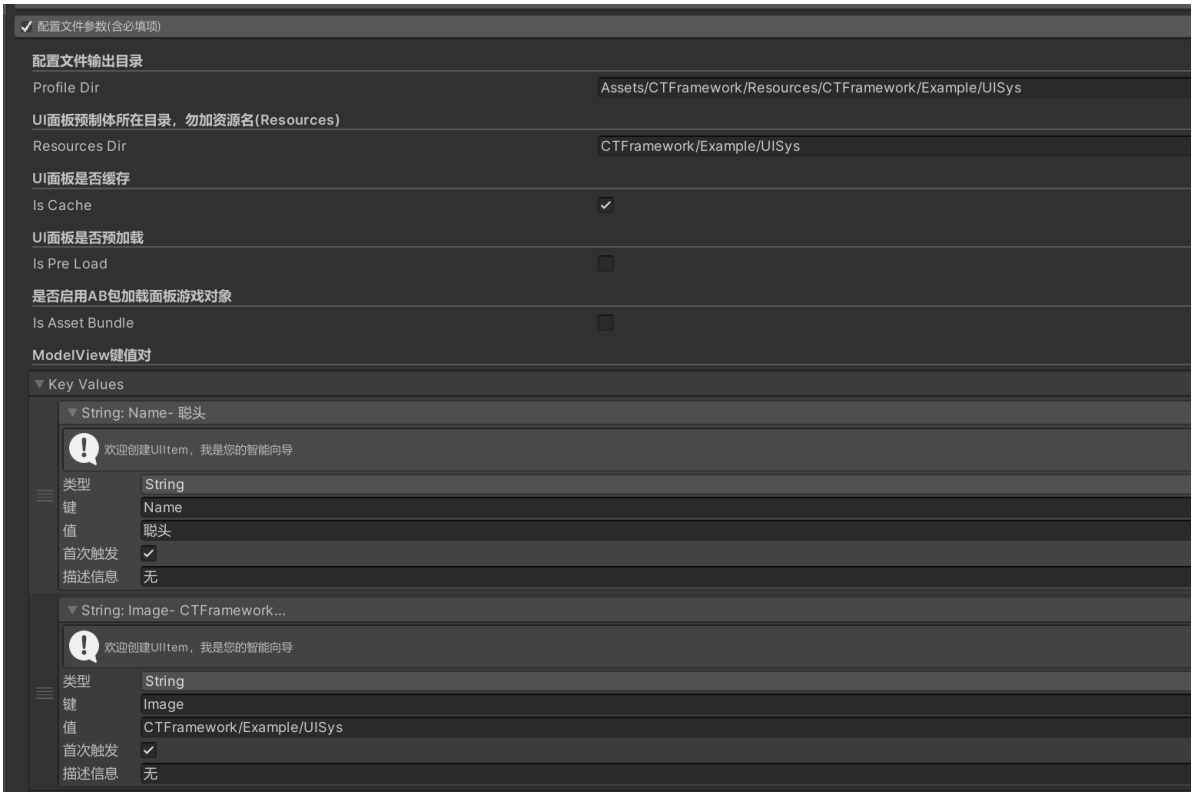
- 这里我们故意把层级关系写乱点，方便等等验证查找算法的高效性
- 需要注意绑定控件的命名方式，`名称_后缀`（后缀的命名要求详见Q2）



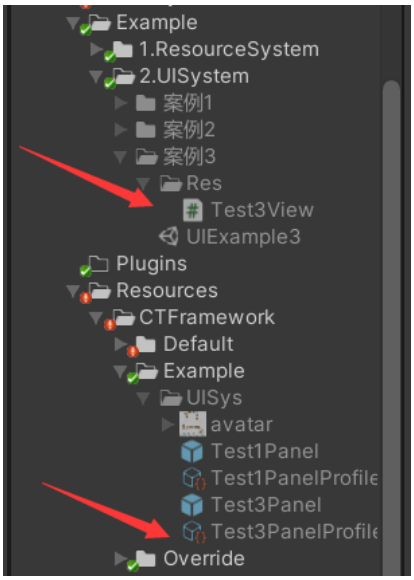
2）菜单栏选择 `聪头框架->工具箱->UI一键生成`，根据需要进行配置

- 具体参数已经在面板解释得很清楚了，这里直接给出我的设置

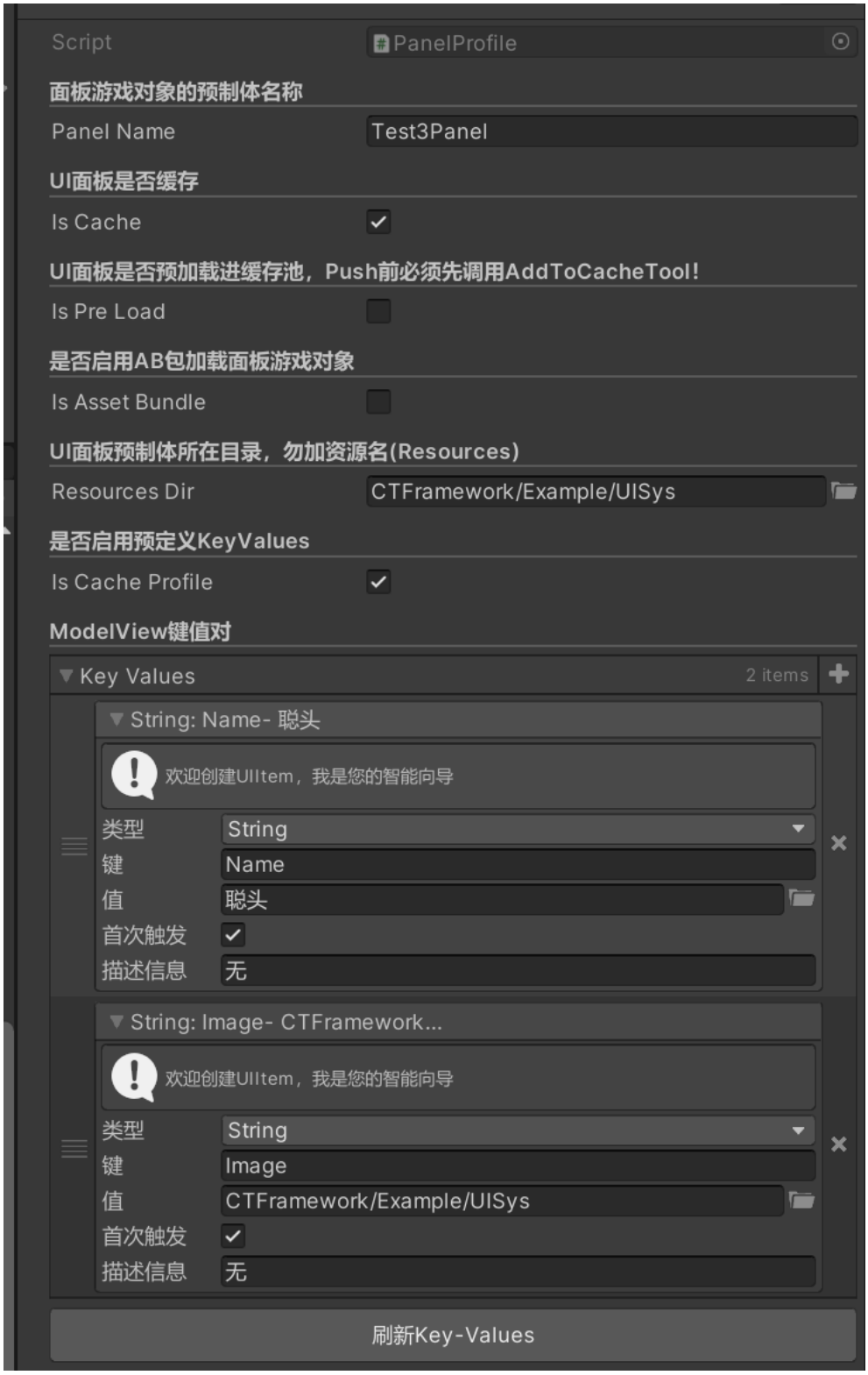




3）点击生成后，会发现多了一些文件，我们具体看看文件的内容吧！



> 首先是**自动化**生成的PanelProfile，可以看到这就是之前在UI自动化工具中设置的参数



> 接着是**自动化**生成的 xxxView.cs，可以看到一些基本信息已经自动生成完毕，我们只需要负责编写监听即可。生成的代码是包含示例的，根据需要，你可以在UI自动化工具中禁用示例

- 以下配置仅供展示，实际开发过程中无需编写

```
public class Test3View : ViewComponent
{
  #region 字段声明
  public const string PANEL_NAME = "Test3Panel"; //面板名称
  //声明常量Key
  public const string Key_Name = "Name";
  public const string Key_Image = "Image";

```

```
    //声明UI组件
    //public Text text;
    public Text Text;
    public Image Image;
    public Button Button;

    #endregion

    #region UI绑定
    //时机：整个生命周期仅初始化时执行一次
    //任务：查找组件 + 设置全局Key-Values + 绑定监听
    public override void OnInit()
    {
        base.OnInit();
        Text = transform.Find("Text_txt").GetComponent<Text>();
        Image = Text.transform.Find("Image_img").GetComponent<Image>();
        Button =
transform.Find("GameObject").Find("Button_but").GetComponent<Button>();

        //示例：绑定监听
        //controller.AddStringAfterListener("title", v => title.text = v); //绑定
Text的监听
    }

    //时机：面板每次Push时执行(整个生命周期可多次)
    //任务：完成数据初始化工作
    public override void OnEnter(object obj)
    {
        //示例：设置普通Key-Value(如果之前添加了监听，这里修改值时要关闭触发器)
        //controller.SetString("title", "标题", false).isTrigger = true;

        controller.InvokeAll(); //渲染所有数据(最后调用)
    }
    #endregion

}
```

- 细心的同学可能会发现在OnFindUI查找的组件时，会自动寻找邻近父组件开始Find，理论上说确实可以提升程序性能，但是万一父组件为null，不就报错了吗？问得好，但我可以负责的告诉你，无须担心。这是因为底层代码是基于树的前序遍历，当访问子组件时，父组件一定会先声明并赋值

4）至此，我们只要参考案例2，完善 `Test3View.cs` 脚本，挂载 `_CT.cs` 脚本，编写 `UITest3.cs` 测试类

- 完善Test3.View.cs

```
public override void OnInit()
{
  base.OnInit();
  Text = transform.Find("Text_txt").GetComponent<Text>();
  Image = Text.transform.Find("Image_img").GetComponent<Image>();
  Button = transform.Find("GameObject").Find("Button_but").GetComponent<Button>();

  //示例：绑定监听
  //controller.AddStringAfterListener("title", v => title.text = v); //绑定Text的监听
  controller.AddStringAfterListener(Key_Name, v => Text.text = v);
  controller.AddStringAfterListener(Key_Image, v => Image.sprite = Resources.Load<Sprite>(v));
}
```

- 测试类完整代码

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

public class UITest3 : MonoBehaviour
{
  private void Update()
  {
    //示例1
    Keyboard board = Keyboard.current;
    if (board.eKey.wasPressedThisFrame) //按下e，push一个面板
    {
      //加载面板配置文件
      //如果设置了UIProfile配置文件的默认加载方式，可以直接传入PanelEnum生成面板
      //_CT.UIMgr.Push("CTFramework/Example/UISys/", PanelEnum.Test3Panel);
      _CT.UIMgr.Push(Test3View.PANEL_NAME);
    }
    else if (board.qKey.wasPressedThisFrame) //按下q，pop一个面板
    {
      _CT.UIMgr.Pop();
    }
    //示例2新增代码！！！
    //示例3在此基础上做了Key的优化
    else if (board.aKey.wasPressedThisFrame) //按下a
    {
      _CT.UIMgr.SetString(Test3View.Key_Name, "谢谢观看"); //修改文本内容
      _CT.UIMgr.SetString(Test3View.Key_Image, "CTFramework/Example/UISys/avatar");
    }
    else if (board.dKey.wasPressedThisFrame) //按下d
    {
      _CT.UIMgr.SetString(Test3View.Key_Name, "完结撒花");
      _CT.UIMgr.SetString(Test3View.Key_Image, ""); //直接给空路径，加载为null
    }
```

```
36      }
37  }
38
```

5）完结撒花

> 至此，恭喜你，学完了整个UI模块的基础内容！其实整个UI系统并不复杂，文档写得算比较详细了，看起来内容可能会有点多。当然，后期我也会不断维护，改进框架，完善文档！加油！

答疑解惑

Q1.配置文件的范围指什么?

答：范围指的是该配置文件的影响力。

- 全局有效：配置文件是依赖某个系统的，即在整个系统运行周期内有效
  - **全局有效**的配置文件路径（默认版本）：`Resources/CTFramework/Default/` 下
  - **全局有效**的配置文件路径（重载版本）：`Resources/CTFramework/Override/` 下
- 局部有效：配置文件不依赖于系统，而依赖于具体对象（即产品）。例如UI系统，每当创建一个新面板就需要创建一个PanelProfile，当面板运行时需要PanelProfile提供数据支持。因此，我们就说PanelProfile是局部有效的
  - 文件的路径根据加载方式自定义
- 初始化有效：配置文件主要为系统的初始化工作提供参数，初始化后被丢弃。这里可以详细再分全局和局部，但是实际使用过程中很少使用此方式，因此不做区分。
  - 文件路径和全局有效的路径一致

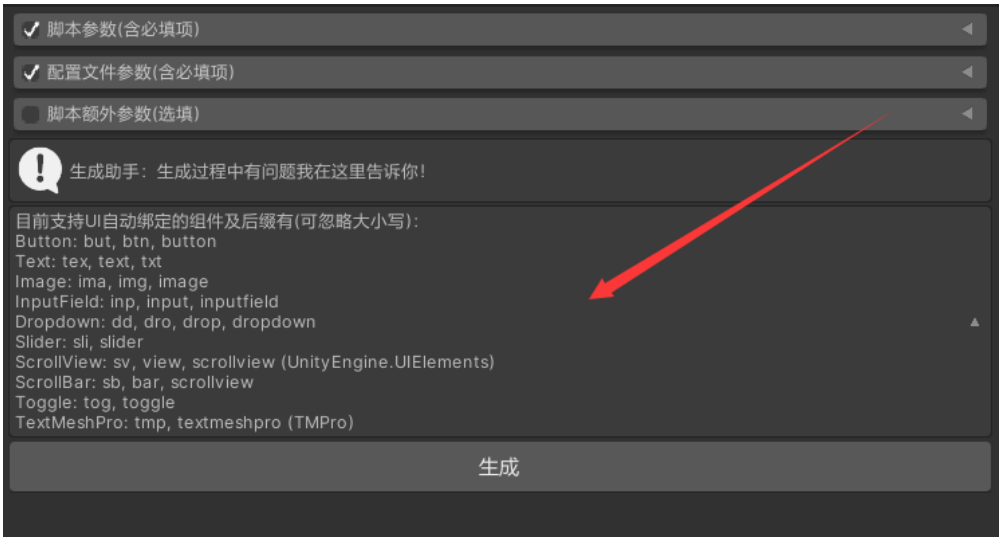
Q2.自动化面板的后缀要求是?

答：UI自动化并非支持所有的UI控件，所有参与自动化的UI控件都要符合命名规范要求：

> 名称_后缀

- 名称：指的是UI名称，没有限制，但通常需要具备高可读性。底层会利用名称生成变量名
- 后缀：目前支持UI自动绑定的组件都需要写明后缀（忽略大小写）。底层会利用后缀识别类型信息，确保查找UI时不出错。下面列出所有支持的UI组件及后缀：

类型	后缀
Button	but, btn, button
Text	tex, txt, text
Image	ima, img, image
InputField	inp, input, inputfield
Dropdown	dd, dro, drop, dropdown
Slider	sli, slider
ScrollView	sv, view, scrollview
ScrollBar	sb, bar, scrollbar
Toggle	tog, toggle
TextMeshProUGUI	tmp, textmeshpro

> 后缀记法：①全称基本没有问题，但是命名累；②首字母前3个（除TextMeshPro，ScrollView和Bar 外）；③常用缩写形式（如img, sv）
>
> 如果实在记不住可以打开点击UI自动化工具，里面有提示选项卡：
>
> 
>

3.音乐系统

音乐概述

说明：音乐系统提供一系列**2D音乐**的解决方案，把括指定播放背景音乐、环境音、音效，随机播放音乐音效等。整体设计思路和UI系统大致相同，但没有缓存池，没有堆栈，设计起来相对简单。音乐系统引入了一个类似BasePanel的概念，叫做组（Group），在游戏运行过程中，加载器会持有一个组，所有播放的音乐都会来自这个组中

音乐配置文件

工厂配置文件

概述：描述音乐组配置文件的默认加载方式

范围：全局有效

所在类名：AuProfile

字段	说明
isABLoad	是否启用AB包加载该配置文件
profileDir	Resources加载时的路径
abName	AB包加载时的包名

音乐组配置文件

概述：描述UI工厂的缓存策略和生产产品的方式

范围：初始化时有效

所在类名：AuGroupProfile

字段	说明
isAssetBundle	是否启用AB包加载AudioClip
resourcesDir_bg	如果使用Resouces加载，背景音乐所在Resources目录
resourcesDir_effect	如果使用Resouces加载，音效所在Resources目录
abName_bg	如果使用ab包加载，背景音乐所在ab包
abName_effect	如果使用ab包加载，音效所在ab包
bgs	背景音乐集合
effects	音效集合

背景音乐或音效集合内容项参数：

字段	说明
clipKey	音乐剪辑Key，用于唯一标识。同集合的Key不可重复
clipName	音乐剪辑名称，用于动态加载
volumn	音量
loop	是否循环播放，仅背景音乐有效
randomGroupID	随机组ID，随机播放音乐或音效时，指定随机组ID就可以在同组内随机播放。播放时，传入-1代表组内全部元素随机播放
isPreload	是否预加载。启用后，在加载阶段会提前将AudioClip加载进内存。通常来讲，背景音乐不宜开启，而音效推荐开启。
isCache	是否缓存。启用后，播放完的音乐或音效会缓存
isAsync	是否异步加载

音乐系统接口

IAudio 接口

概述：提供音乐组的创建，音乐音效的播放、暂停等

```
public interface IAudio : IRelease
{
  //设置音乐组
  //参数①：音乐组配置文件名称
  void SetGroup(string groupName);

  //设置音乐组
  //参数①：音乐组配置文件Resources加载目录或ab包名
  //参数②：音乐组配置文件名称
  void SetGroup(string resDir, string groupName);

  //根据key播放音乐
  //参数①：clipKey，musicDict对应的key
  //参数②：volumn，音量
  //参数③：是否是环境音
  void Play(string clipKey, float volumn, bool isEnv = false);

  //根据key播放音效（不会影响当前AudioSource）
  //参数①：clipKey，effectDict对应的key
  //参数②：volumn，音量
  void PlayOneShot(string clipKey, float volumn);

  //在随机组内随机播放音乐
  //参数①：volumn，音量
  //参数②：随机组ID(-1代表全部随机）
  void PlayRandom(float volumn, int randomID = -1);

  //随机播放音效（不会影响当前AudioSource）
  //参数①：volumn，音量
```

```
    //参数②：随机组ID(-1代表全部随机)
    void PlayOneShotRandom(float volumn, int randomID = -1);

    //暂停播放
    //参数①：是否是环境音
    void Pause(bool isEnv = false);

    //继续播放
    //参数①：是否是环境音
    void Resume(bool isEnv = false);

    //停止播放
    //参数①：是否是环境音
    void Stop(bool isEnv);

    //得到AudioSource
    //参数①：是否是环境音
    AudioSource GetAudioSource(bool isEnv);

    //设置音量（如果有，该音量会和系统音量做乘积）
    //参数①：音量值
    void SetVolume(float volume, bool isEnv);

    //获取音量大小（直接返回当前Source的音量值）
    //参数①：是否是环境音
    float GetVolume(bool isEnv);

    //设置系统音量
    //参数①：音量
    //参数②：音量类型
    void SetSystemVolume(float volume, VolumeType type);

    //得到系统音量
    //参数①：音量类型
    float GetSystemVolume(VolumeType type);

    //得到音乐组配置文件的音量
    float GetProfileVolume(VolumeType type, string key);
}
```

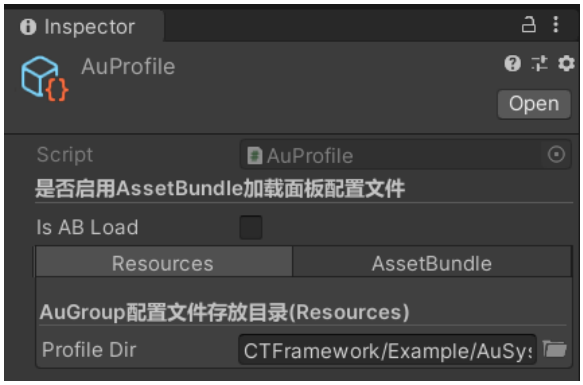
示例.音乐系统使用

介绍：本节将用一个实例让你弄懂整个音乐系统!

学习目标：掌握音乐系统的实验

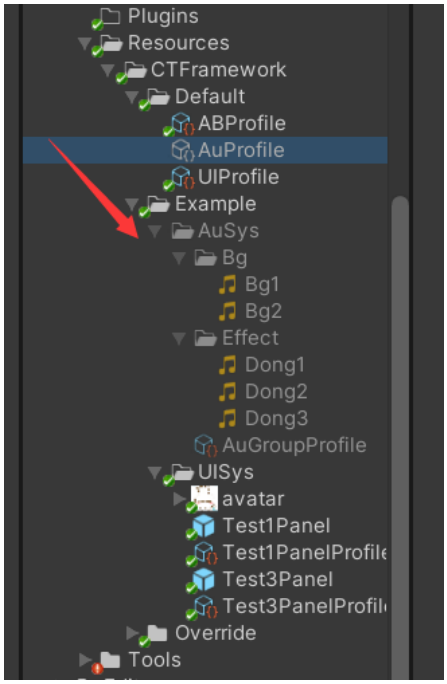
1）准备阶段，先配置好AuProfile，它是音乐系统最底层的配置项

- 位置：`Assets/CTFramework/Resources/Default/`
- 目的：配置AuProfile就是告诉工厂怎么生产AuGroupProfile

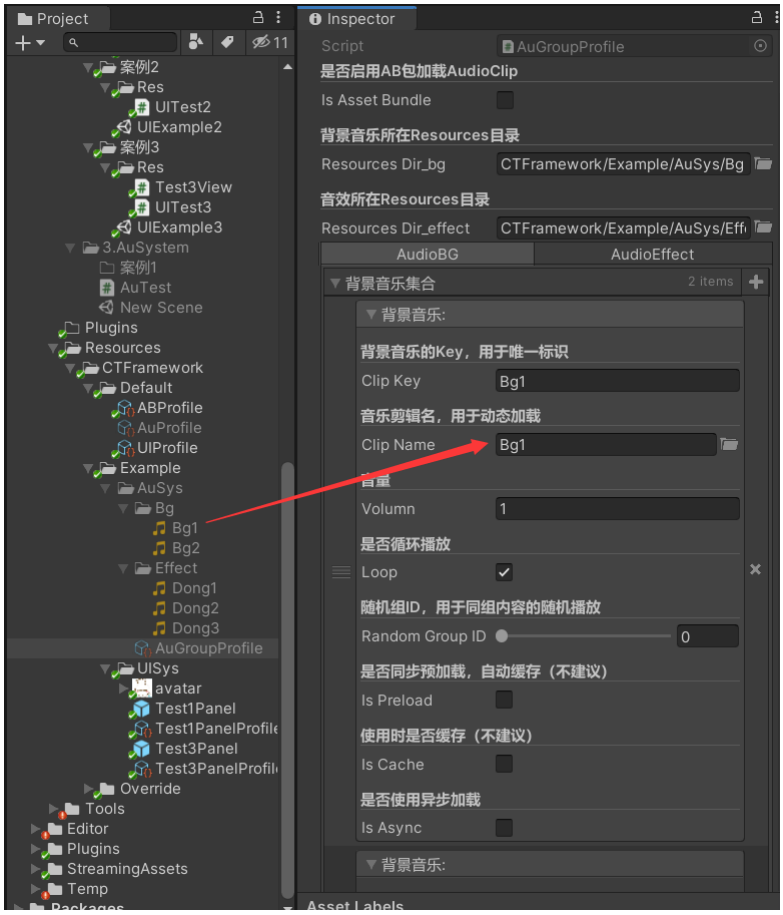


2）准备好音乐资源，创建并编写AuGroupProfile配置文件

- 本案例中，**音乐组配置文件**和所有**音乐资源**还是采用Resources的方式动态加载
  - 音乐资源位置：`Assets/CTFramework/Resources/CTFramework/Example/AuSys`



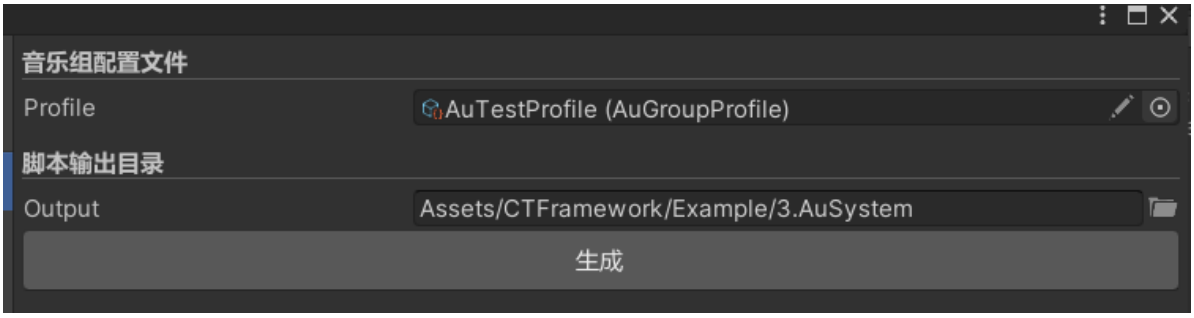
- 下面我们来看看具体怎么编写音乐组文件，见名知意，下面我就简单解释一下参数设置



- 首先我们使用Resources加载，所以不勾选`IsAsssetBundle`。按照要求，配置两个Resources目录。接着就可以创建背景音乐集合和音效集合。这里以`背景音乐Bg1`为例，我们只需要直接拖拽Bg1到ClipName后，即可自动赋值`ClipName`和`ClipKey`（二者不一定保持一致）。后面就一系列参数设定，不懂的童鞋可以看音乐组配置文件
- 后面的赋值操作我就不一一演示了，就是把剩下的音乐剪辑绑定到音乐组配置文件中

3）自动生成所有Key枚举，这一步是**可选**的，旨在使用时提高可读性和容错率

- 选择顶部菜单栏的`聪头框架->工具箱->AuKey一键生成`，简单配置后点击生成，会生成如下文件




```
namespace CT
{
  //AuGroup Key的枚举
  public partial class AuKeyEnum
  {
    //背景音乐的Key
    public const string AUTEST_BG1_BG = "Bg1";
    public const string AUTEST_BG2_BG = "Bg2";
    //音效的Key
    public const string AUTEST_DONG1_EF = "Dong1";
    public const string AUTEST_DONG2_EF = "Dong2";
    public const string AUTEST_DONG3_EF = "Dong3";
```

```
  }
}
```

4）准备完毕，将_CT拖拽到任意激活对象上，编写测试脚本`AuTest.cs`并拖拽到任意激活对象

- AuTest.cs内容如下：

```
public class AuTest : MonoBehaviour
{
  private void Start()
  {
    //设置音乐组
    _CT.AuMgr.SetGroup("AuTestProfile");
    //播放Bg2
    _CT.AuMgr.Play(AuKeyEnum.AUTEST_BG1_BG, 1.0f);
  }

  private void Update()
  {
    Keyboard board = Keyboard.current;
    //按下Q会切换到Bg2
    if (board.qKey.wasPressedThisFrame)
    {
      _CT.AuMgr.Play(AuKeyEnum.AUTEST_BG2_BG, 1.0f);
    }
    //按下W会切换到Bg1
    else if (board.wKey.wasPressedThisFrame)
    {
      _CT.AuMgr.Play(AuKeyEnum.AUTEST_BG1_BG, 1.0f);
    }
    //按下E会播放音效Dong1
    else if (board.eKey.wasPressedThisFrame)
    {
      _CT.AuMgr.PlayOneShot(AuKeyEnum.AUTEST_DONG1_EF, 1.0f);
    }
    //按下R会随机播放音效
    else if (board.rKey.wasPressedThisFrame)
    {
      //第二个参数默认为-1，代表从所有音效中随机选择
      //指定相应随机组，则只在相应随机组中选择
      _CT.AuMgr.PlayOneShotRandom(1.0f);
    }
    //按下T只播放随机组1的音效
    else if (board.tKey.wasPressedThisFrame)
    {
      _CT.AuMgr.PlayOneShotRandom(1.0f, 1);
    }
    //按下A暂停播放
    else if (board.aKey.wasPressedThisFrame)
    {
      _CT.AuMgr.Pause();
    }
    //按下S继续播放
    else if (board.sKey.wasPressedThisFrame)
    {
```

```
            _CT.AuMgr.Resume();
        }
    }
}
```

> 音乐系统，完结撒花！别忘了到游戏中实战一番哦，相信聪明的你很快就能掌握啦~！

工具集

概述：提供了一系列非常实用的工具供用户使用，掌握常用的工具并灵活运行往往能在项目开发时达到事半功倍的效果

---------- 补充：框架 v0.2----------

1.UI系统

示例4.自动绑定属性

修改示例3代码如下

> 修改特性

```
//声明UI组件
//public Text text;
[UIAutoBind(Key_Name)]
public Text Text;
[UIAutoBind(Key_Image)]
public Image Image;
public Button Button;
```

> 调用自动绑定监听的函数

```
public override void OnInit()
{
    base.OnInit();
    Text = transform.Find("Text_txt").GetComponent<Text>();
    Image = Text.transform.Find("Image_img").GetComponent<Image>();
    Button =
transform.Find("GameObject").Find("Button_but").GetComponent<Button>();

    //示例：绑定监听
    //controller.AddStringAfterListener("title", v => title.text = v); //绑定
Text的监听
    //controller.AddStringAfterListener(Key_Name, v => Text.text = v);
    //controller.AddStringAfterListener(Key_Image, v => Image.sprite =
Resources.Load<Sprite>(v));
    this.AutoBindUI();
}
```

运行结果和示例3完全一致

结语

我发现设计模式这种东西没有太大必要刻意去记，它是一种思维，一种主观存在的东西。我发现在开发框架时普遍存在的现象：不是你学了这个设计模式去应用于框架，而是你先应用之后，才明白原来用到了某某设计模式。所以对于思维的学习，要循序渐进，灵活应变，多思考，多实践。没必要刻意说我用了什么设计模式，或刻意按照某种设计模式去组织代码，这样反而使思维更加僵化。我始终认为，编程的最高境界不是你写得语句多么nb，多么高大上，恰恰相反，而是多么精简，多么优雅。看来，编程是一门艺术！