将飞龙射击工程拷贝一份，作为热更实践案例。

#### Unity资源管理插件Addressables

Unity的Asset Bundle（教材里有介绍）设计比较简陋，在其基础上开发“热更新”架构是一件比较复杂的工作。所以近年来Unity推出了Addressables插件，用于替代原始的资源管理系统。

Addressables插件主要解决以下问题：

1. 资源单独打包，并列出资源列表。
2. 动态资源加载功能。
3. 必须适用于热更新、资源分版本更新等各种实际情况。

Addressable的几种使用模式：

1. 尽量不影响开发流程，通过资源引用“AssetReference”的方式尽量不改变开发流程。
2. 由开发者仔细安排更新流程，比如在Loading界面加载资源，或者在游戏一开始更新所有资源。

有了Addressable能减少一些开发负担，但是对于初学者来说，理清动态资源管理的思路依然是一项很难抓住头绪的工作（主要原因是资源管理涉及的流程较多，需要实践经验）。

接下来会用多个小例子让大家了解Addressable的多种功能。

Addressables安装方法：在Package Manager中安装。

如果VS代码识别出问题，可以在Preference -> External Tools里重建VS工程。

#### Addressable测试——主角换皮肤

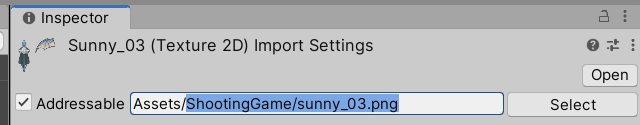
Addressable最简单的使用方法是使用资源引用“AssetReference”。资源引用不是真的资源，但通过它可以去更新到实际的资源。

比如我们给主角飞龙新增一个脚本组件“PlayerSprites”，目的是动态改变主角的皮肤（图片）。

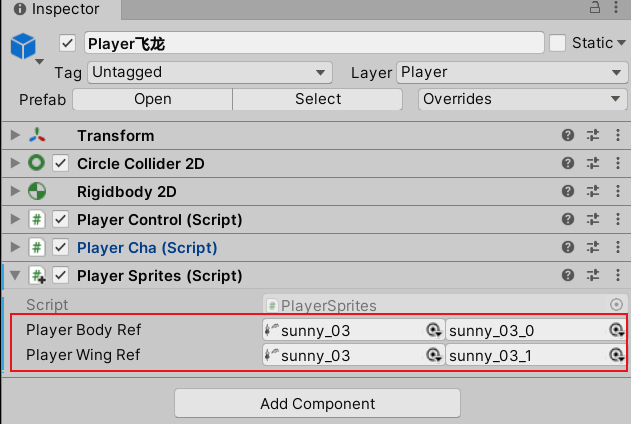
1. 编写换肤脚本：

|  |
| --- |
| using UnityEngine;  using UnityEngine.AddressableAssets;  public class PlayerSprites : MonoBehaviour  {  public AssetReferenceT<Sprite> playerBodyRef;  public AssetReferenceT<Sprite> playerWingRef;  void Start()  {  SpriteRenderer playerBody = transform.GetChild(0).GetComponent<SpriteRenderer>();  SpriteRenderer playerWing1 = transform.GetChild(1).GetComponent<SpriteRenderer>();  SpriteRenderer playerWing2 = transform.GetChild(2).GetComponent<SpriteRenderer>();  playerBodyRef.LoadAssetAsync().Completed += (sp) => { playerBody.sprite = sp.Result; };  playerWingRef.LoadAssetAsync().Completed += (sp) => {  playerWing1.sprite = sp.Result;  playerWing2.sprite = sp.Result;  };  }  } |

1. 将需要动态加载的图片标记为Addressable：



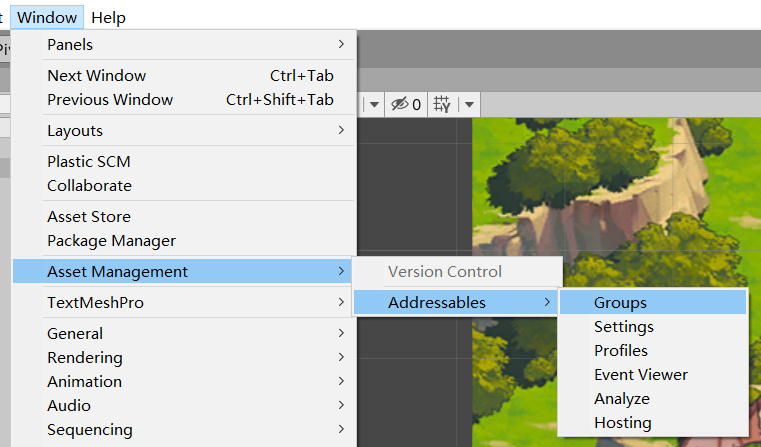
1. 给主角挂载脚本，并设置好AssetReference。

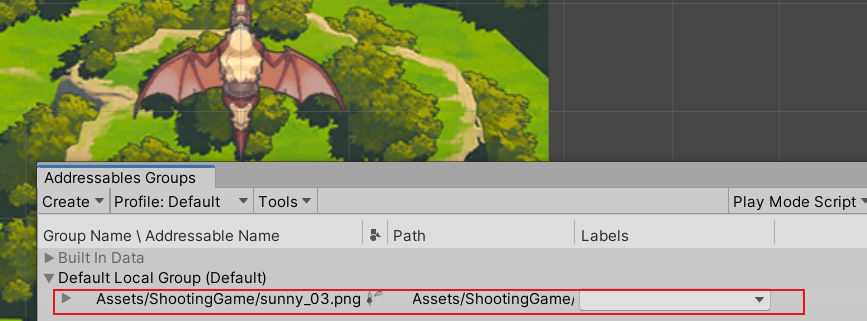


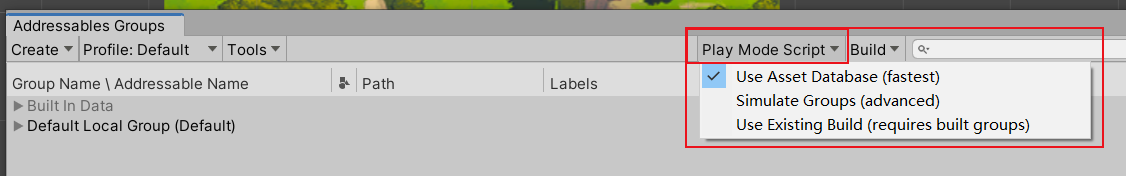
playerBodyRef.LoadAssetAsync()是一个异步方法，测试中发现大概到第6帧左右，皮肤才会被换过来。一般在进入主场景的时候，皮肤已经被下载到磁盘中，替换会比较快。

#### 3、Addressables资源组（Group）窗口

标记为“Addressable”的资源，会出现在资源列表里。用下面的方式打开资源group窗口：



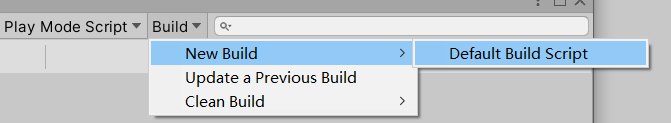




在Play Mode Script标签中，可以选择三种测试模式：

1. 使用内置的资产缓存，开发期适用。
2. 模拟资源组。
3. 使用已经存在的资源包（需要提前构建资源包）。

第一种适合开发期使用，第三种是与发布版本保持一致的流程。如果选择第三项，则需要点击Build提前创建资源包。



#### 4、加载玩家预制体

删除场景中的玩家物体，把它整个都动态加载出来。通用的实例化预制体脚本：

|  |
| --- |
| using UnityEngine;  using UnityEngine.AddressableAssets;  public class CreateGameObject : MonoBehaviour  {  public AssetReferenceGameObject prefabRef;  void Start()  {  if (prefabRef != null)  {  prefabRef.InstantiateAsync().Completed += (a) => {  a.Result.transform.position = transform.position;  };  }  }  } |

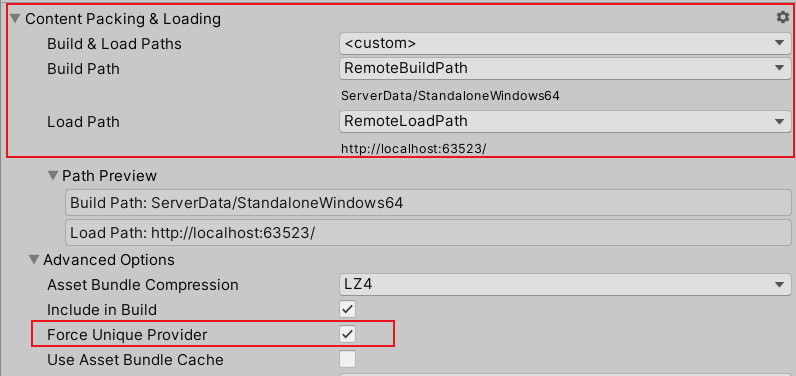
这个脚本可以实例化任意预制体到场景中。

既然整个预制体可以动态加载，那么关卡、敌人、主角均可以动态加载。

可以看出Addressables功能很强大。基本能够满足现代游戏开发动态更新的要求。

#### 5、试验从网络加载资源

将打包地址和加载地址都选择为Remote... 变成远程地址。

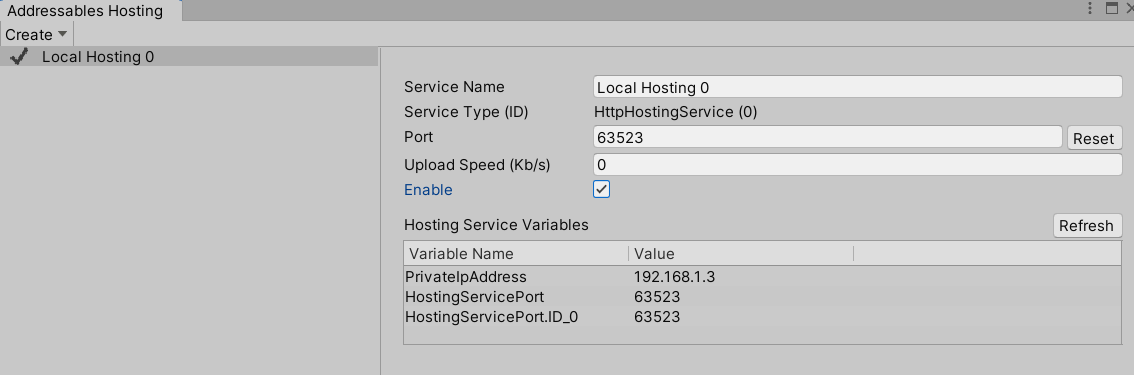


为了测试网络，有一个很重要的选项——高级选项中的Force Unique Provider，意思是强制只有一个资源来源。

**如果不勾选Force Unique Provider，Unity可能会从资源缓存中加载，达不到测试网络加载的目的。**

在上述设置后，加载应该会失败，因为我们并没有提供网络服务器（http服务器）。

为了方便测试，Unity为我们提供了一个好用的HTTP测试工具——Hosting。也在主菜单的Addressable选项里面。

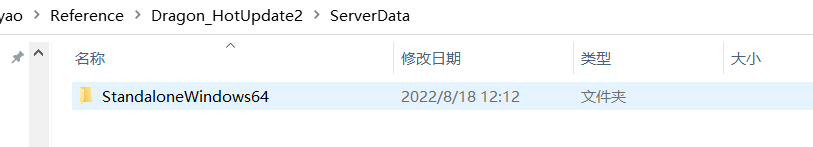


点击“Enable”就可以启动服务器了。

启动之后，可以用浏览器进行测试。但是会遇到最大的问题——路径。

#### 6、理解和解决路径问题

首先，确定ServerData目录中，打包资源是否准备好。



确认打包资源正常以后，开启Hosting，或者可以按Hosting的Reset按钮进行重启。

启动之后，用浏览器访问

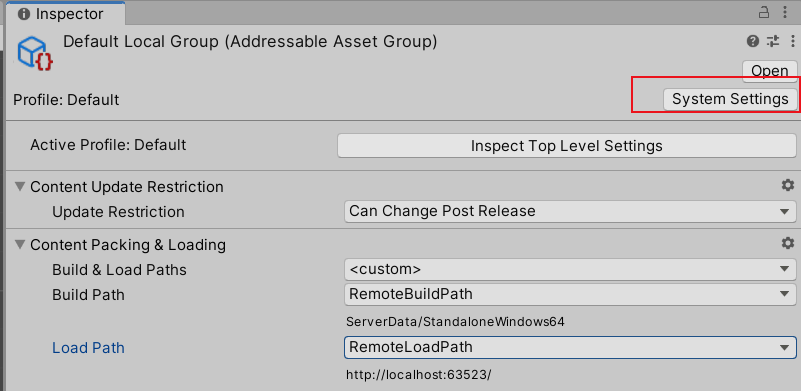
<http://localhost:63523/defaultlocalgroup_xxxxxxxxx.bundle>

格式说明：<http://localhost:[端口]/[bundle文件名]>

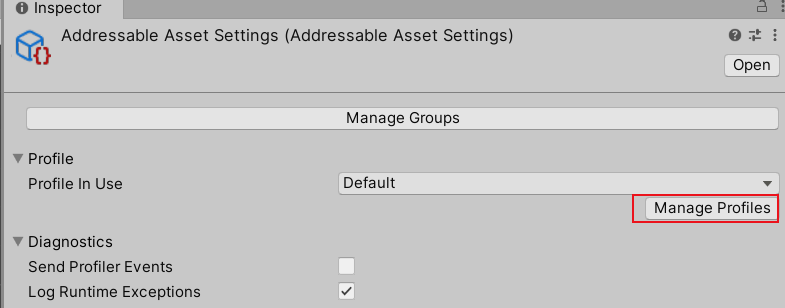
端口就是hosting的port。bundle文件名去ServerData里查。

通过浏览器访问的方法，可以确保Hosting本身是正常的。但是之后在Unity测试，还是会因为RemotePath产生路径问题。

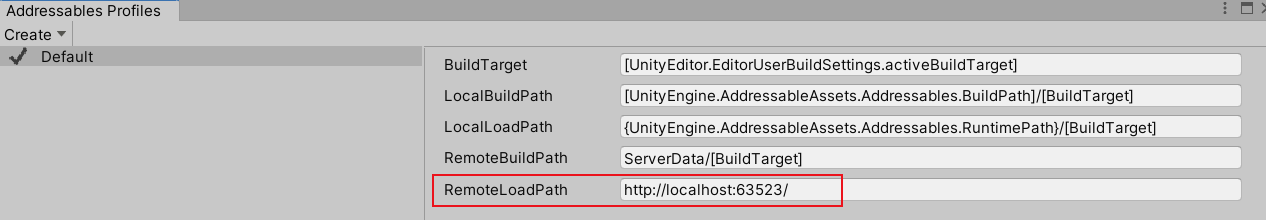
修改RemotePath的正确方法，是点击右上方的System Settings。



在设置中，点击Manage Profiles：



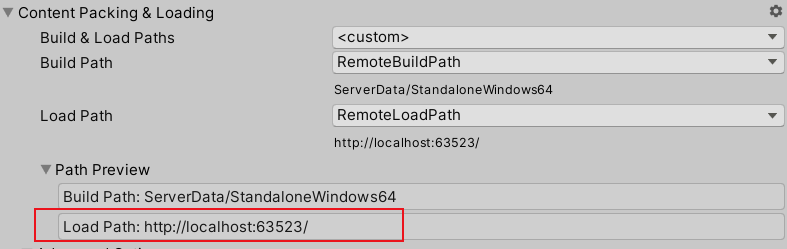
在Addressable Profiles窗口中修改RemoveLoadPath，即远程加载地址：



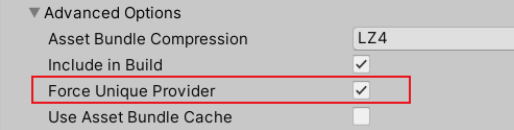
这里的格式为http://localhost:[端口]/

理论上，正式使用应该带上[BuildTarget]，[BuildTarget]的作用是把不同平台的资源包放在单独的目录里面。但是因为Hosting本身功能不全，没法设置，这里只好这么写。

回到Group设置，看看那个Load Path和你在网页浏览器中的是否一致，之后再次测试。



再次提醒：确保已经勾选 Force Unique Provider：



如果没有问题，应该可以直接在编辑器中测试http加载资源的效果。

#### 7、正反验证测试

一定要反复验证配置正确性。

1. 关闭hosting，即没有http服务器的情况下，资源加载应该失败，失败时要查看log。
2. 再次开启hosting，资源加载成功。
3. 将场景打包为可直接运行的游戏，再次测试hosting开启、关闭两种情况。

以上测试均通过，才能说明我们的设置没有问题。

**路径问题总结：**

老师在实验Addressable路径的过程中，也花费了数个小时的时间。

1. 路径问题是我们实际开发中的拦路虎，它的本质不复杂，但是只要有一点偏差，就会耗费开发者几小时、几天的宝贵时间。
2. 会看log，会定位问题，比做出来结果重要的多。
3. Addressable默认设置不完善，给了我们了解选项功能的机会。
4. 未来只要把ServerData中的资源包，全部拷贝到正式的HTTP服务器里，就实现了热更（例如阿里云OSS等）。这是服务器技术力不足的小团队或独立开发者的福音。

#### 8、使用资源名称加载资源

前面是使用AssetReference去加载资源，很方便。

但如果只需要指定资源名称就可以加载资源，那么管理会更灵活，普适性会更强。

关键函数：

|  |
| --- |
| Addressables.LoadAssetsAsync<TextAsset>(  new List<string> { "PlayerCharacter.lua", "PlayerController.lua" },  null,  Addressables.MergeMode.Union);  // 说明：LoadAssetsAsync中间的“Assets”是复数，代表它能加载多个资源  // 泛型<TextAsset>说明加载的一批资源必须是同一种类型。  // 第1个参数以List<string>的方式传入，代表一个到多个addressable资源名称  // 第2个参数是回调函数。如果不采用回调函数的使用方式，可以填null  // 第3个参数必填，否则容易错误匹配到其它函数重载……Union代表并集 |

通常，我们经常需要等待资源加载完毕再继续执行下一步逻辑。这里就有三种形式：

1. 回调函数方式：

|  |
| --- |
| Addressables.LoadAssetsAsync<TextAsset>(  new List<string> { "PlayerCharacter.lua", "PlayerController.lua" },   1. => { Debug.Log($”{a}加载完毕”)},   Addressables.MergeMode.Union); |

按这种写法，每加载完毕一个资源，执行一次回调函数。

2、现在的Unity引入了新的C#异步机制 async/await，由于此机制还在完善当中，同学们只要简单了解即可。

await等待：

|  |
| --- |
| private async void LoadAssets()  {  var load2 = Addressables.LoadAssetsAsync<TextAsset>(new List<string> { "PlayerCharacter.lua", "PlayerController.lua" }, null, Addressables.MergeMode.Union);  await load2.Task;  SceneManager.LoadScene(1);  } |

这种写法，函数本身必须用async修饰，需要等待的时候用await。

理论上这个写法比Unity协程要高级很多，但是它在运作原理方面存在一些问题。现在有越来越多的商业项目开始使用await/async。

3、Unity协程等待：

|  |
| --- |
| IEnumerator CoGameStart()  {  Debug.Log(Time.time);  yield return Addressables.LoadAssetsAsync<GameObject>(assetNames, null, Addressables.MergeMode.Union);  Debug.Log(Time.time);  } |