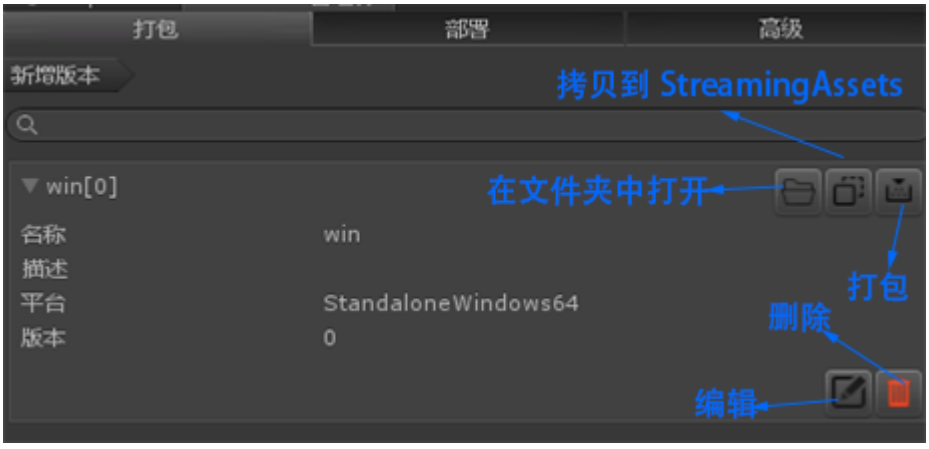
FinalPatch

1. 打包



1.1 新增版本

点击这个按钮来新建一个补丁版本. 版本指的是针对某个平台的游戏包, 例如你的游戏需要支持Windows, Mac, Android, 和IOS, 那么就**至少**需要分别建立这4个平台的版本.
如果你修改了C#代码, 在IOS平台或者使用IL2CPP功能时,你将无法热更新这部分代码, 此时你需要发布一个新的

游戏包. 为了让线上玩家下载新版本客户端, 你也需要在这里新增一个版本, 然后在部署页签中调整对应渠道的版本为这个新版本. 详情参见新增渠道.

1.2 在文件夹中打开

点击这个按钮可以在文件夹中浏览当前版本的所有资源.

1.3 拷贝到StreamingAssets

点击这个按钮可以将当前版本最新资源拷贝到StreamingAssets中. 这意味着此时你构建的游戏包体已经包含最新资源. 如果你希望玩家下载这个游戏包后不再需要更新额外资源, 就可以在Build步骤完毕后点击这个按钮. 但是这会导致你构建的游戏包尺寸增加, 有些游戏发布平台可能会限制游戏包体尺寸, 此时你可以考虑不要使用该功能, 即在发布游戏包体时不携带任何资源, 让玩家通过热更新来下载游戏资源.

注意: 若你希望在包体内包含资源,请务必只通过该按钮来操作,因为这会生成一些必要的数据. FinalPatch并不会直接使用你手动放进StreamingAssets目录的资源.

1.4 打包

点击这个按钮来进行打包. 生成的资源位于:`Application.dataPath/../AssetBundles/`,资源包含两部分:`Full`和`Package`.
`Full`中包含所有资源.
`Package`中是根据当前版本与上个版本的`Full`资源计算出的差异部分, 上传CDN时需要只需要关心`Package`内的资源即可.

注意: 虽然使用时只需关心`Package`部分, 但是`Full`也不可以删除, 因为`Package`的计算过程依赖上个版本的`Full`资源, 因此请至少在`Full`中保留最新的一个版本.

2. 部署

当某个版本的资源准备完毕后, 为了让线上玩家能够更新这些资源, 你需要将资源发布出去. 部署页签就是对资源进行发布和版本管理的地方.

2.1 部署数据

每个游戏有一个部署数据与之对应, 里面记录了该游戏所支持的所有渠道信息. 客户端请求热更新时会先下载这份数据, 然后根据自己的渠道来判断需要从哪个地址下载资源.
首次打开编辑器时需要创建或者选择一个部署数据, 点击对应按钮操作即可.

注意: 客户端热更新时需要下载这份数据,因此部署数据需要上传到CDN以供下载.

2.2 新增渠道

新增渠道时需要填写一些必要的参数.

- 渠道名称: 热更新时会查找与自己对应名称的渠道来比对版本和更新资源.
- 地址: 热更新时会从该地址更新资源.
- 版本: 指定客户端版本. 客户端不允许跨版本更新, 即若客户端当前版本为`android_1.0`,若这里将版本设置为`android_1.1`, 则会提示需要下载新版客户端.因此不要轻易修改版本, 因为这会导致玩家需要下载

新版客户端才能进入游戏.

- 版本号: 客户端会比对版本号来确定是否需要更新. 版本号比客户端高时客户端会升级到该版本, 比客户端低时客户端会降级到该版本.

注意: 热更新的实际URL为: 地址/版本/版本号/.
例如地址为:http://localhost:8000/AssetBundles. 版本为:demo. 版本号为:3. 则实际资源下载地址为:
http://localhost:8000/AssetBundles/demo/3/

2.3 编辑渠道

若有补丁发布时,点击编辑按钮后调整版本号即可.

2.4 删除渠道

请不要轻易删除一个渠道, 这会导致线上玩家无法进入游戏.

3. 设置

在高级页签中有一些设置项, 所有名称上有(?)的都表示有提示信息, 你可以让鼠标悬浮在名称上来查看这些信息.

3.1 打包

- 资源路径: 所有位于该路径下的资源都会被打包, 详情参见打包
- 提取重复依赖: 假设CubeA和CubeB都依赖同一个Material, 假设CubeA的大小为1KB, CubeB的大小为1KB, Material的大小为1KB, 假设我们只打包CubeA和CubeB,那么结果为: CubeA(1KB+1KB), CubeB(1KB+1KB), 即Material的内容在CubeA和CubeB的AssetBundle中都存在. 打开这个设置以后, 打包时检测到Material被多个AssetBundle依赖,所以被强制也打包成AssetBundle. 此时结果为: CubeA(1KB), CubeB(1KB), Material(1KB). 因此强烈建议打开这个选项来减少包体总体积.
- 包位移: 在打包时会在AssetBundle前写入一定数量的空字节来做简单加密, 防止某些软件对AssetBundle的破解. 这只是非常简单的一种方式, 并不能完全防止破解的发生, 因此请根据自己实际需要来设置这个数值, 过高的数值将导致每个AssetBundle的体积都变大.
- 打包回调: 你可以继承`BBGo.FinalPatch.IBuildCallback`来实现自己的打包回调代码, 实现完自己的类型后需要在这里选中你的类. 你可以使用这个来实现打包完毕后将所有`Package`下的资源拷贝到CDN目录. 详情参见:`BBGo.FinalPatch.BuildEditorCallback.cs`
- 路径转换: 资源加载时需要提供全名称路径,例如:`Assets/Res/Cube.prefab`(包括扩展名), 但是根据这个名称并不知道该资源位于哪个AssetBundle中, 该功能是将路径名转换为AssetBundle名称. 若没有特殊需求,可以直接使用`BBGo.FinalPatch.DefaultPathConverter.cs`.

3.2 更新

- 自动更新: 勾选后游戏启动时FinalPatch会自动检查更新. 若有特殊需求可以关闭改功能, 然后通过API来手动更新资源.
- 重试次数: 自动更新失败时会尝试几次.
- 重试间隔: 每次尝试的间隔.

3.3 池

FinalPatch提供了一个基本的GameObjectPool来帮你管理GameObject, 详情参见FinalPool.

- 最大缓存: 池中允许缓存的最大数量, 超过这个数量的资源在回收时将会被直接销毁.
- 自动释放: 若池处于空闲状态(没有任何引用)时, 是否自动释放来降低内存使用. 强烈建议打开该功能.
- 释放延迟: 当池处于空闲状态多久后自动释放该池.

3.4 其他

- 编辑器模式: 编辑器模式下将直接跳过热更新阶段, 并且也不会从AssetBundle中加载资源. 建议打开该功能以避免在开发过程中频繁的打包.
  注意: 即便打开该功能, 你所构建的任何版本客户端也不会受到影响. 因此你可以一直打开.
- 沙盒目录: 热更新的资源会放在该目录下, 你可以手动删掉这里的资源来反复更新和测试.

4. 热更新

为了正确的比对和更新资源, 客户端需要`FinalPatchClient`. 点击`Tools->BBGo->Add Final Patch Client`来添加. 你需要填写一些必要的参数:

- 部署数据地址: 客户端会从该地址下载部署数据, 详情参见: 部署数据.
- 部署渠道: 客户端会从部署数据中查找与这里匹配的渠道来比对版本和下载资源. 详情参见: 新增渠道

4.1 自动更新

如果你打开了自动更新, 此时运行游戏，FinalPatch会为您自动检查版本，更新资源。可以参考:`FinalPatch.AutoPatch()`中。

4.2 手动更新

若你有特殊需求， 可以手动进行资源更新。
`FinalPatcher.Collect()` 将收集所需更新资源的信息。返回值为`PatchReport`,里面有所需下载的资源，总字节数，Patch状态等信息。
`FinalPatcher.Patch(PatchReport report)` 用于更新资源。 这是一个异步方法。具体的更新方法请参考:`FinalPatcher.AutoPatch()`.

注意： 不管使用哪种方式, 更新步骤都必不可少， 如果未开启自动更新又没有进行手动更新， 资源加载将会失败。

5. 资源加载

5.1 FinalLoader

FinalLoader提供资源加载的接口.
`FinalLoader.LoadAssetAsync<T>(string assetName)`用于加载资源.
`FinalLoader.UnloadAsset(string assetName)`用于卸载资源.
`FinalLoader.LoadSceneAsync(string sceneName, LoadSceneMode mode)`用于加载场景.
`FinalLoader.UnloadSceneAsync(string sceneName)`用于卸载场景.

```
public async void Start()
{
    //load a material
    Material mat = await FinalLoader.LoadAssetAsync<Material>
("Assets/Res/Red.mat");
```

```
    //unload asset
    FinalLoader.UnloadAsset(mat);

    //load a scene
    await FinalLoader.LoadSceneAsync("Assets/Scenes/demo.unity",
LoadSceneMode.Additive);

    //unload scene
    await FinalLoader.UnloadSceneAsync("Assets/Scenes/demo.unity");
}
```

这里使用了async-await异步编程方式. 若对这种方式比较陌生请自行查找相关文档.

5.2 FinalPool

当你需要加载一个GameObject时可以调用FinalPool的接口. 在内部, FinalPool也是通过FinalLoader来加载资源的, 同时FinalPool使用池来帮你缓存这些GameObject来减少频繁实例化带来的性能开销. 因此若你需要GameObject, 强烈推荐使用FinalPool.
`FinalPool.GetGameObjectAsync(string assetName)`用于获取一个GameObject.
`FinalPool.ReleaseGameObject(GameObject obj)`用于释放一个GameObject.
`FinalPool.DestroyUnusedPools()`用于销毁所有未使用中的池.

```
public async void Start()
{
    //get a cube from pool. FinalPool will load this cube via FinalLoader
at first time you request and create a pool named "Assets/Res/Cube.prefab"
    GameObject cube1 = await
FinalPool.GetGameObjectAsync("Assets/Res/Cube.prefab");

    //if you request a gameobject which alred cached, final pool will
instantiate a copy for you instead of loading from asset bundles.
    GameObject cube2 = await
FinalPool.GetGameObjectAsync("Assets/Res/Cube.prefab");

    //release cube1 to pool. FinalPool just hide cube1. there is 1
Assets/Res/Cube.prefab in FinalPool after releasing.
    FinalPool.ReleaseGameObject(cube1);

    //FinalPool will show the cube you just released in last step. there
is 0 Assets/Res/Cube.prefab in FinalPool now.
    GameObject cube3 = await
FinalPool.GetGameObjectAsync("Assets/Res/Cube.prefab");

    //release cube2 to pool
    FinalPool.ReleaseGameObject(cube2);

    //Nothing happend because cube3 is in use.
    FinalPool.DestroyUnusedPools();
```

```
    //release cube3 to pool
    FinalPool.ReleaseGameObject(cube3);

    //the pool Asset/Res/Cube.prefab will release immediately. if you
enabled AutoRelease in advanced tab, you don't need to call this function
manually.
    FinalPool.DestroyUnusedPools();
}
```

联系我

如果有任何疑问或者建议，请发送邮件到teddyzhang29@gmail.com联系我, 我会在第一时间与您联系。