

# GameDriver Pro



GameDriver Pro 是一款基于Unity引擎的游戏开发框架

## 开始

### 概述

本用户指南旨在为 GameDriver Pro 用户提供关于工具的基本概述、功能和用法。

### 安装

1. 从Unity的Asset Store下载GameRiver Pro后，请转到：“Assets->Import Package->Custom Package...”。
2. 在“导入资产”窗口中，查找并选择GameDriver Pro UnityPackage文件。

3. 在“导入软件包”窗口中出现Unity之后，请验证所有项目选择导入，然后单击窗口右下方的导入按钮。
4. GameDriver Pro的全部文件将加入到Assets/GameDriver中。

您也可以选择所需的部分进行导入。

## 支持

如果您想快速了解 GameDriver Pro 中的重要功能，可以直接参考 GameDriver/Samples中的示例。

如果您想详细了解 GameDriver Pro 中的各功能模块的设计思路和细节，您可以在[在线帮助](#)中找到更多信息和常见问题解答。

如果您无法要查找您寻求的信息，请 [file an issue](#)(或者提交一个 pull request) 来描述一下你的遭遇。

或者联系作者：xuzhuoxi@gmail.com 或 mailxuzhuoxi@163.com

## 功能

GameDriver Pro中的源代码存放位置是有规律的。

- **GameDriver/Runtime/CSharp** 中的源码只依赖于C#标准库，不依赖Unity标准库。是对C#的功能扩展。
- **GameDriver/Runtime/Actions** 中的源码依赖于CSharp和Unity标准库。是对Unity引擎的功能扩展。
- **GameDriver/Runtime/Games** 中的源码依赖于CSharp与Actions。是游戏开发过程常用的系统功能的通用实现。

## 1. 事件(Event) - 高效的事件模块，支持Unity多线程

事件采用“监听-捕获”机制，监听时支持捕获次数、捕获优先级的设置。

- **JLGames.GameDriver.CSharp.Event** 负责事件系统的核心逻辑实现。
- **JLGames.GameDriver.Actions.ThreadEvent** 针对多线程提供支持。

### 1.1 核心功能 - 添加监听、移除监听、分发事件

#### 1.1.1 监听

接口IEventListener中 AddEvent Listener系列函数用于添加事件监听。  
实现类EventDispathver中已经完成逻辑实现。

```
/// <summary>
/// Add single event listener
/// 添加单次事件监听
/// </summary>
/// <param name="type" >Event Type<br/>事件类型</param>
/// <param name="handler" >Listener Function<br/>监听函数</param>
/// <param name="weight" >Response Weight<br/>响应权重</param>
/// <param name="tag" >Function Tag<br/>函数的唯一标签</param>
void OnceEventListener ( string type, EventDelegates.EventHandler handler, int weight =
EventConst.DefaultWeight, string tag = null );

/// <summary>
/// Add event listener
/// 添加事件监听
/// </summary>
/// <param name="type" >Event Type<br/>事件类型</param>
/// <param name="handler" >Listener Function<br/>监听函数</param>
/// <param name="tag" >Function Tag<br/>函数的唯一标签</param>
void AddEventListener ( string type, EventDelegates.EventHandler handler, string tag = null );

/// <summary>
/// Add event listener
/// 添加事件监听
/// </summary>
/// <param name="type" >Event Type<br/>事件类型</param>
/// <param name="handler" >Listener Function<br/>监听函数</param>
/// <param name="weight" >Response Weight<br/>响应权重</param>
/// <param name="tag" >Function Tag<br/>函数的唯一标签</param>
void AddEventListener ( string type, EventDelegates.EventHandler handler, int weight, string tag =
null );

/// <summary>
/// Add event listener
/// 添加事件监听
/// </summary>
/// <param name="type" >Event Type<br/>事件类型</param>
/// <param name="handler" >Listener Function<br/>监听函数</param>
/// <param name="listeningTimes" >Times of responses<br/>响应次数</param>
/// <param name="tag" >Function Tag<br/>函数的唯一标签</param>
void AddEventListener ( string type, EventDelegates.EventHandler handler, uint listeningTimes ,
string tag = null );

/// <summary>
/// Add event listener
/// 添加事件监听
/// </summary>
/// <param name="type" >Event Type<br/>事件类型</param>
/// <param name="handler" >Listener Function<br/>监听函数</param>
/// <param name="weight" >Response Weight<br/>响应权重</param>
/// <param name="listeningTimes" >Times of responses<br/>响应次数</param>
/// <param name="tag" >Function Tag<br/>函数的唯一标签</param>
void AddEventListener ( string type, EventDelegates.EventHandler handler, int weight, uint
listeningTimes , string tag = null );
```

## 1.1.2 移除监听

接口IEventListener中 RemoveEventListerner系列函数用于移除事件监听。  
实现类EventDispathver中已经完成逻辑实现。

```
/// <summary>
/// Delete event listener
/// 删除事件监听
/// </summary>
/// <param name="type">事件类型</param>
/// <param name="handler">监听函数</param>
/// <param name="tag"></param>
void RemoveEventListerner(string type, EventDelegates.EventHandler handler, string tag = null);

/// <summary>
/// Delete event listener
/// 删除事件监听
/// </summary>
/// <param name="type">事件类型</param>
/// <param name="tag"></param>
void RemoveEventListerner(string type, string tag);

/// <summary>
/// Delete a type of event listeners
/// 删除一类事件监听
/// </summary>
/// <param name="type">事件类型</param>
void RemoveEventListerner(string type);

/// <summary>
/// Clear all event listeners
/// 清除全部事件监听
/// </summary>
void RemoveEventListerner();
```

## 1.1.3 分发事件

接口IEventDispatcher中 DispatchEvent系列函数用于分发事件。  
实现类EventDispathver中已经完成逻辑实现。

```
/// <summary>
/// Trigger an event of a certain type and pass data
/// 触发某一类型的事件，并传递数据
/// </summary>
/// <param name="evd">Evd.</param>
void DispatchEvent(EventData evd);

/// <summary>
/// Trigger an event of a certain type and pass data
/// 触发某一类型的事件，并传递数据
/// </summary>
/// <param name="type">事件类型</param>
/// <param name="data">事件的数据(可为null)</param>
void DispatchEvent(string type, object data);
```

## 1.1.4 多线程支持

接口IThreadEventProxy中声明的函数定义了多线程支持规范。  
实现类ThreadEventDispatcher中已经完成逻辑实现。

```

/// <summary>
/// Enable main thread proxy mode
/// 开启主线程代理模式
/// </summary>
/// <param name="carrier"></param>
void OpenMainThreadProxy(GameObject carrier);

/// <summary>
/// Turn off main thread proxy mode
/// 关闭主线程代理模式
/// </summary>
void CloseMainThreadProxy();

```

### 1.1.5 扩展

继承EventDispatcher或ThreadEventDispatcher并实现自定义接口，可用于扩展事件行为。

### 1.2 示例

GameDriver/Samples/Event



## 2. 加载管理模块(Loader) - 支持Resources、Editor、Assetbundle三种模式自由切换

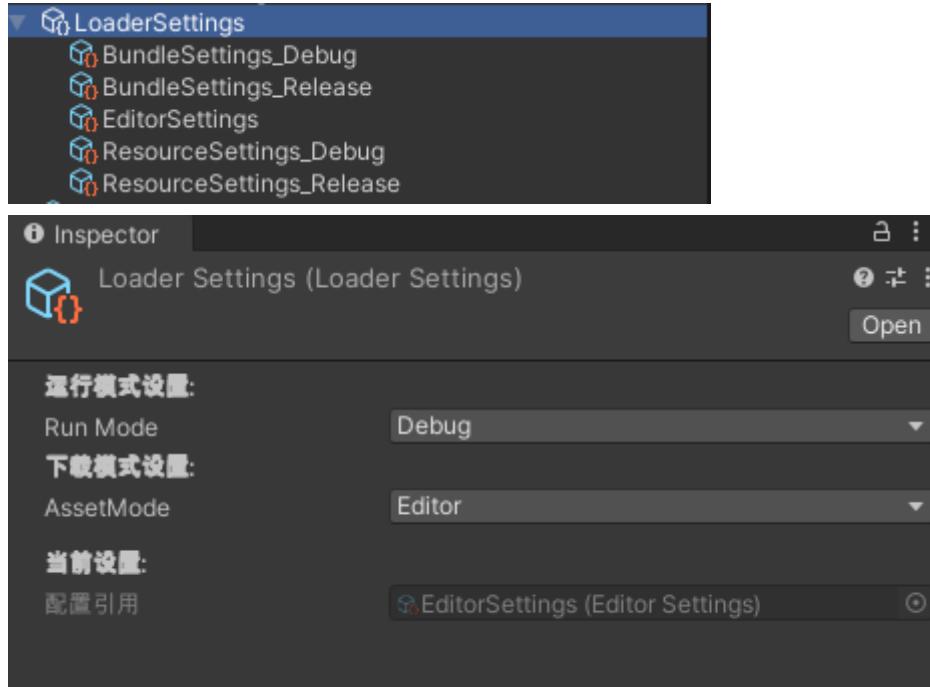
- **JLGames.GameDriver.Actions.Loader** 提供了加载器全部的功能支持。
- “Tools -> GameDriver -> Project -> Gen LoaderSettings”提供了加载器配置文件的生成入口。

### 2.1 初始化

#### 2.1.1 生成配置资产

执行菜单 “Tools -> GameDriver -> Project -> Gen LoaderSettings”。

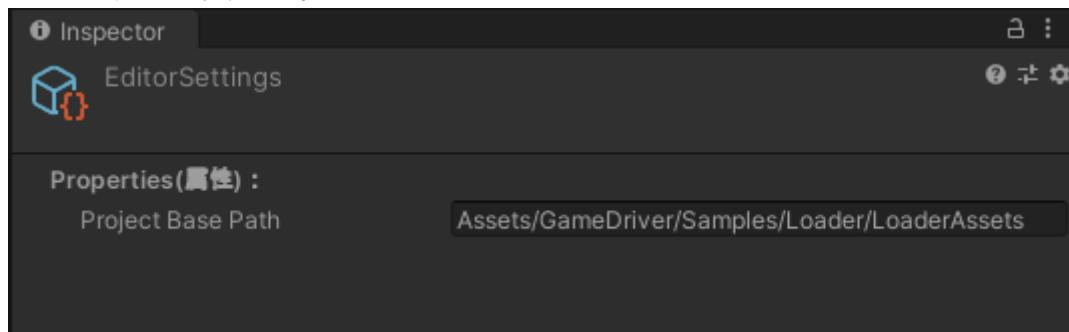
在项目Assets/Resources下会生成 LoaderSettings.asset(可重命名) 文件。



### 2.1.2 按项目需求设置配置

LoaderSettings中共有5个可用配置，一个Editor模式，两个Resource模式，两个AssetBundle模式。

- Editor配置 ProjectBasePath可以设置项目的资源查找的基础路径部分，在加载资源时可拼接到资源路径前。

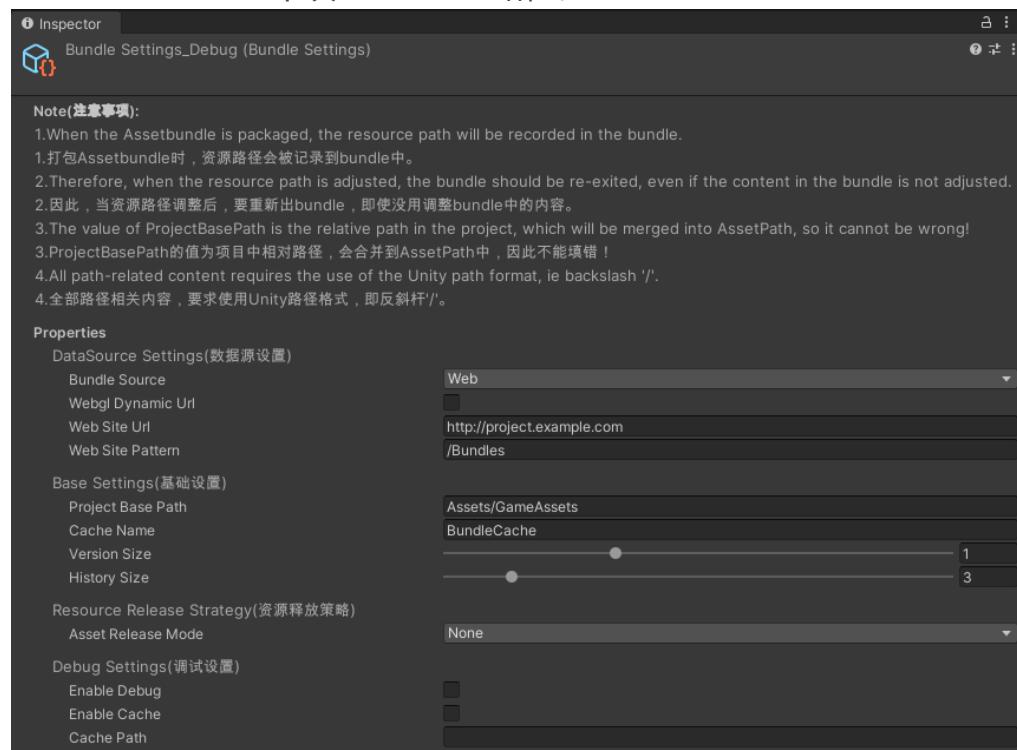


- Resource配置 ProjectBasePath可以设置项目的资源查找的基础路径部分，在加载资源时可拼接到资源路径前。



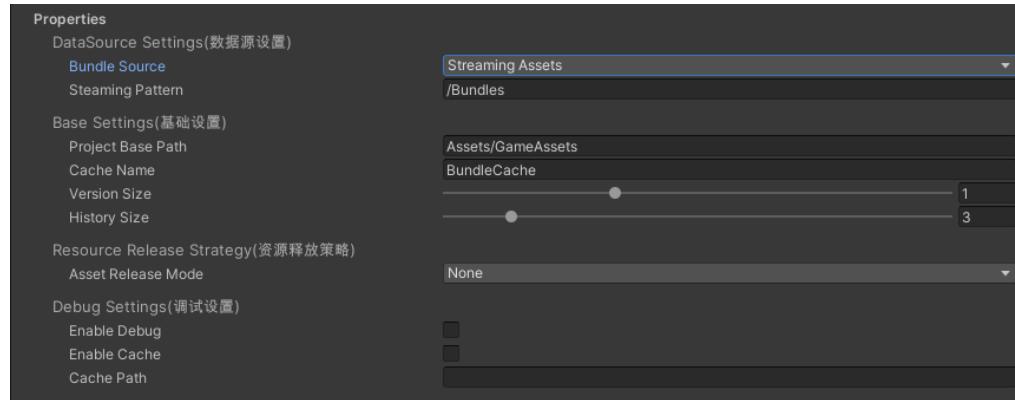
- AssetBundle配置

- BundleSource 可选择Web、Streaming Assets、File Debug三种模式。
  - Web 用于加载直接存入在站点上的AssetBundle资源。  
Web Site Url 中填入站点Url。  
Web Site Pattern 中填入Pattern路径。



- Streaming Assets 用于加载存放在项目StreamingAssets中的AssetBundle资源。

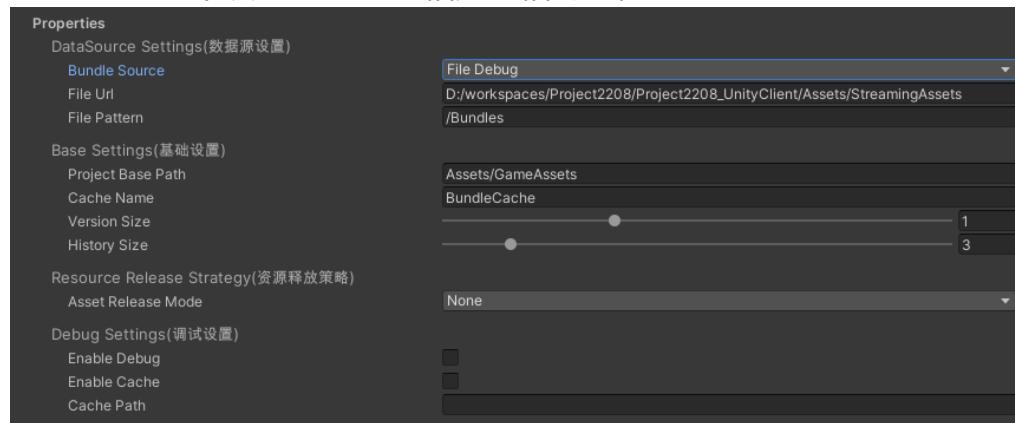
## Streaming Parttern 中填入相对于StreamingAssets的相对路径。



- File Debug 用于加载本地文件系统中的AssetBundle资源，多用于调试。

File Url 中填入本机文件路径。

File Pattern 中填入与File Url相关的相对路径。



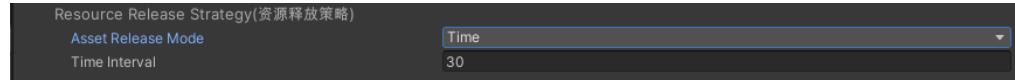
### o Base Settings

主要是关于项目的Bundle目录信息的设置与加载时缓存信息的设置。

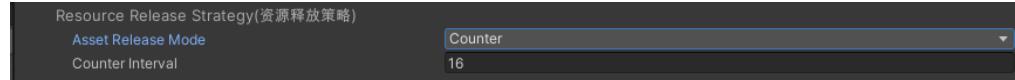
- Project Base Path 设置项目的Bundle目录路径
- CacheName 设置缓存区的名称，这个可以自定义。
- Version Size 设置每个Bundle资产在缓存中保留的版本数量。
- History size 设置缓存区数量，每一个CacheName会产生一个缓存区。

### o Resource Release Strategy 可选择None、Time、Counter三种。

- None为不选择释放策略，释放Bundle与资产后产生的无效内存占用需要自己调用gc释放，该策略适合自行管理内存与gc时机。
- Time为计时释放策略，会按照设定的时间周期调用gc。



- Counter为计数释放策略，当bundle加载资源到达设定值及倍数值时调用gc。



## 2.1.3 使用配置初始化加载器

使用以下API初始化加载器：

- 使用由**第一点**生成的配置文件名称进行初始化。

```
/// <summary>
/// Initialize an instance of ILoader
/// 初始化ILoader的实例
/// </summary>
/// <param name="loaderName"></param>
/// <param name="settingsName"></param>
/// <returns></returns>
public static ILoader InitLoader(string loaderName = DefaultName, string settingsName = DefaultName)
{
    if (string.IsNullOrEmpty(settingsName)) return null;
    var settings = LoadLoaderSettings(settingsName);
    return InitLoader(loaderName, settings);
}
```

- 使用配置实例进行初始化。

```
/// <summary>
/// Initialize an instance of ILoader
/// 初始化ILoader的实例
/// </summary>
/// <param name="loaderName"></param>
/// <param name="settings"></param>
/// <returns></returns>
public static ILoader InitLoader(string loaderName, LoaderSettings settings)
{
    if (string.IsNullOrEmpty(loaderName) || null == settings) return null;
    if (m_Loaders.ContainsKey(loaderName)) return m_Loaders[loaderName];
    var loader = InitLoader(settings);
    if (null == loader) return null;
    m_Loaders[loaderName] = loader;
    if (DefaultName == loaderName)
    {
        m_DefaultLoader = loader;
    }

    return loader;
}
```

- 使用Loader静态类中有快捷初始化函数：

```
/// <summary>
/// Initialize the default loader
/// 初始化默认加载器
/// </summary>
/// <param name="settingsName"></param>
public static void InitLoader(string settingsName = LoaderManager.DefaultName)
{
    LoaderManager.InitLoader(LoaderManager.DefaultName, settingsName);
}
```

## 2.1.4 初始化Bundle版本信息

调用加载器实例中函数：

```
/// <summary>
/// Initialize version information
/// Version information is not allowed to be cached
/// 初始化版本信息
/// 版本信息获取不允许进行缓存
/// </summary>
/// <param name="onVersionAssetBundleLoaded"></param>
IEnumerator InitVersion(LoaderDelegate.OnAssetLoaded<AssetBundleManifest> onVersionAssetBundleLoaded);
```

- 函数回调为初始化版本结束后执行，内部可判断初始化结果：成功 或 失败。

- 函数要求开启协程调用, 可以使用LoaderManager.Mono开启协程:

```
/// <summary>
/// Initialize version information
/// Version information is not allowed to be cached
/// 初始化版本信息
/// 版本信息获取不允许进行缓存
/// </summary>
/// <param name="onVersionLoaded"></param>
public static void InitVersion(LoaderDelegate.OnAssetLoaded<AssetBundleManifest> onVersionLoaded)
{
    DebugUtil.Log("InitVersion:", LoaderManager.Mono);
    LoaderManager.Mono.StartCoroutine(LoaderManager.DefaultLoader.InitVersion(onVersionLoaded));
}
```

## 2.2 使用

### 2.2.1 加载Bundle资产

在Bundle版本信息初始化完成后，才可以加载Bundle资产。

加载器实现了IBundleLoader接口, 包含的函数与加载Bundle资产相关。

加载Bundle资产要求使用协程，可以使用LoaderManager.Mono实例启用协程加载Bundle。

```
/// <summary>
/// Asynchronous download bundle
/// 异步下载资源包
/// </summary>
/// <param name="bundleName">资源包名称</param>
/// <param name="onBundleLoaded">回调</param>
/// <param name="autoRelease">是否自动释放资源包</param>
/// <param name="unloadAllLoadedObjects">是否全部释放实例化的资源</param>
/// <returns></returns>
public static void LoadBundleAsync(string bundleName, LoaderDelegate.OnBundleLoaded onBundleLoaded,
    bool autoRelease = true, bool unloadAllLoadedObjects = false)
{
    LoaderManager.Mono.StartCoroutine(LoaderManager.DefaultLoader.LoadBundleAsync(bundleName, onBundleLoaded,
        autoRelease, unloadAllLoadedObjects));
}
```

- bundleName: bundle资产名称。
- onBundleLoaded: 执行结果回调, 内部可判断成功与否。
- autoRelease: 指明在onBundleLoaded执行结束后是否释放bundle实例
- unloadAllLoadedObjects: 指明在onBundleLoaded执行结束后是否释放从bundle实例中实例化的资源资产
- 如果autoRelease为true, 在onBundleLoaded中应该实例化全部要使用的资源资产。

### 2.2.2 加载资源资产

在获得bundle实例的情况下，可以从bundle实例中实例化出资源资产的实例，然后再克隆使用。

加载资源资产建议使用同步函数，不建议使用异步。原因是Unity对于在协程内部开启协程支持得不友好，层级过多(好像是16层)会出现无法预测的报错。

IAssetLoader接口函数分四类：

- 单个资源资产加载(同步|异步)

```
/// <summary>
/// Load asset from bundle synchronously with relative path
/// 使用相对路径从Bundle中加载资源
/// </summary>
/// <param name="assetPath">资源路径</param>
/// <param name="bundle">资源包</param>
/// <typeparam name="T">资源类型</typeparam>
/// <returns></returns>
T LoadAssetSync<T>(string assetPath, AssetBundle bundle) where T : Object;

/// <summary>
/// Load asset from bundle synchronously with full path
/// 使用完整路径从Bundle中加载资源
/// The full path: refers to the relative path after removing ProjectbasePath
/// 完整路径：指的是去除ProjectbasePath后的相对路径
/// </summary>
/// <param name="fullPath">完整资源路径,忽略设置的ProjectbasePath</param>
/// <param name="bundle">资源包</param>
/// <typeparam name="T">资源类型</typeparam>
/// <returns></returns>
T LoadAssetSyncFull<T>(string fullPath, AssetBundle bundle) where T : Object;

/// <summary>
/// Asynchronously load resources from bundle
/// 异步从资源包中加载资源
/// </summary>
/// <param name="assetPath">资源路径</param>
/// <param name="bundle">资源包</param>
/// <param name="onAssetLoaded">回调</param>
/// <typeparam name="T">资源类型</typeparam>
/// <returns></returns>
IEnumerator LoadAssetAsync<T>(string assetPath, AssetBundle bundle,
    LoaderDelegate.OnAssetLoaded<T> onAssetLoaded) where T : Object;
```

- 批量资源资产加载(同步|异步)

```
/// <summary>
/// Synchronized load resources from bundle in batches
/// 同步批量从资源包中加载资源
/// </summary>
/// <param name="assetPaths">资源路径</param>
/// <param name="bundle">资源包</param>
/// <typeparam name="T">资源类型</typeparam>
/// <returns></returns>
T[] LoadAssetsSync<T>(string[] assetPaths, AssetBundle bundle) where T : Object;

/// <summary>
/// Synchronized load resources from bundle in batches
/// 同步批量从资源包中加载资源
/// </summary>
/// <param name="fullPaths">完整资源路径,忽略设置的ProjectbasePath</param>
/// <param name="bundle">资源包</param>
/// <typeparam name="T">资源类型</typeparam>
/// <returns></returns>
T[] LoadAssetsSyncFull<T>(string[] fullPaths, AssetBundle bundle) where T : Object;

/// <summary>
/// Batch asynchronously load resources from bundle
/// 异步从资源包中批量加载资源
/// </summary>
/// <param name="assetPaths">路径数组</param>
/// <param name="bundle">资源包</param>
/// <param name="onMultiAssetLoaded">回调</param>
/// <typeparam name="T"></typeparam>
/// <returns></returns>
IEnumerator LoadMultiAssetAsync<T>(string[] assetPaths, AssetBundle bundle,
LoaderDelegate.OnMultiAssetLoaded<T> onMultiAssetLoaded)
where T : Object;
```

- 单个子资源资产加载(同步|异步)

```

/// <summary>
/// Load sub-resources (sprites, prefabs, etc.)
/// 加载子资源(子图、预置等)
/// </summary>
/// <param name="path"></param>
/// <param name="subName"></param>
/// <param name="ab"></param>
/// <returns></returns>
Object LoadSubAssetSync(string path, string subName, AssetBundle ab);

/// <summary>
/// Load sub-resources (sprites, prefabs, etc.)
/// 加载子资源(子图、预置等)
/// </summary>
/// <param name="path"></param>
/// <param name="subName"></param>
/// <param name="type"></param>
/// <param name="ab"></param>
/// <returns></returns>
Object LoadSubAssetSync(string path, string subName, Type type, AssetBundle ab);

/// <summary>
/// Load sub-resources (sprites, prefabs, etc.)
/// 加载子资源(子图、预置等)
/// </summary>
/// <param name="path"></param>
/// <param name="subName"></param>
/// <param name="ab"></param>
/// <returns></returns>
T LoadSubAssetSync<T>(string path, string subName, AssetBundle ab) where T : Object;

/// <summary>
/// Load sub-resources (sprites, prefabs, etc.) from bundles asynchronously
/// 异步从资源包中加载加载子资源(子图、预置等)
/// </summary>
/// <param name="assetPath">资源路径</param>
/// <param name="subName"></param>
/// <param name="bundle">资源包</param>
/// <param name="onAssetLoaded">Callback(回调)</param>
/// <typeparam name="T">Resource type(资源类型)</typeparam>
/// <returns></returns>
IEnumerator LoadSubAssetAsync<T>(string assetPath, string subName, AssetBundle bundle,
LoaderDelegate.OnAssetLoaded<T> onAssetLoaded)
where T : Object;

```

- 批量子资源资产加载(同步|异步)

```

/// <summary>
/// Load all sub-resources (sprites, prefabs, etc.)
/// 加载子资源(子图、预置等)
/// </summary>
/// <param name="path"></param>
/// <param name="type"></param>
/// <param name="bundle"></param>
/// <returns></returns>
Object[] LoadSubAssetsSync(string path, Type type, AssetBundle bundle);

/// <summary>
/// Load all sub-resources (sprites, prefabs, etc.)
/// </summary>

```

```

    ///> 加载子资源(子图、预置等)
    ///> </summary>
    ///> <param name="path"></param>
    ///> <param name="bundle"></param>
    ///> <typeparam name="T"></typeparam>
    ///> <returns></returns>
    T[] LoadSubAssetsSync<T>(string path, AssetBundle bundle) where T : Object;

    ///> <summary>
    ///> Load sub-resources (sprites, prefabs, etc.)
    ///> 加载子资源(子图、预置等)
    ///> </summary>
    ///> <param name="path"></param>
    ///> <param name="subNames"></param>
    ///> <param name="bundle"></param>
    ///> <returns></returns>
    Object[] LoadSubAssetsSync(string path, string[] subNames, AssetBundle bundle);

    ///> <summary>
    ///> Load sub-resources (sprites, prefabs, etc.)
    ///> 加载子资源(子图、预置等)
    ///> </summary>
    ///> <param name="path"></param>
    ///> <param name="subNames"></param>
    ///> <param name="type"></param>
    ///> <param name="ab"></param>
    ///> <returns></returns>
    Object[] LoadSubAssetsSync(string path, string[] subNames, Type type, AssetBundle ab);

    ///> <summary>
    ///> Load sub-resources (sprites, prefabs, etc.)
    ///> 加载子资源(子图、预置等)
    ///> </summary>
    ///> <param name="path"></param>
    ///> <param name="subNames"></param>
    ///> <param name="bundle"></param>
    ///> <typeparam name="T"></typeparam>
    ///> <returns></returns>
    T[] LoadSubAssetsSync<T>(string path, string[] subNames, AssetBundle bundle) where T : Object;

    ///> <summary>
    ///> Load sub-resources (sprites, prefabs, etc.) from bundles asynchronously
    ///> 异步从资源包中加载加载子资源(子图、预置等)
    ///> </summary>
    ///> <param name="assetPath">资源路径</param>
    ///> <param name="bundle">资源包</param>
    ///> <param name="onAssetsLoaded">Callback(回调)</param>
    ///> <typeparam name="T">Resource type(资源类型)</typeparam>
    ///> <returns></returns>
    IEnumerator LoadSubAssetsAsync<T>(string assetPath, AssetBundle bundle,
    LoaderDelegate.OnMultiAssetLoaded<T> onAssetsLoaded)
    where T : Object;

```

- 更多用法请参考示例、API或源码。

## 2.3 示例

## GameDriver/Samples/Loader



## 3. 国际化(i18n) - 轻量级国际化解决方案

- **JLGAMES.GameDriver.Actions.i18n** 提供了国际化模块全部的功能支持。
- 建议使用 [BabelEdit](#) 创作管理数据集。

### 3.1 初始化

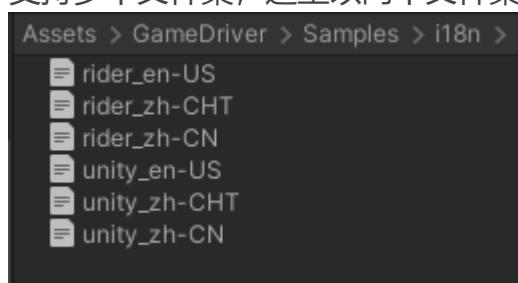
按照以下流程，为您的项目配置国际化功能：

#### 3.1.1 理清楚项目需要支持的国际化语言种类

这里以 **英文[英文]**、**中文[简体]**、**中文[繁体]** 举例。

#### 3.1.2 准备国际化数据文件

支持多个文件集，这里以两个文件集 **unity**、**rider** 举例。



数据格式以 Key-Value 形式：

```
{
    "i18n.rider.menu.Assets": "Assets",
    "i18n.rider.menu.Code": "Code",
    "i18n.rider.menu.Component": "Component",
    "i18n.rider.menu.Edit": "Edit",
    "i18n.rider.menu.File": "File",
    "i18n.rider.menu.GameObject": "GameObject",
    "i18n.rider.menu.Help": "Help",
    "i18n.rider.menu.Tools": "Tools",
    "i18n.rider.menu.Window": "Window"
}
```

当前测试数据集使用 JetBrains 的 [BabelEdit](#) 创作，这是一个不错的工具。

### 3.1.3 向注册表注册语言。

调用 I18NRegister 实例中的 RegisterLang 函数：

```
/// <summary>
/// Register language
/// 注册语言
/// </summary>
/// <param name="langName">Language name 语言名称</param>
/// <param name="langSuffix">Language file suffix 语言文件后缀</param>
/// <param name="default">Set as default? 是否设置为默认</param>
public void RegisterLang(string langName, string langSuffix, bool @default = false)
{
    m_LangSuffixMap[langName] = langSuffix;
    if (@default) m_DefaultLang = langName;
}
```

- langName 与 langSuffix 建议一样。
- default=true 时设置当前语言为默认语言。

### 3.1.4 向注册表注册文件集加载信息

调用 I18NRegister 实例中的 RegisterFile 函数：

```
/// <summary>
/// Register loading info of the file set.
/// 注册文件集加载信息
/// </summary>
/// <param name="fileKey">file set key name 文件映射Key</param>
/// <param name="fileBundle">bundle of the file set 文件集所在bundle</param>
/// <param name="filePath"></param>
/// <param name="default"></param>
public void RegisterFile(string fileKey, string fileBundle, string filePath, bool @default = false)
{
    m_FilePathMap[fileKey] = new RegisterFileInfo(fileKey, fileBundle, filePath);
    if (@default) m_DefaultFileKey = fileKey;
}
```

- fileKey 用于标识当前设置的文件集。
- default=true 时设置当前文件集为默认文件集。

### 3.1.5 根据注册表，加载数据到管理器中

调用 II18NManager 实例中的 LoadData 函数：

```
I18NManagerShared.Manager.LoadData();

/// <summary>
/// Reload data according to the settings in Register
/// 根据Register中设置重新加载数据
/// </summary>
void LoadData();
```

**注意：**如果在过去已经执行为加载数据，那您应该先清除旧数据：

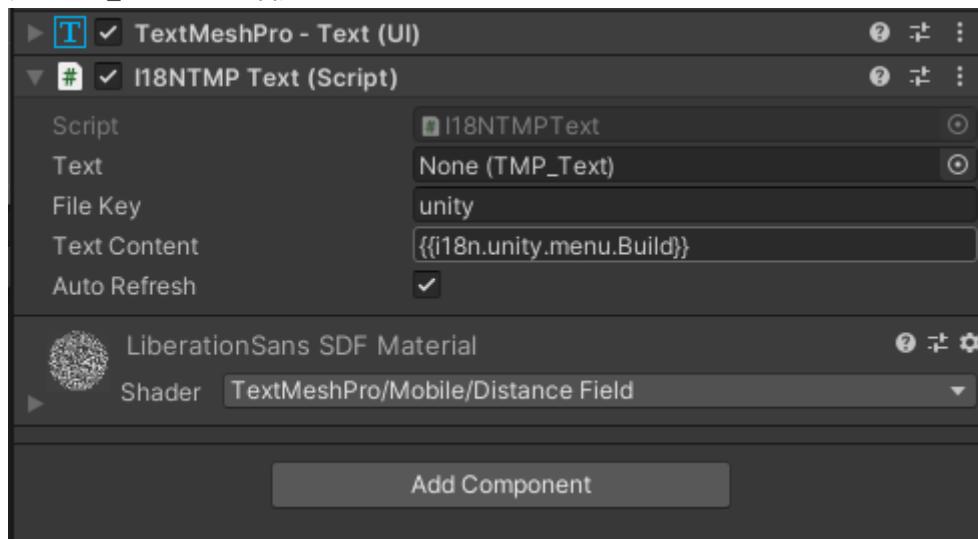
```
I18NManagerShared.Manager.ClearData();

/// <summary>
/// Clear all
/// 清除全部
/// </summary>
void ClearData();
```

## 3.2 使用

### 3.2.1 使用 I18NTMPText 组件，为TMP\_Text增加国际化支持。

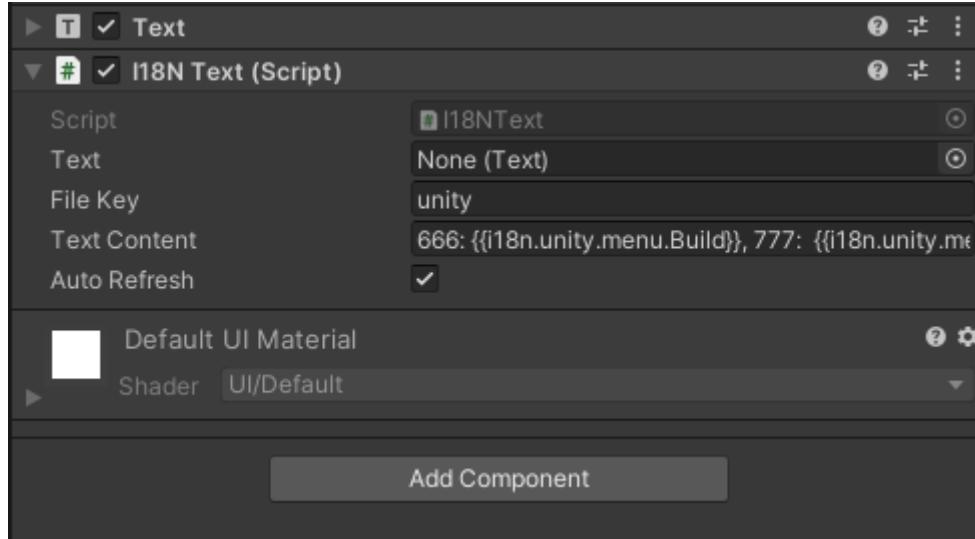
为TMP\_Text节点增加 I18NTMPText 组件



- File Key 为注册文件集时的fileKey参数。
- Auto Refresh 选中时，管理器重新加载数据后会自刷新。

### 3.2.2 使用 I18NText 组件，为Text增加国际化支持。

## 为Text节点增加 I18NText 组件



- File Key 为注册文件集时的fileKey参数。
- Auto Refresh 选中时，管理器重新加载数据后会自刷新。

### 3.2.3 在代码中读取国际化数据。

```
I18NManagerShared.Manager.GetValue(id, fileKey, lang);
```

```
/// <summary>
/// Get data, use default language and default file
/// 取数据，使用默认语言，默认文件
/// </summary>
/// <param name="id"></param>
/// <returns></returns>
string GetValue(string id);

/// <summary>
/// Get data, use default language
/// 取数据，使用默认语言
/// </summary>
/// <param name="id"></param>
/// <param name="fileKey"></param>
/// <returns></returns>
string GetValue(string id, string fileKey);

/// <summary>
/// Get data
/// 取数据
/// </summary>
/// <param name="id"></param>
/// <param name="fileKey"></param>
/// <param name="langName"></param>
/// <returns></returns>
string GetValue(string id, string fileKey, string langName);
```

### 3.2.4 在代码中把文本内容进行国际化。

在文件内容中支持脚本格式:

```
I18NManagerShared.Manager.GetContent(content, fileKey, lang);

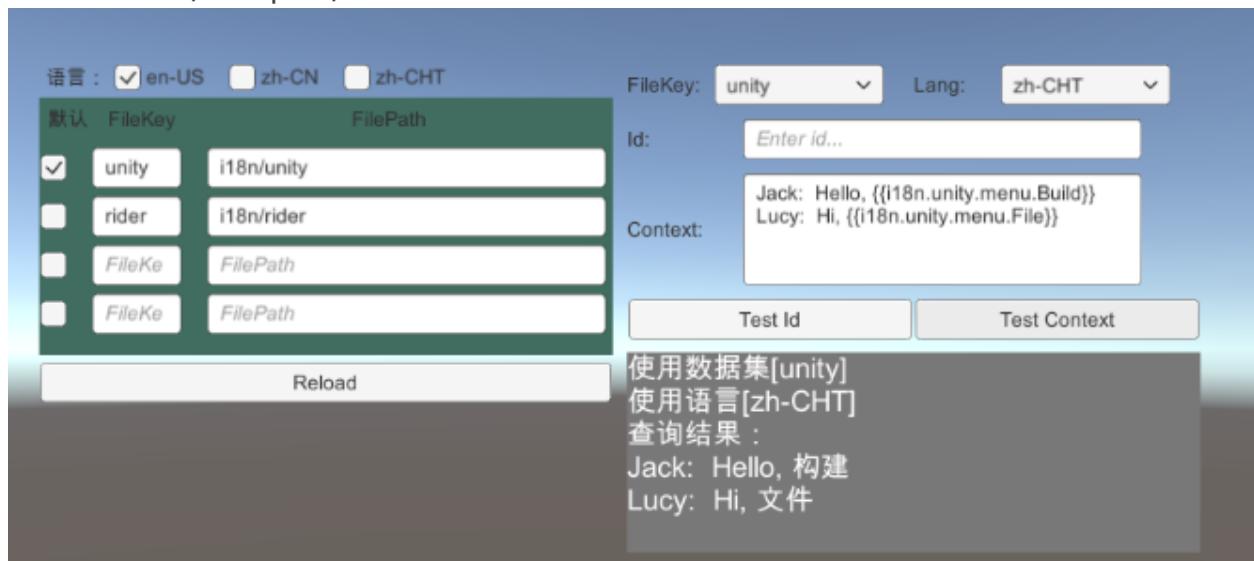
/// <summary>
/// Get Internationalized Content, use default language and default file
/// 获取国际化后的内容，使用默认语言，默认文件
/// </summary>
/// <param name="content"></param>
/// <returns></returns>
string GetContent(string content);

/// <summary>
/// Get Internationalized Content, use default language
/// 获取国际化后的内容，使用默认语言
/// </summary>
/// <param name="content"></param>
/// <param name="fileKey"></param>
/// <returns></returns>
string GetContent(string content, string fileKey);

/// <summary>
/// Get Internationalized Content
/// 获取国际化后的内容
/// </summary>
/// <param name="content"></param>
/// <param name="fileKey"></param>
/// <param name="langName"></param>
/// <returns></returns>
string GetContent(string content, string fileKey, string langName);
```

### 3.3 示例

GameDriver/Samples/i18n



## 4. 网络管理扩展模块(NetManager)

简单易用的网络链接管理模块

- JLGames.GameDriver.CSharp.Net 提供了C# 针对 Socket API 的一个封装，使开发者可以对TCP、UDP、HTTP(HTTPS)、WebSocket/WebSockets等协议在同一

一套API上进行开发。

- **JLGames.GameDriver.CSharp.Net** 同时提供了对数据进行封包解包的支持与扩展。
- **JLGames.GameDriver.Games.NetManager** 中提供了对链接引用的管理功能。
- **JLGames.GameDriver.Games.NetManager.Virtual** 中单机服务器的扩展支持，可以为无实际网络链接的情况下模拟服务器协议捕捉与逻辑处理的行为。

## 4.1 VirtualServer的使用

### 4.1.1 实例化并加入到NetManager管理

```
var server = new VirtualServer("server", "none");
NetManager.Shared.Register(server, true);
```

```
/// <summary>
/// Add the instance of INetEntity to the management
/// Currently, instance management of INetClient and INetServer is supported.
/// 把INetEntity的实例加入到管理中
/// 当前支持INetClient与INetServer的实例管理
/// </summary>
/// <param name="entity"></param>
/// <param name="default"></param>
public void Register(INetEntity entity, bool @default = false)
{
    if (null == entity) return;
    if (entity is INetClient)
        m_ClientCache.RegisterEntity(entity as INetClient, @default);
    if (entity is INetServer)
        m_ServerCache.RegisterEntity(entity as INetServer, @default);
}
```

加入到NetManager中的对象可以通过名称获得。

### 4.1.2 注册扩展

```
server.RegisterExtension(virtualReqIds.Id0,
    OnRequestHandler);
```

```
/// <summary>
/// Register extension
/// 注册扩展
/// </summary>
/// <param name="protoId"></param>
/// <param name="onProtoRequestHandler"></param>
void RegisterExtension(string protoId, VirtualServerDelegate.OnProtoRequestHandler onProtoRequestHandler);
```

- protoId: 协议号，由开发者定义。
- onProtoRequestHandler: 响应函数，对应协议号的响应管理逻辑。

### 4.1.3 协议响应

### 响应委托声明: VirtualServerDelegate.OnProtoRequestHandler

```
/// <summary>
/// Delegate declaration for protocol handle function
/// 协议响应函数的委托声明
/// </summary>
/// <param name="protoId"></param>
/// <param name="data"></param>
public delegate void OnProtoRequestHandler(IVirtualServer server, string protoId, object data);
```

- server: IVirtualServer实例，在函数中主要用于发送协议结果和数据的广播通知。
- protoId: 协议号
- data: 协议请求时传送的数据

#### 4.1.4 消息推送

- INotifyServer接口包含了消息推送用的函数声明。

```
public interface INotifyServer
{
    /// <summary>
    /// Message notify to client
    /// 向客户端发送消息推送
    /// </summary>
    /// <param name="protoId"></param>
    /// <param name="data"></param>
    void NotifyToClient(string protoId, object data = null);
}
```

- IMaterialNotifyServer接口包含了材料数据相关的消息推送用函数声明。

```
public interface IMaterialNotifyServer
{
    /// <summary>
    /// Notify Material Update - Offset
    /// 通知材料更新 - 偏移
    /// </summary>
    /// <param name="offset"></param>
    void NotifyMaterialEntryUpdate(DataOffset offset);

    /// <summary>
    /// Notify Material Update - Number
    /// 通知材料更新 - 数量
    /// </summary>
    /// <param name="num"></param>
    void NotifyMaterialEntryUpdate(DataNum num);

    /// <summary>
    /// Notify Material Update - Number and Offset
    /// 通知材料更新 - 偏移+数量
    /// </summary>
    /// <param name="notify"></param>
    void NotifyMaterialEntryUpdate(UserNotifyData notify);

    /// <summary>
    /// Notify Multiple Material Update - Offset
    /// 通知材料更新 - 偏移
    /// </summary>
    /// <param name="offsetArr"></param>
    void NotifyMaterialEntryUpdate(DataOffset[] offsetArr);

    /// <summary>
    /// Notify Multiple Material Update - Number
    /// 通知材料更新 - 数量
    /// </summary>
    /// <param name="numArr"></param>
    void NotifyMaterialEntryUpdate(DataNum[] numArr);

    /// <summary>
    /// Notify Multiple Material Update - Number and Offset
    /// 通知材料更新 - 偏移+数量
    /// </summary>
    /// <param name="notifyArr"></param>
    void NotifyMaterialEntryUpdate(UserNotifyData[] notifyArr);
}
```

- 响应委托virtualServerDelegate.OnProtoRequestHandler中的server实例，已经实现了INotifyServer和IMaterialNotifyServer接口，通过调用消息推送函数，实现数据的广播功能。

## 4.2 VirtualClient的使用

### 4.2.1 实例化并加入到NetManager管理。

```
var client = new virtualclient("client", "none");
NetManager.Shared.Register(client, true);
```

```
/// <summary>
/// Add the instance of INetEntity to the management
/// Currently, instance management of INetClient and INetServer is supported.
/// 把INetEntity的实例加入到管理中
/// 当前支持INetClient与INetServer的实例管理
/// </summary>
/// <param name="entity"></param>
/// <param name="default"></param>
public void Register(INetEntity entity, bool @default = false)
{
    if (null == entity) return;
    if (entity is INetClient)
        m_ClientCache.RegisterEntity(entity as INetClient, @default);
    if (entity is INetServer)
        m_ServerCache.RegisterEntity(entity as INetServer, @default);
}
```

加入到NetManager中的对象可以通过名称获得。

#### 4.2.2 连接服务器

```
var serverProxy = NetManager.Shared.
    GetServer<IVirtualServer>("server") as IVirtualServerProxy;
client.Connect(serverProxy);
```

```
/// <summary>
/// Connect to server
/// 连接Server
/// </summary>
void Connect(IVirtualServerProxy server);

/// <summary>
/// Disconnect from server
/// 断开Server连接
/// </summary>
void Disconnect();

/// <summary>
/// Reconnect to server
/// 重连
/// </summary>
void Reconnect(IVirtualServerProxy server);
```

- 从管理器中取得服务器代理(实际为VirtualServer对象)。
- 连接代理函数为Connect。
- 取消代理连接函数为Disconnect。

#### 4.2.3 监听请求响应、监听消息推送

监听功能建立在事件机制下：

```
client.AddEventListener(VirtualClientEvents.EventResponse,
    OnClientResponse);
client.AddEventListener(VirtualClientEvents.EventNotify,
    OnClientNotify);
```

```
/// <summary>
/// Received protocol response event
/// 收到协议响应事件
/// Data Format: VirtualClientResponse
/// 数据格式: VirtualClientResponse
/// </summary>
public const string EventResponse = "VirtualClientEvents.EventResponse";

/// <summary>
/// Received protocol notify event
/// 收到协议通知事件
/// Data Format: VirtualClientResponse
/// 数据格式: VirtualClientResponse
/// </summary>
public const string EventNotify = "VirtualClientEvents.EventNotify";
```

#### 4.2.4 发送消息请求

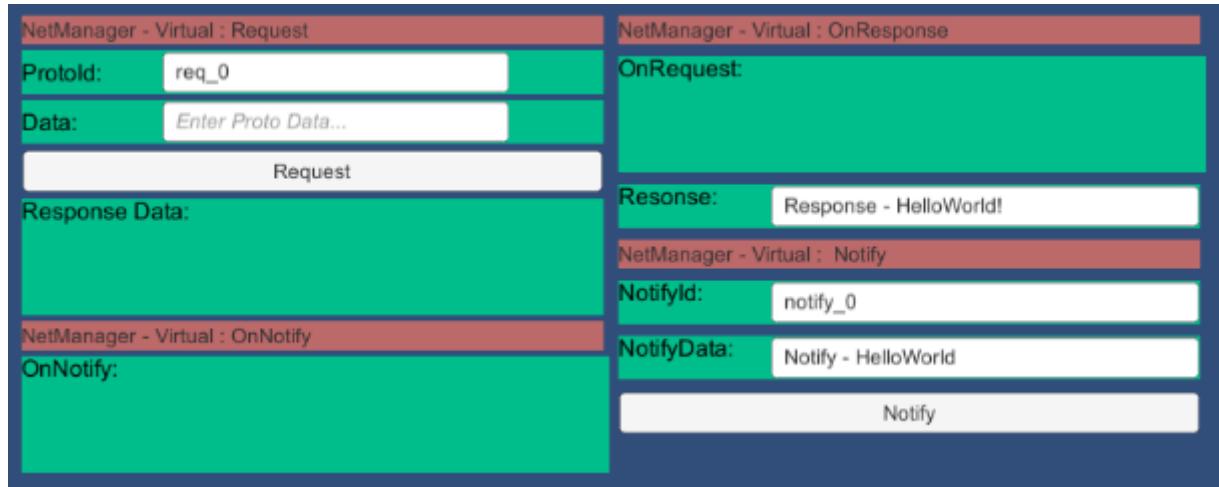
使用接口IVirtualClient下的Request函数可以向服务器发送消息请求。

```
NetManager.Shared.GetClient<IVirtualClient>("client")
    .Request(protoId, m_InputrequestData.text.Trim());
```

```
/// <summary>
/// Send a protocol request to the server
/// 向Server发送协议请求
/// </summary>
/// <param name="protoId"></param>
/// <param name="data"></param>
void Request(string protoId, object data);
```

### 4.3 示例

## GameDriver/Samples/NetManager



## 5. 音频管理模块(AudioManager)

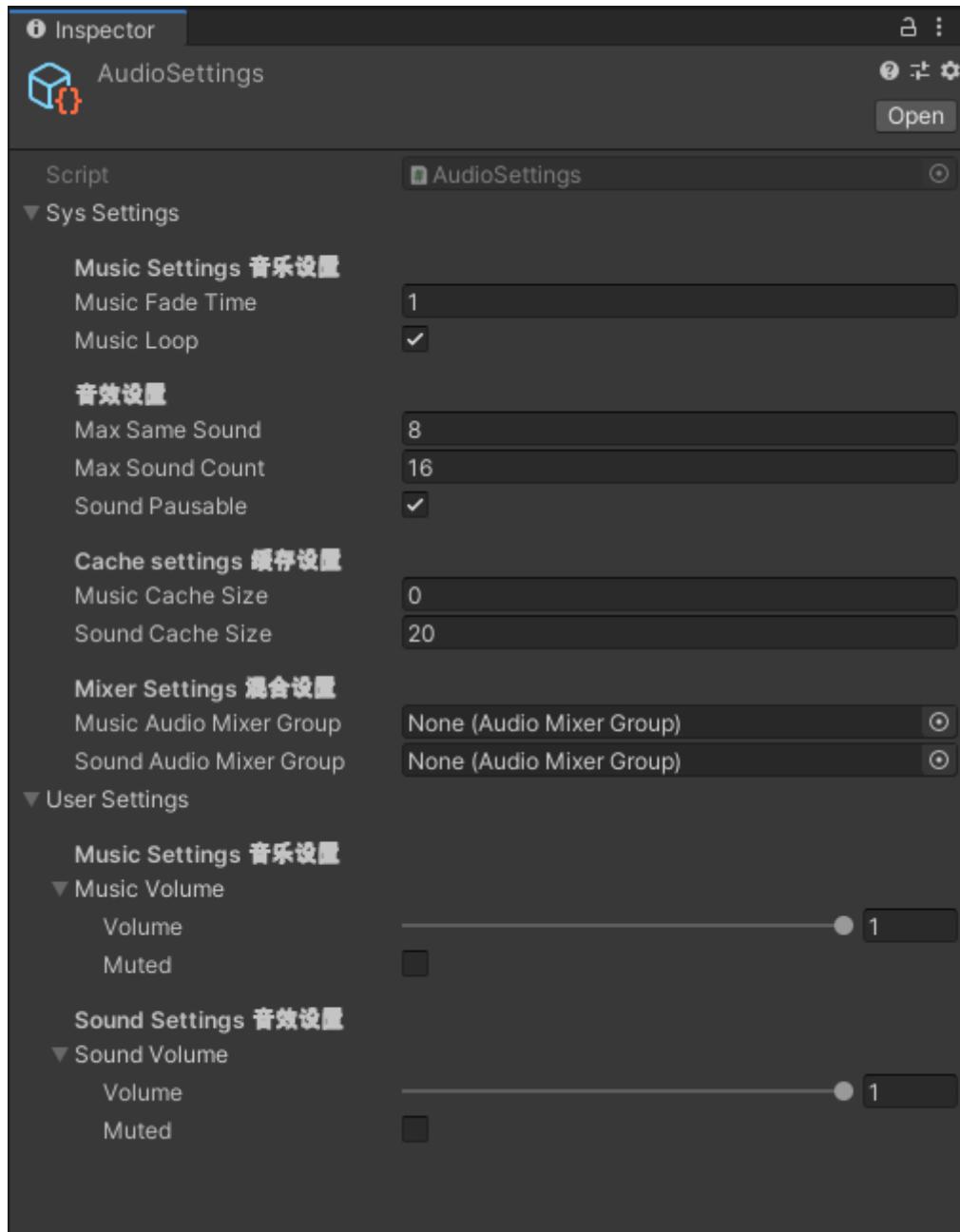
完善的音频管理模块，支持场景与UI的音乐音效同时，去除Assetbundle的依赖。

- **JLGames.GameDriver.Actions.Audio** 提供了音频管理全部的功能支持。
- 音频的加载使用默认的加载器(Loader)，同时也支持自定义的加载器。
- 采用Ico方式注入音频资源信息。
- 无须直接引用音频资源，实现了资源使用与资源打包的分离。
- 支持缓存设置，减少加载消耗。

### 5.1 初始化

#### 5.1.1 生成配置资产

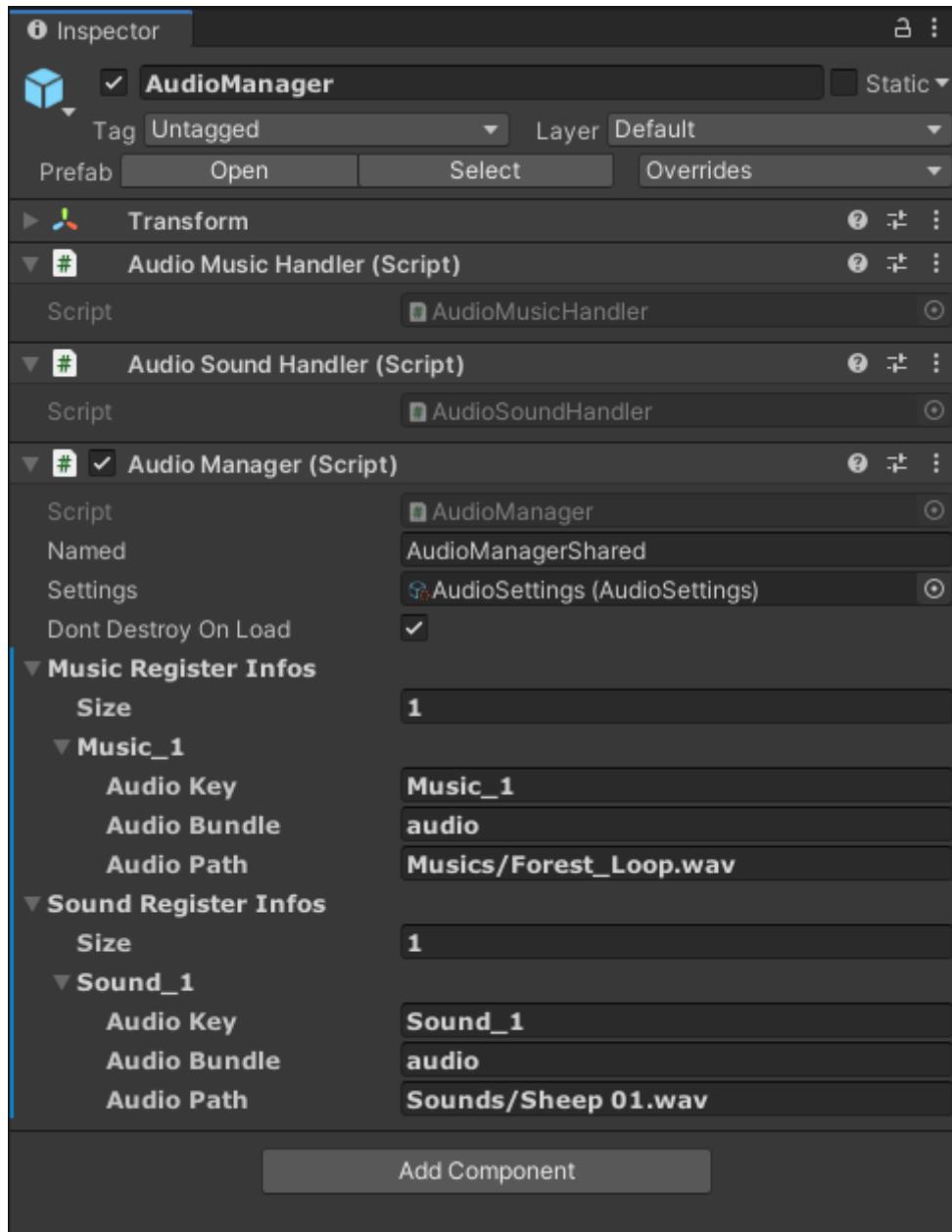
执行菜单 “Tools -> GameDriver -> Project -> Gen AudioSettings”。  
在项目Assets/Resources下会生成 AudioSettings.asset(可重命名) 文件。



- SysSettings为系统级别的设置，用户无法修改。
- UserSettings为用户级别的设置，开放AP修改。I

### 5.1.2 设置管理物件

复制GameDriver/Assets/ AudioManager/Prefabs/ AudioManager.prefab到项目初始化场景中，并重新关联AudioSettings.asset



- **Dont Destroy OnLoad** 场景销毁时是否保存物件
- **Music Register Infos** 音乐信息预处理注册表
- **Sound Register Infos** 音效信息预处理注册表

### 5.1.3 设置音频加载器：

```
AudioManagerPool.Shared.SetLoaderAdapter(new AudioLoader());
```

如果要必要自定义加载， 实现IAudioLoader接口即可。

## 5.2 使用

### 5.2.1 注册音频信息(可选)

可以在脚本中使用AudioManager函数注册音频信息

```
/// <summary>
/// Register music info
/// 注册音乐信息
/// </summary>
/// <param name="key"></param>
/// <param name="bundleName"></param>
/// <param name="path"></param>
void RegisterMusic(string key, string bundleName, string path);

/// <summary>
/// Clear music register
/// 清理音乐注册表
/// </summary>
void ClearMusicRegister();

/// <summary>
/// Register sound info.
/// 注册音效信息
/// </summary>
/// <param name="key"></param>
/// <param name="bundleName"></param>
/// <param name="path"></param>
void RegisterSound(string key, string bundleName, string path);

/// <summary>
/// Clear sound register
/// 清理音效注册表
/// </summary>
void ClearSoundRegister();
```

## 5.2.2 播放音乐、播放音效

- AudioManger中以SwitchMusic开头的函数为播放音乐提供支持。
- AudioManager中以PlaySound开头的函数为播放音效提供支持。
- 音乐音效的播放支持**不注册**直接播放。

## 5.2.3 音频设置

- AudioManager 支持音乐静音与音量设置。

```
/// <summary>
/// The name of the music playing
/// 播放中的音乐名
/// </summary>
string PlayingMusicName { get; }

/// <summary>
/// Set music volume
/// 设置音乐音量
/// </summary>
/// <param name="volume"></param>
/// <param name="refresh"></param>
void SetMusicVolume(float volume, bool refresh = true);
```

- AudioManager 支持音效静音与音量设置。

```
/// <summary>
/// Set sound volume
/// 设置音效音量
/// </summary>
/// <param name="volume"></param>
/// <param name="refresh"></param>
void SetSoundVolume(float volume, bool refresh = true);

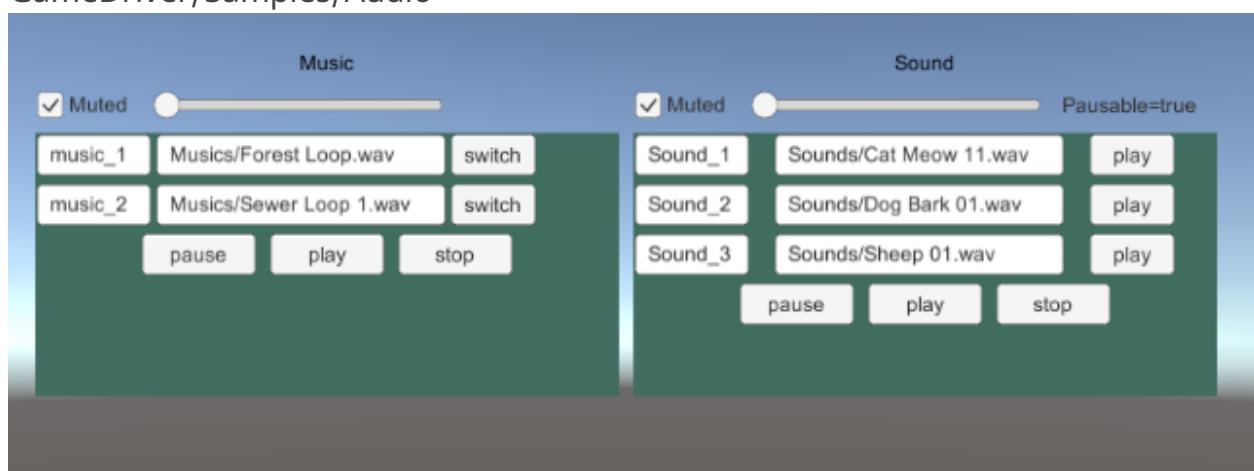
/// <summary>
/// Set sound mute
/// 设置音效静音
/// </summary>
/// <param name="muted"></param>
/// <param name="refresh"></param>
void SetSoundMuted(bool muted, bool refresh = true);
```

#### 5.2.4 建议包装组件进行使用，如：

示例中的AudioDemoMusicNode、AudioDemoSoundNode组件。

### 5.3 示例

GameDriver/Samples/Audio



## 6. 面板管理模块(PanelManager)

- **JLGames.GameDriver.Actions.Layer** 提供了容器层级管理功能。
- **JLGames.GameDriver.Games.PanelManager** 提供了面板管理功能。
- 面板管理模块的使用流程：注册 -> 展示 -> 关闭
- 若要缓存面板不销毁，建议把面板移出屏幕外。

### 6.1 注册信息

#### 6.1.1 注册层容器信息

```
PanelManagerShared.Manager.Register.RegisterLayer(containerName
    tranContainer, isDefault);
```

```
/// <summary>
/// Register panel container layer
/// 注册面板容器
/// </summary>
/// <param name="layerName"></param>
/// <param name="layer"></param>
/// <param name="default"></param>
void RegisterLayer(string layerName, Transform layer, bool @default = false);
```

- **layerName**: 为注册的层信息指定一个名称，与**layer**的**name**属性无关。
- **layer**: 层的**Transform**组件引用。
- **default**: 是否设置为默认，如果为**true**，当打开面板时不指定层名称，则选择默认层。

#### 6.1.2 注册背景处理信息

```
PanelManagerShared.Manager.Register.RegisterBackground(background
    backgroundOrigin, backgroundScript);
```

```
/// <summary>
/// Register background meta object
/// 注册背景元预制件
/// </summary>
/// <param name="key"></param>
/// <param name="origin"></param>
/// <param name="script"></param>
void RegisterBackground(string key, GameObject origin, string script);
```

- **key**: 为注册的背景信息指定一个**key**，用于标识信息的唯一性。
- **origin**: 背景的预制件，当展示面板时，克隆预制件添加到面板容器的最底层。
- **script**: 处理背景逻辑的脚本组件的类名(包含命名空间)。

### 6.1.3 注册动画信息

```
PanelManagerShared.Manager.Register.RegisterAnimator(animKey, a
```

```
/// <summary>
/// Register RuntimeAnimatorController
/// 注册动画控制器
/// </summary>
/// <param name="key"></param>
/// <param name="animator"></param>
void RegisterAnimator(string key, RuntimeAnimatorController animator);
```

- key: 为注册的动画信息指定一个Key，用于标识信息的唯一性。
- animator: 动画RuntimeAnimatorController组件。

### 6.1.4 设置面板的基础容器

```
PanelManagerShared.Manager.Register.RegisterPanelContainer(cont
```

```
/// <summary>
/// Register panel instance meta container
/// 注册实例容器
/// </summary>
/// <param name="container"></param>
void RegisterPanelContainer(GameObject container);
```

- container: 面板容器的预制件，当展示面板时，克隆预制件作为面板根节点添加到展示层中，面板的实际内容任务子节点添加到面板容器中。

### 6.1.5 注册面板信息

```
PanelManagerShared.Manager.Register.RegisterPanelInfo(panelId,
settings, maxDisplayNum, extendType);
```

```

/// <summary>
/// Register panel info
/// 注册面板信息
/// </summary>
/// <param name="info"></param>
void RegisterPanelInfo(IPanelInfo info);

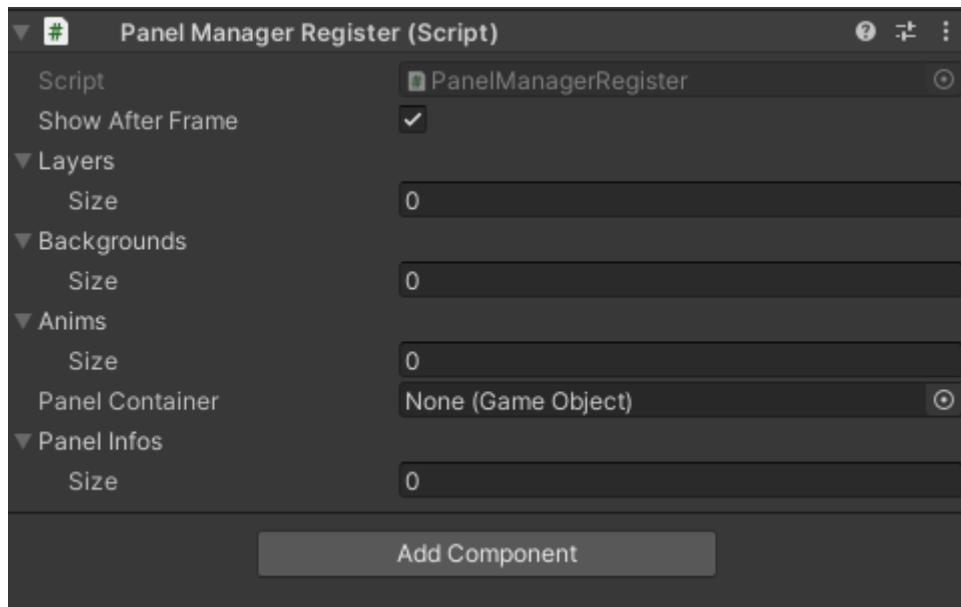
/// <summary>
/// Register panel info
/// 注册面板信息
/// </summary>
/// <param name="id"></param>
/// <param name="bundleName"></param>
/// <param name="assetPath"></param>
void RegisterPanelInfo(string id, string bundleName, string assetPath);

/// <summary>
/// Register panel info
/// 注册面板信息
/// </summary>
/// <param name="id"></param>
/// <param name="settings"></param>
/// <param name="maxDisplayNum"></param>
/// <param name="extendType">业务扩展用面板类型</param>
void RegisterPanelInfo(string id, IPanelSettings settings, int maxDisplayNum, int extendType);

```

- 面板信息最终以IPanelInfo的实例保存起来用于查找与展示。
- 项目使用时，应该读取项目面板的配置信息，并注册进来。

### 6.1.6 使用PanelManagerRegister组件，可以同时处理层容器、背景、动画以及面板信息的注册功能



#### 6.1.7 注意

**背景注册 和 动画注册 非必要**，当面板信息中有关联要求时才注册。

## 6.2 展示面板

### 6.2.1 通过面板id弹出面板，可选择指定面板层容器。

`PanelManagerShared.Manager.ShowPanel(panelId);`

```
/// <summary>
/// Show a panel.
/// 打开面板
/// </summary>
/// <param name="panelId">Panel Id<br/>面板的配置Id</param>
/// <param name="layerName">The container layer name corresponding to the panel to be displayed<br/>展示时指定的容器名称</param>
IPanelInstance ShowPanel(string panelId, string layerName = null);
```

## 6.2.2 通过面板id弹出面板，并传入参数对象，可选择指定面板层容器。

`PanelManagerShared.Manager.ShowPanel(panelId, panelParams);`

```
/// <summary>
/// Show a panel.
/// 打开面板
/// </summary>
/// <param name="panelId">Panel Id<br/>面板的配置Id</param>
/// <param name="params">Parameters injected into IParamsPanel<br/>展示时IParamsPanel注入的参数对象</param>
/// <param name="layerName">The container layer name corresponding to the panel to be displayed<br/>展示时指定的容器名称</param>
IPanelInstance ShowPanel(string panelId, object @params, string layerName = null);
```

## 6.3 高级应用

### 6.3.1 自定义注册器

IPanelManager中的注册器可自定义，只要实现IPanelRegister接口即可。

```
/// <summary>
/// Set panel register.
/// 设置注册表
/// </summary>
/// <param name="register"></param>
void SetRegister(IPanelRegister register);
```

### 6.3.2 自定义加载器

IPanelManager中的加载器可自定义，只要实现IPanelLoaderAdapter接口即可。

```
/// <summary>
/// Set asset loader adapter.
/// 设置加载器
/// </summary>
/// <param name="loader"></param>
void SetLoader(IPanelLoaderAdapter loader);
```

### 6.3.3 设置面板展示时机

IPanelManager中的SetShowMoment可以设置面板展示时机(立即展示|帧结束展示)

```
/// <summary>
/// Set show moment
/// 设置显示时机
/// </summary>
/// <param name="showAfterFrame"></param>
void SetShowMoment(bool showAfterFrame);
```

- 立即展示 代码执行时，马上进行加载处理。当资源准备完成后立即添加面板到显示节点。
- 帧结束展示 代码执行时，马上进行加载处理。当资源准备完成后，开启协程，等待 WaitForEndOfFrame后添加面板到显示节点。

### 6.3.4 IPanelSettings说明

IPanelSettings实例为面板注册时保存的面板配置信息接口，包含资源配置、背景配置和动画配置。

#### 6.3.4.1 资源配置

IPanelAssetSettings实例

主要配置属性：BundleName、AssetPath、MainScriptName、MainScriptParams

```
/// <summary>
/// Assetbundle name
/// Assetbundle名称
/// </summary>
string BundleName { get; }

/// <summary>
/// Asset path
/// 资源路径
/// </summary>
string AssetPath { get; }

/// <summary>
/// Main script full name
/// 主脚本完整类名
/// </summary>
string MainScriptName { get; }

/// <summary>
/// Json string params of script
/// 主脚本注入的参数字符串
/// </summary>
string MainScriptParams { get; }
```

- BundleName 面板资源所在的Assetbundle的名称
- AssetPath 面板资源处于Assetbundle中的路径
- MainScriptName 面板如果有主脚本，则返回主脚本的完整类名(包含命名空间)。  
没有主脚本则返回null或空字符串。
- MainScriptParams 当MainScriptName存在实现IPanelSettings接口时生效，为函数SetPanelStringParams的传入参数。

#### 6.3.4.2 背景配置

IPanelBackgroundSettings实例

- 基础参数

- OriginKey 背景信息注册时指定的key
- OriginInfo 通过背景信息注册时指定的key，查找出注册信息，取其中的origin预制件。
- Mode 背景模式支持四种：None、Color、Image、Screenshot
  - None 依赖系数：无
  - Color 依赖系数：Color
  - Image 依赖系数：ImageSprite、ImageBundle、ImagePath、BlurFactor
  - Screenshot 依赖系数：ScreenshotFactor

- 模式相关参数

- Color [Mode=Color时生效] 背景颜色值, 用于设置背景颜色值
- ImageSprite [Mode=Image时生效] 两种情况：ImageSprite不为空、ImageSprite为空。
  - ImageSprite不为空 忽略ImageBundle和ImagePath参数，使用ImageSprite填充。
  - ImageSprite为空 通过Register中的加载器，使用ImageBundle和ImagePath加载Sprite填充。
- ImageBundle和ImagePath [Mode=Image时生效] 当ImageSprite为空时使用，用回加载Sprite资源。
- BlurFactor [Mode=Image时生效] 模糊系数：[0,1]
- ScreenshotFactor [Mode=Screenshot时生效] 截屏图像模糊系数

#### 6.3.4.3 动画配置

## IPanelAnimSettings实例

```

/// <summary>
/// Open animator mapping Key
/// 展示时动画控制器映射Key
/// </summary>
string OpenKey { get; }

/// <summary>
/// Animation state in open animator mapping Key
/// 展示时指定动画State
/// </summary>
string OpenState { get; }

/// <summary>
/// Close animator mapping Key
/// 关闭时动画控制器映射Key
/// </summary>
string CloseKey { get; }

/// <summary>
/// Animation state in open animator mapping Key
/// 关闭时指定动画State
/// </summary>
string CloseState { get; }

```

- OpenKey 动画信息注册时使用的key, 用于查找动画
- OpenState 动画Animator中的状态名称, 用于播放指定动画
- CloseKey 动画信息注册时使用的key, 用于查找动画
- CloseState 动画Animator中的状态名称, 用于播放指定动画

### 6.3.5 面板功能扩展

现阶段有4个接口与面板功能扩展相关:

IInitPanel、IParamsPanel、IReloadPanel、IDisposePanel

**注意:** IShowPanel、IClosePanel已弃用。

- IInitPanel 用于面板初始化, 调用时机在OnEnable之后

```

/// <summary>
/// Show panel execution timing: Awake > OnEnable > IShowPanel > IInitPanel > IParamsPanel > IReloadPanel > Start
/// 展示面板执行时机: Awake > OnEnable > IShowPanel > IInitPanel > IParamsPanel > IReloadPanel > Start
/// </summary>
public interface IInitPanel
{
    /// <summary>
    /// 初始化
    /// </summary>
    /// <param name="instance"></param>
    void InitPanel(IPanelInstance instance);
}

```

- IParamsPanel 用于面板注入参数。

```
/// <summary>
/// Show panel execution timing: Awake > OnEnable > IShowPanel > IInitPanel > IParamsPanel > IRefreshPanel > Start
/// 展示面板执行时机: Awake > OnEnable > IShowPanel > IInitPanel > IParamsPanel > IRefreshPanel > Start
/// </summary>
public interface IParamsPanel
{
    /// <summary>
    /// 设置面板参数
    /// </summary>
    /// <param name="instance"></param>
    /// <param name="str"></param>
    void SetPanelStringParams(IPanelInstance instance, string str);

    /// <summary>
    /// 设置面板参数
    /// </summary>
    /// <param name="instance"></param>
    /// <param name="params"></param>
    void SetPanelObjectParams(IPanelInstance instance, object @params);
}
```

- SetPanelObjectParams 当调用ShowPanel时传入参数后被触发调用。
- SetPanelStringParams 触发要求为同时具备以下两个条件:
  1. 调用ShowPanel时没有传入参数。
  2. 注册面板信息时IPanelAssetSettings配置中MainScriptParams不为空。传入参数为MainScriptParams的值。
- IRefreshPanel 用于面板刷新逻辑不适合放在Start或Awake中时使用。

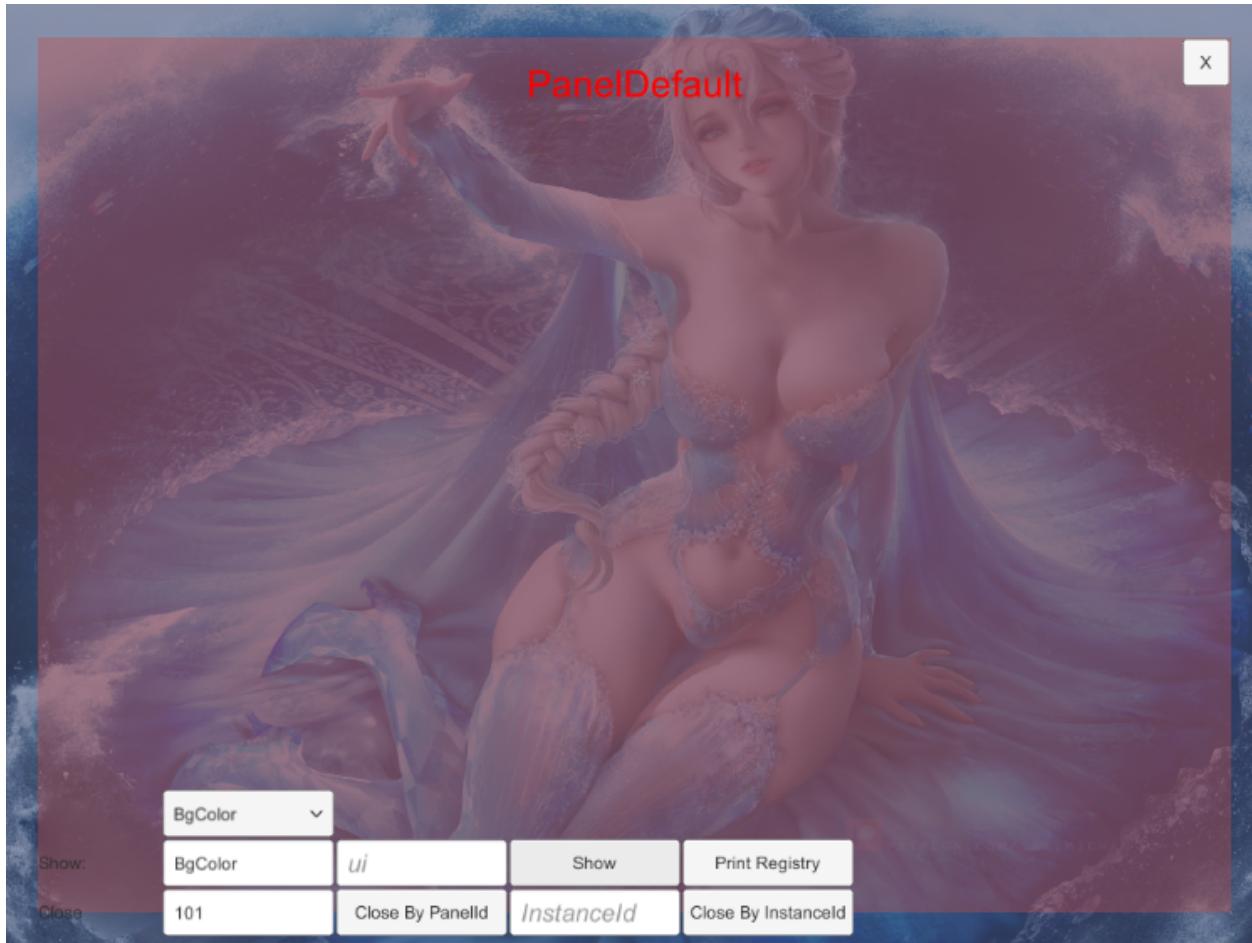
```
/// <summary>
/// Show panel execution timing: Awake > OnEnable > IShowPanel > IInitPanel > IParamsPanel > IRefreshPanel > Start
/// 展示面板执行时机: Awake > OnEnable > IShowPanel > IInitPanel > IParamsPanel > IRefreshPanel > Start
/// </summary>
public interface IRefreshPanel
{
    /// <summary>
    /// 刷新面板
    /// </summary>
    /// <param name="instance"></param>
    void RefreshPanel(IPanelInstance instance);
}
```

- IDisposePanel 当面板即将被销毁前调用，用于执行释放行为。

```
/// <summary>
/// Remove panel execution timing: IDisposePanel > IClosePanel > OnDisable > OnDestroy
/// 移除面板执行时机: IDisposePanel > IClosePanel > OnDisable > OnDestroy
/// </summary>
public interface IDisposePanel
{
    /// <summary>
    /// 销亡处理
    /// </summary>
    /// <param name="instance"></param>
    void DisposePanel(IPanelInstance instance);
}
```

## 6.4 示例

## GameDriver/Samples/Panel



## 7. 服务框架(Service)

- **JLGames.GameDriver.CSharp.Service** 提供了服务框架的基础接口与行为规范。
- **JLGames.GameDriver.Actions.Service** 提供了针对Unity的服务配置资产(未启用)。
- **JLGames.GameDriver.Games.Service** 提供了常用的服务功能。

### 7.1 设计目的

- 实现业务模块逻辑分离
  - 不同业务间的逻辑分离
  - 同一业务下数据与显示的逻辑分离
- 规范化业务模块的扩展支持
- 统一跨业务行为调用的规范

### 7.2 设计思路

- 按照IOC思想，扩展业务服务以注入的形式加入到框架管理。
- 按照游戏引擎的设计思路，给扩展业务服务加入了固定时机调用的接口函数，用于服务内部的初始化数据、释放数据或特殊逻辑实现。
- 全部扩展服务都由框架进行管理，方便于初始化、更新、释放的统一处理。

## 7.3 接口说明

### 7.3.1 服务基础

- IService 只有实现的IService接口的类，才能识别为服务并加入到框架管理。ServiceName属性返回服务名称，要求具有**唯一性**。
- IEventDispatcher 服务的时机管理依赖于**事件模块**，因此服务实现类要求实现IEventDispatcher接口。
- ServiceBase 实现服务类的一些必要功能，可用于继承。
- IProgressingService 提供**进度更新**的服务接口。

当服务实例抛出**ServiceEvents.OnServiceProcessing**事件后触发一次进度更新。

服务实例要求提供**进度总量与进度当前量**的属性函数。

```
/// <summary>
/// Progress metering interface
/// 进度计量接口
/// </summary>
public interface IProgressingService : IEventDispatcher
{
    /// <summary>
    /// Total progress
    /// 进度总量
    /// [0,int.MaxValue)
    /// </summary>
    uint ProgressingLen { get; }

    /// <summary>
    /// Current amount of progress
    /// 进度当前量
    /// [0,Total]
    /// </summary>
    uint ProgressingCurrent { get; }
}
```

### 7.3.2 服务初始化相关

执行时机：IArgumentService -> IAwakableService -> IInitService

- IArgumentService 提供**参数注入**的服务接口。  
函数执行完成代表**注入参数**这一过程结束。

```
/// <summary>
/// Data injectioin interface
/// 注入数据接口
/// </summary>
public interface IArgumentService
{
    /// <summary>
    /// Inject data.
    /// 注入数据
    /// </summary>
    /// <param name="args"></param>
    void InjectArgument(object[] args);
}
```

- IAwakableService 提供**激活**逻辑的服务接口。

函数执行完成代表**激活**这一过程结束。

```
/// <summary>
/// Service Awaka interface
/// 服务激活接口
/// </summary>
public interface IAwakableService
{
    /// <summary>
    /// Awake service
    /// 激活Service
    /// Async is not allowed
    /// 不允许使用异步
    /// </summary>
    void Awake();
}
```

- IInitService 提供**初始化**逻辑的服务接口。

当服务实例抛出**ServiceEvents.OnServiceInitiated**事件后代表**初始化**这一过程结束。

```
/// <summary>
/// Service initialization interface
/// 服务初始化接口
/// Only when the current interface is implemented and configured into ServiceConfig,
/// the init method will be executed during the initialization process
/// 只有实现了当前接口，并配置到ServiceConfig中时，在初始化过程中才会执行init方法
/// </summary>
public interface IInitService : IService, IEventDispatcher
{
    /// <summary>
    /// Whether the initialization has been completed
    /// 是否已经完成初始化
    /// </summary>
    bool IsInitiated { get; }

    /// <summary>
    /// Initialize base data
    /// 初始化基础数据
    /// </summary>
    void Init();
}
```

- IInitDataService 提供**数据初始化**逻辑的服务接口。

当服务实例抛出**ServiceEvents.OnServiceDataInitiated**事件后代表**数据初始化**这

一过程结束.

```
/// <summary>
/// Progress metering interface
/// 进度计量接口
/// </summary>
public interface IProgressingService : IEventDispatcher
{
    /// <summary>
    /// Total progress
    /// 进度总量
    /// [0,int.MaxValue)
    /// </summary>
    uint ProgressingLen { get; }

    /// <summary>
    /// Current amount of progress
    /// 进度当前量
    /// [0,Total]
    /// </summary>
    uint ProgressingCurrent { get; }
}
```

### 7.3.3 管理调度相关

- IClearService 提供**清理**逻辑的服务接口。  
多用于数据重置、事件移除等逻辑。  
函数执行完成代表清理这一过程结束。

```
/// <summary>
/// Reset service interface, providing reset service to initialized state
/// 重置服务接口，提供把服务重置到初始化状态
/// </summary>
public interface IClearService
{
    /// <summary>
    /// reset
    /// 重置
    /// Clear events, clear timers, etc.
    /// 清除事件、清除计时器等
    /// </summary>
    void Clear();
}
```

- ILoadDataService 提供**数据加载**逻辑的服务接口。  
当服务实例抛出**ServiceEvents.OnServiceLoaded**事件后代表**数据加载**这一过程

结束。

```
/// <summary>
/// Load data processing interface
/// 加载数据处理接口
/// </summary>
public interface ILoadDataService : IEventDispatcher
{
    /// <summary>
    /// Load Data
    /// 加载数据
    /// </summary>
    void LoadData();
}
```

- ISaveDataService 提供**数据保存**逻辑的服务接口。

当服务实例抛出**ServiceEvents.OnServiceSaved**事件后代表**数据保存**这一过程结束。

```
/// <summary>
/// Save data processing interface
/// 保存数据处理接口
/// </summary>
public interface ISaveDataService : IEventDispatcher
{
    /// <summary>
    /// Save data
    /// 保存数据
    /// </summary>
    void SaveData();
}
```

## 7.3.4 事件说明

### 7.3.4.1 进度更新相关事件

```
/// <summary>
/// A single service initializes the progress update event, which is dispatched by the service instance.
/// The service instance must be an implementation class of the IProgressingService interface
/// 单个服务初始化进度更新事件，由服务实例调度。服务实例必须为IProgressingService接口的实现类
/// </summary>
public const string OnServiceProcessing = "ServiceEvents.OnServiceProcessing";

/// <summary>
/// Service initialization process progress update
/// 服务初始化进程进度更新
/// </summary>
public const string OnInitializationProcessing = "ServiceEvents:OnInitializationProcessing";

/// <summary>
/// Service initialization process completed
/// 服务初始化进程完成
/// </summary>
public const string OnInitializationFinish = "ServiceEvents:OnInitializationFinish";
```

- ServiceEvents.OnServiceProcessing
  - 调度主体：实现了IProgressingService接口的服务实例
  - 调度时机：有必要告知监听者更新进度时。

- ServiceEvents.OnInitializationProcessing
  - 调度主体: ServiceManager
  - 调度时机: 调用StartInitialization到endCall初始执行期间, 捕获到OnServiceProcessing、OnServiceInitied、OnServiceDataInitied三类事件时调度。
  
- ServiceEvents.OnInitializationFinish
  - 调度主体: ServiceManager
  - 调度时机: 调用StartInitialization结束时, 在endCall执行之后。

#### 7.3.4.2 服务准备相关事件

```

/// <summary>
/// Service argument injection result event, dispatched by ServiceManager.
/// Succ=true when the service implements the IArgumentService interface.
/// 服务参数注入结果事件, 由ServiceManager调度。当服务实现IArgumentService接口时, Succ=true。
/// Event data format: ServiceResultEventData
/// 事件数据格式: ServiceResultData
/// </summary>
public const string OnServiceInjected = "ServiceEvents:OnServiceInjected";

/// <summary>
/// All service argument injection completion event
/// 全部服务参数注入完成事件
/// Event data format: null
/// 事件数据格式: null
/// </summary>
public const string OnServiceAllInjected = "ServiceEvents:OnServiceAllInjected";

/// <summary>
/// Service activation result event, dispatched by ServiceManager.
/// Succ=true when the service implements the IWakableService interface.
/// 服务激活结果事件, 由ServiceManager调度。当服务实现IWakableService接口时, Succ=true。
/// Event data format: ServiceResultEventData
/// 事件数据格式: ServiceResultData
/// </summary>
public const string OnServiceAwaked = "ServiceEvents:OnServiceAwaked";

/// <summary>
/// All service activation completion event
/// 全部服务激活完成事件
/// Event data format: null
/// 事件数据格式: null
/// </summary>
public const string OnServiceAllAwaked = "ServiceEvents:OnServiceAllAwaked";

```

调度顺序: OnServiceInjected > OnServiceAllInjected > OnServiceAwaked > OnServiceAllAwaked

- ServiceEvents.OnServiceInjected
  - 调度主体: ServiceManager
  - 调度时机: 调用StartInitialization期间
  
- ServiceEvents.OnServiceAllInjected
  - 调度主体: ServiceManager
  - 调度时机: 调用StartInitialization期间, 全部实现IArgumentService的服务处理完成后。

- ServiceEvents.OnServiceAwaked
  - 调度主体: ServiceManager
  - 调度时机: 调用StartInitialization期间
  
- ServiceEvents.OnServiceAllAwaked
  - 调度主体: ServiceManager
  - 调度时机: 调用StartInitialization期间, 全部实现IAwakableService的服务处理完成后。

#### 7.3.4.3 服务初始化相关事件

```

/// <summary>
/// Single service initialization start event, dispatched by ServiceManager
/// 单个服务初始化开始事件, 由ServiceManager调度
/// Event data format: ServiceResultData
/// 事件数据格式: ServiceResultData
/// </summary>
public const string OnServiceInitStart = "ServiceEvents:OnServiceInitStart";

/// <summary>
/// A single service initialization complete event, which are dispatched by the service instance.
/// Re-dispatched after being captured by ServiceManager.
/// 单个服务初始化完成事件, 由服务实例调度。被ServiceManager捕获后重新调度。
/// Event data format: {named:string}
/// 事件数据格式: {named:string}
/// </summary>
public const string OnServiceInitiated = "ServiceEvents:OnServiceInitiated";

/// <summary>
/// All service initialization complete event
/// 全部服务初始化完成事件
/// Event data format: len:int
/// 事件数据格式: len:int
/// </summary>
public const string OnServiceAllInitiated = "ServiceEvents:OnServiceAllInitiated";

/// <summary>
/// Single service data initialization start event, dispatched by ServiceManager
/// 单个服务数据初始化开始事件, 由ServiceManager调度
/// Event data format: ServiceResultData
/// 事件数据格式: ServiceResultData
/// </summary>
public const string OnServiceDataInitStart = "ServiceEvents:OnServiceDataInitStart";

/// <summary>
/// A single service data initialization complete event, which are dispatched by the service instance.
/// Re-dispatched after being captured by ServiceManager.
/// 单个服务数据初始化完成事件, 由服务实例调度。被ServiceManager捕获后重新调度。
/// Event data format: {named:string}
/// 事件数据格式: {named:string}
/// </summary>
public const string OnServiceDataInitiated = "ServiceEvents:OnServiceDataInitiated";

/// <summary>
/// All service data initialization complete event
/// 全部服务数据初始化完成事件
/// Event data format: len:int
/// 事件数据格式: len:int
/// </summary>
public const string OnServiceDataAllInitiated = "ServiceEvents:OnServiceDataAllInitiated";

```

调度顺序: OnServiceInitStart > OnServiceInitiated > OnServiceAllInitiated > OnServiceDataInitStart > OnServiceDataInitiated > OnServiceDataAllInitiated

- ServiceEvents.OnServiceInitStart
  - 调度主体: ServiceManager

- 调度时机：调用StartInitialization期间，处理每一个服务关于IInitService接口行为之前。
- ServiceEvents.OnServiceInitiated
  - 调度主体：实现IInitService的服务实例、ServiceManager
  - 调度时机：实现IInitService的服务实例处理Init行为完成后；ServiceManager捕获前者事件后重新调度。
- ServiceEvents.OnServiceAllInitiated
  - 调度主体：ServiceManager
  - 调度时机：调用StartInitialization期间，全部实现IInitService的服务处理完成后。
- ServiceEvents.OnServiceDataInitStart
  - 调度主体：ServiceManager
  - 调度时机：调用StartInitialization期间，处理每一个服务关于IInitDataService接口行为之前。
- ServiceEvents.OnServiceDataInitiated
  - 调度主体：实现IInitDataService的服务实例、ServiceManager
  - 调度时机：实现IInitDataService的服务实例处理InitData行为完成后；ServiceManager捕获前者事件后重新调度。
- ServiceEvents.OnServiceDataAllInitiated
  - 调度主体：ServiceManager
  - 调度时机：调用StartInitialization期间，全部实现IInitDataService的服务处理完成后。

#### 7.3.4.4 数据加载相关事件

```
/// <summary>
/// A single data service load data start event, dispatched by ServiceManager
/// Succ=true when the service implements the ILoadDataService interface.
/// 单个服务数据加载数据开始事件，由ServiceManager调度。当服务实现ILoadDataService接口时，Succ=true。
/// Event data format: ServiceResultData
/// 事件数据格式: ServiceResultData
/// </summary>
public const string OnServiceDataLoadStart = "ServiceEvents:OnServiceDataLoadStart";

/// <summary>
/// A single data service load data completion event, which are dispatched by the service instance.
/// Re-dispatched after being captured by ServiceManager.
/// 单个数据服务加载数据完成事件，由服务实例调度。被ServiceManager捕获后重新调度。
/// Event data format: {named:string}
/// 事件数据格式: {named:string}
/// </summary>
public const string OnServiceDataLoaded = "ServiceEvents:OnServiceDataLoaded";

/// <summary>
/// All data service loading data completion event
/// 全部数据服务加载数据完成事件
/// Event data format: len:int
/// 事件数据格式: len:int
/// </summary>
public const string OnServiceDataAllLoaded = "ServiceEvents:OnServiceDataAllLoaded";
```

- ServiceEvents.OnServiceDataLoadStart
  - 调度主体: ServiceManager
  - 调度时机: 调用LoadServicesData期间, 处理每一个服务关于 ILoadDataService接口行为之前。
  
- ServiceEvents.OnServiceDataLoaded
  - 调度主体: 实现ILoadDataService的服务实例、ServiceManager
  - 调度时机: 实现ILoadDataService的服务实例处理LoadData行为完成后; ServiceManager捕获前者事件后重新调度。
  
- ServiceEvents.OnServiceDataAllLoaded
  - 调度主体: ServiceManager
  - 调度时机: 调用LoadServicesData期间, 全部实现ILoadDataService的服务处理完成后。

#### 7.3.4.5 数据保存相关事件

```
/// <summary>
/// A single data service save data start event, dispatched by ServiceManager
/// Succ=true when the service implements the ISaveDataService interface.
/// 单个服务数据保存数据开始事件, 由ServiceManager调度。当服务实现ISaveDataService接口时, Succ=true。
/// Event data format: ServiceResultData
/// 事件数据格式: ServiceResultData
/// </summary>
public const string OnServiceDataSaveStart = "ServiceEvents:OnServiceDataSaveStart";

/// <summary>
/// A single data service saves data completion events, which are dispatched by the service instance.
/// Re-dispatched after being captured by ServiceManager.
/// 单个数据服务保存数据完成事件, 由服务实例调度。被ServiceManager捕获后重新调度。
/// Event data format: {named:string}
/// 事件数据格式: {named:string}
/// </summary>
public const string OnServiceDataSaved = "ServiceEvents:OnServiceDataSaved";

/// <summary>
/// All data services save data complete event
/// 全部数据服务保存数据完成事件
/// Event data format: len:int
/// 事件数据格式: len:int
/// </summary>
public const string OnServiceDataAllSaved = "ServiceEvents:OnServiceDataAllSaved";
```

- ServiceEvents.OnServiceDataSaveStart
  - 调度主体: ServiceManager
  - 调度时机: 调用SaveServicesData期间, 处理每一个服务关于 ISaveDataService接口行为之前。
  
- ServiceEvents.OnServiceDataSaved
  - 调度主体: 实现ISaveDataService的服务实例、ServiceManager
  - 调度时机: 实现ISaveDataService的服务实例处理SaveData行为完成后; ServiceManager捕获前者事件后重新调度。
  
- ServiceEvents.OnServiceDataAllSaved
  - 调度主体: ServiceManager

- 调度时机：调用SaveServicesData期间，全部实现ISaveDataService的服务处理完成后。

## 7.4 如何使用框架

在获取服务使用前，必须先注册到框架管理，然后初始化后才能使用。这是为了确保数据已经完整地处理完成。

### 7.4.1 把服务注册到框架管理

通过调用ServiceConfig实例中的AddConfig函数，能把服务注册到框架管理中去。

```
ServiceConfig.Shared.AddConfig(new ServiceInfo(serviceName,
serviceImpl, args));
```

```
/// <summary>
/// Add a service to the end of the configuration list
/// 添加服务到配置列表尾部
/// </summary>
/// <param name="sc"></param>
/// <param name="ignoreSame"></param>
/// <exception cref="Exception"></exception>
public void AddConfig(ServiceInfo sc, bool ignoreSame = true)
{
    if (!ignoreSame && ContainsService(sc.ServiceName))
    {
        throw new Exception($"重复的ServiceName:{sc.ServiceName}");
    }

    m_Config.Add(sc);
    scServiceImpl.ServiceName = sc.ServiceName;
}
```

**建议：**

1. 使用一个专门处理注册服务的类管理注册逻辑。如示例中的ServiceRegister。
2. 注册逻辑调用应该保证唯一。

### 7.4.2 服务初始化

通过调用ServiceManager实例中的StartInitialization函数，开始初始化。

**注意：** 初始化开始后不应该重复调用。

```

/// <summary>
/// Initialize the configured service
/// 初始化配置好的服务
/// If it has been initialized once, try to execute the callback directly
/// 如果已经初始化一次，就尝试直接执行回调
/// </summary>
/// <param name="endCall"></param>
public void StartInitialization(Callback endCall)
{
    m_Services = ServiceConfig.Shared.ServiceInfos;
    m_FinishCall = endCall;

    InjectToServices();
    ActiveServices();
    m_ProcessingLen = GetServiceTotalLen();
    AddProcessingEvents();

    StartInitServices();
}

```

### 7.4.3 服务调用

在全部服务初始完成后，通过调用ServiceConfig中的GetService函数，可取得服务实例。

ServiceConfig中获取服务实例的函数有多个，但最终调用的都是GetService函数。

```

/// <summary>
/// 通过ServiceName取得Service实例
/// </summary>
/// <param name="serviceName"></param>
/// <returns></returns>
public static object GetService(string serviceName)
{
    return ServiceConfig.Shared.GetServiceImpl(serviceName);
}

```

**建议：**

1. 使用一个专门管理获取服务实例的类管理获取逻辑。如示例中的ServiceCenter。

### 7.4.4 服务清理

当游戏要实现软重启时，有必要重置全部的服务再重新初始化。

通过调用ServiceManager中的ClearServices，可实现全部服务的重置。

只有实现IClearService接口的服务才会执行重置逻辑。

```

/// <summary>
/// Service cleanup
/// 服务清理
/// </summary>
public void ClearServices()
{
    RemoveEventListener();
    for (var i = m_Services.Length - 1; i >= 0; i--)
    {
        (m_Services[i].ServiceImpl as IClearService)?.Clear();
    }
}

```

```

/// <summary>
/// Reset service interface, providing reset service to initialized state
/// 重置服务接口，提供把服务重置到初始化状态
/// </summary>
public interface IClearService
{
    /// <summary>
    /// reset
    /// 重置
    /// Clear events, clear timers, etc.
    /// 清除事件、清除计时器等
    /// </summary>
    void Clear();
}

```

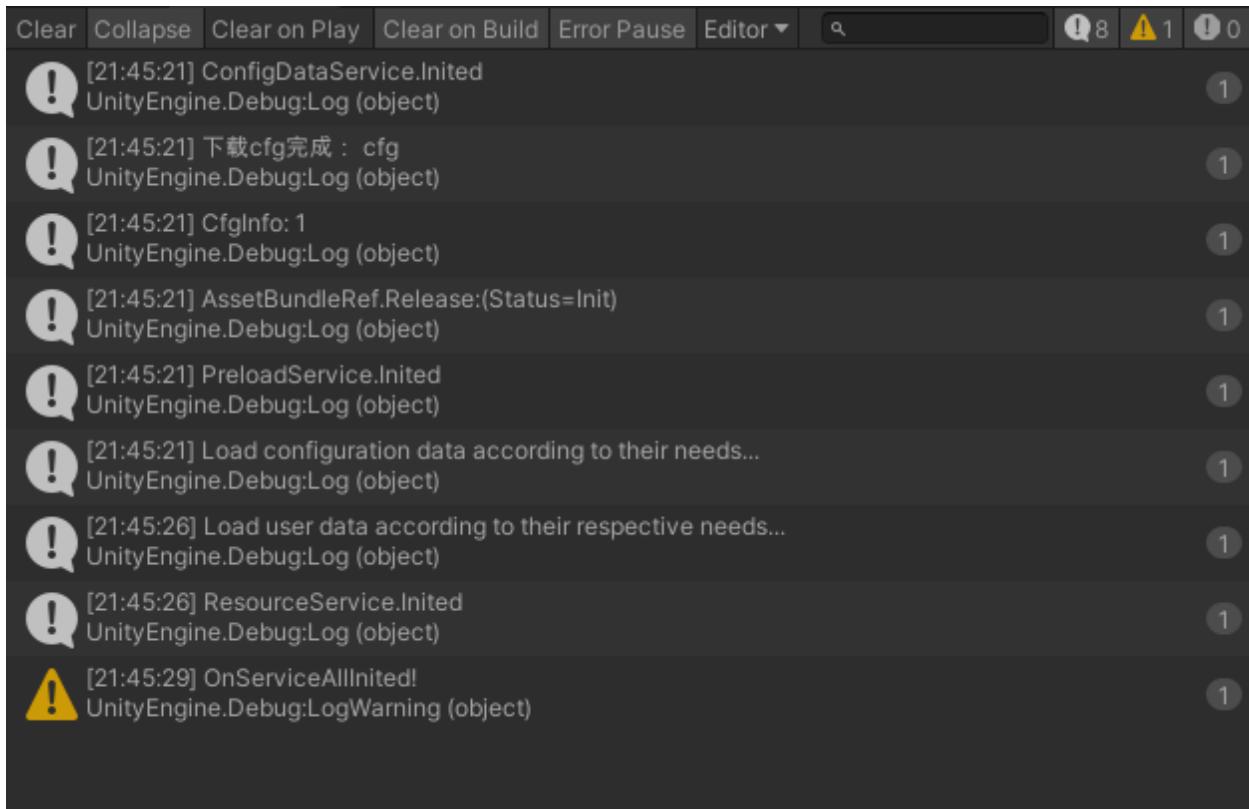
**建议：**

1. 每个服务都建议实现IClearService接口。
2. 在IClearService接口中的Clear函数中把事件监听也全部清除。

## 7.5 示例

GameDriver/Samples/Service

| Name                      | IArgumentService | IAwakableService | IInitService | IInitDataService |
|---------------------------|------------------|------------------|--------------|------------------|
| ServiceNames.ConfigData   | Disable          | Finish           | Finish       | Waiting          |
| ServiceNames.Preloaded    | Disable          | Finish           | Finish       | Waiting          |
| ServiceNames.Material_1   | Disable          | Disable          | Finish       | Waiting          |
| ServiceNames.DemoService1 | Finish           | Disable          | Finish       | Waiting          |
| ServiceNames.DemoService2 | Finish           | Disable          | Doing        | Waiting          |
| ServiceNames.DemoService3 | Finish           | Disable          | Waiting      | Waiting          |
| ServiceNames.DemoService4 | Finish           | Disable          | Waiting      | Waiting          |



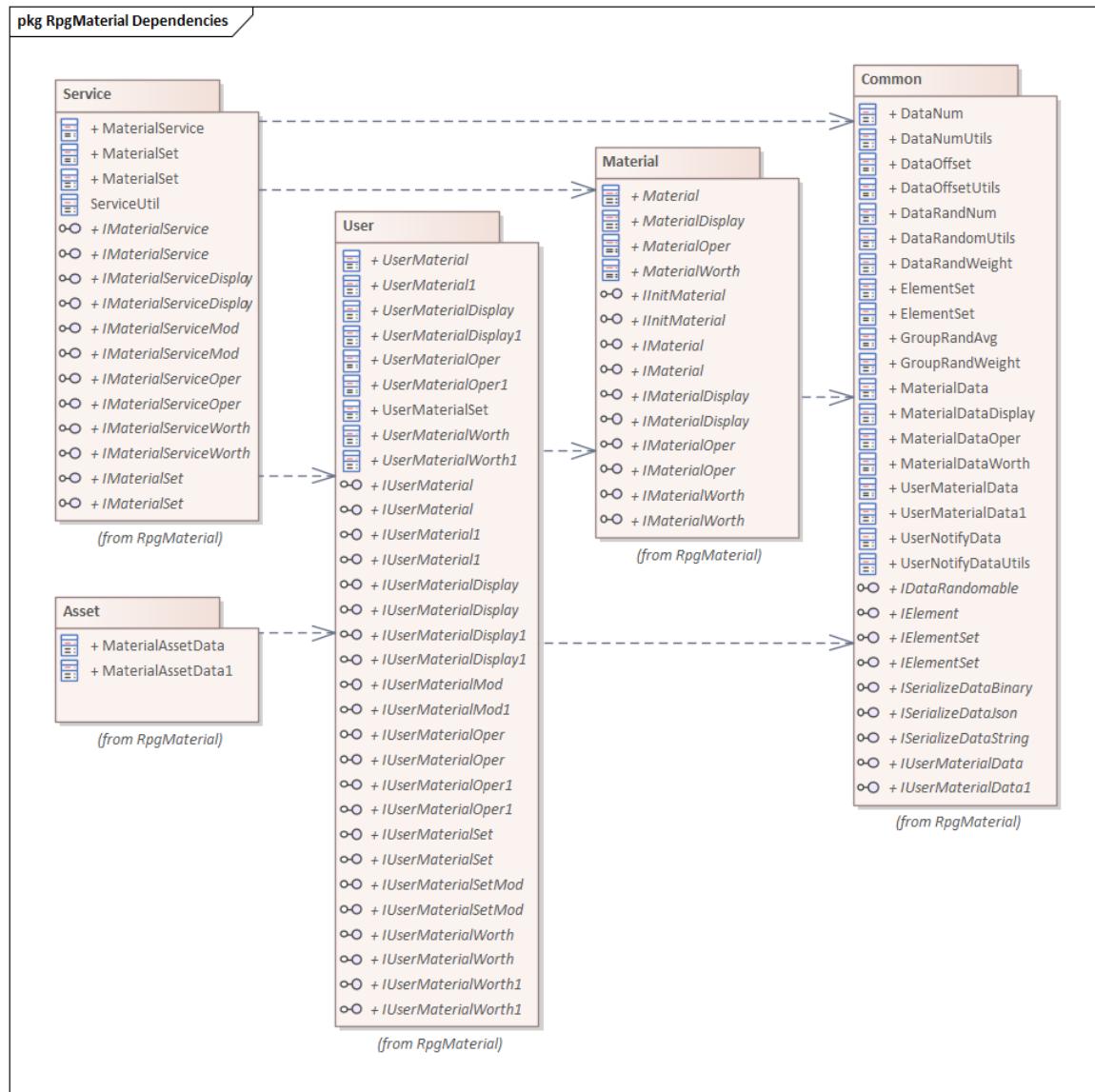
示例中的服务说明：

- ConfigDataService
  1. 使用ExcelExport导出了Excel表头和数据。
  2. cfg.asset是DirFilesIndex资产管理器，用于管理Excel的Json数据的路径。
  3. 使用内置的Loader加载cfg.asset和Json文件，并缓存起来。
  4. Init处理完成。
- PreloadService
  1. 使用PreloadSettings管理预加载资源的配置信息。
  2. 使用内置的Loader加载PreloadSettings资产。
  3. 根据PreloadSettings配置信息，加载资源。
  4. Init处理完成。
- ResourceService 材料数据框架下的服务实现。
  1. 从ConfigDataService服务中读取配置。
  2. 通过配置初始化。
  3. Init处理完成。
  4. 模拟用户数据的生成。
  5. InitData处理完成。
- DemoService 样板服务，模拟了通过注入参数，在同一个服务逻辑下表现不同的功能。

## 8. Rpg材料数据系统

这是一套通用的用户数据管理系统。  
针对玩家的数值型数据，以KTV(Key-Type-Value)形式进行管理。

- **JLGames.GameDriver.Games.RpgMaterial** 提供了Rpg材料数据系统的功能支持。依赖关系如下：



- **JLGames.GameDriver.Games.RpgMaterial.Common** 系统的基础数据结构。
- **JLGames.GameDriver.Games.RpgMaterial.Material** 系统中关于材料定义的数据结构。
- **JLGames.GameDriver.Games.RpgMaterial.User** 系统中关于用户数据存储的数据结构。
- **JLGames.GameDriver.Games.RpgMaterial.Service** 系统中关于对外提供接口支持的数据结构。
- **JLGames.GameDriver.Games.RpgMaterial.Asset** 系统中针对Unity进行序列化的数据结构。

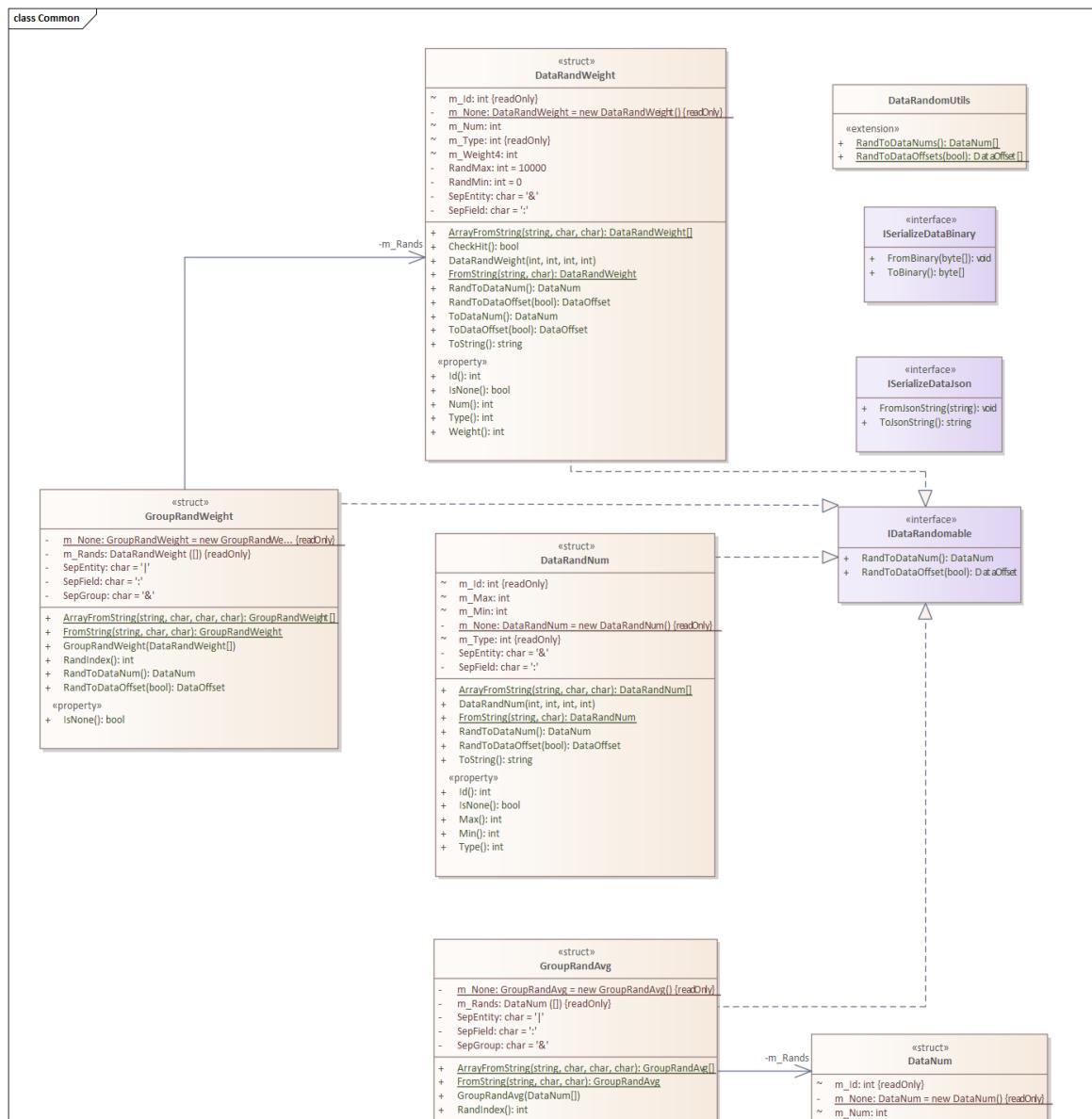
## 8.1 设计思路

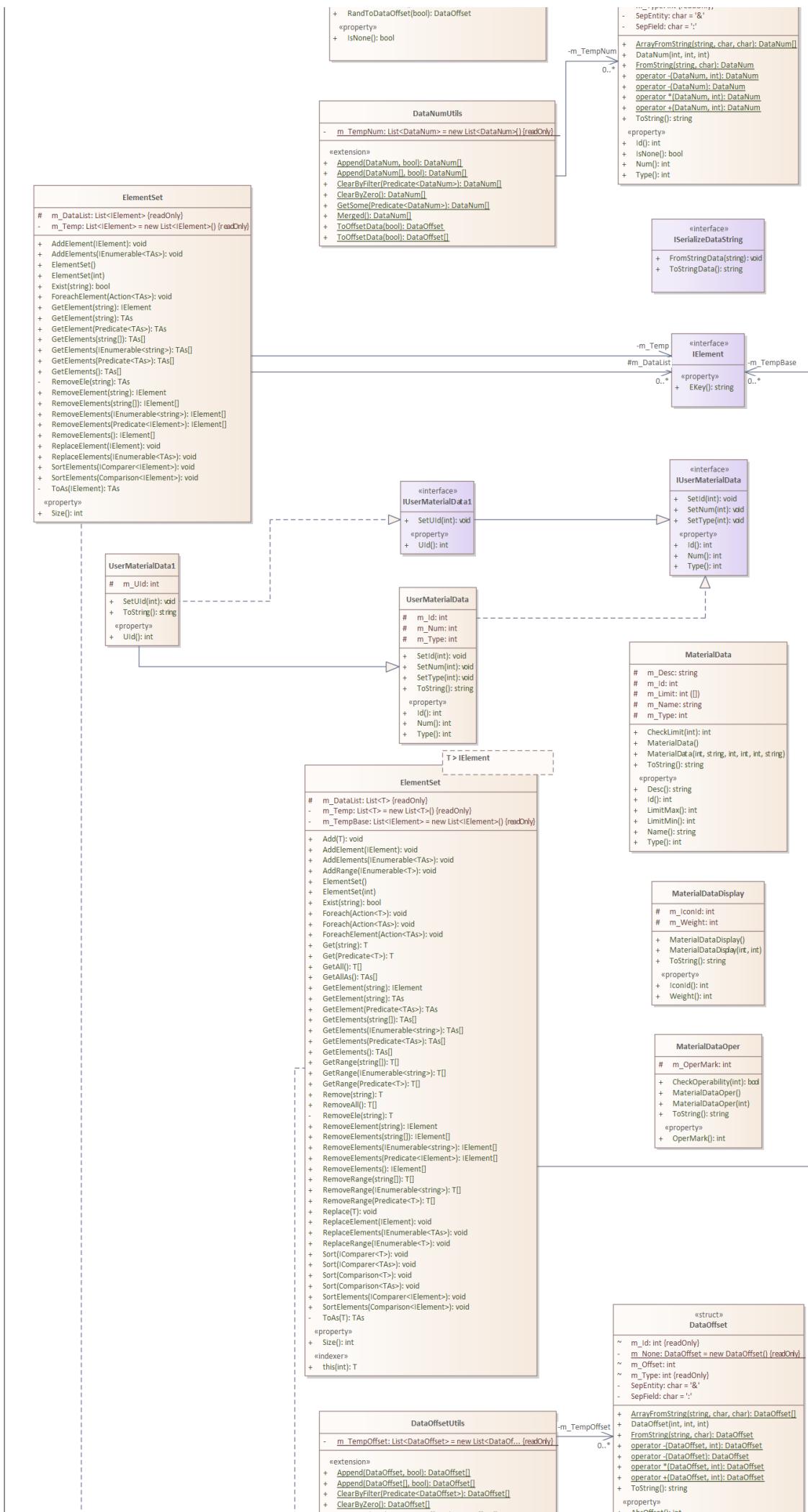
- 用户数据的元数据(配置数据)以 **KTD(Key-Type-Define)** 格式进行管理。

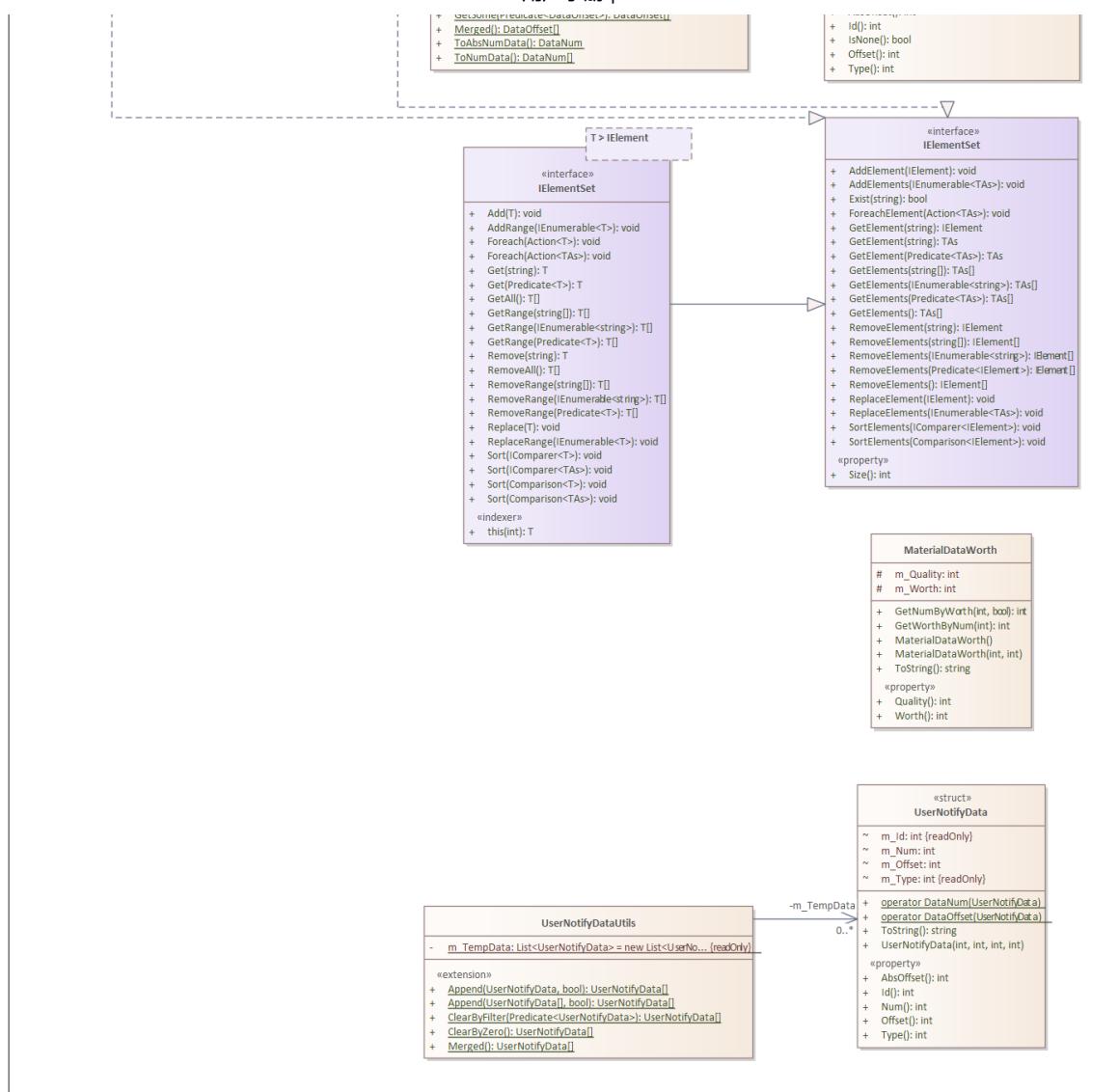
- 元数据(配置数据)的来源可以是数值表导出、服务器获取、资产配置等方式得到。
- 通过 **KT(Key&Type)** 能得到元数据(配置数据)定义(Define)，并且可以在用户数据设置时进行有效性检查。
- 用户数据以 **KTV(Key-Type-Value)** 格式进行管理(内存、本地文件、服务器等)。
- 同一条元数据定义下，如果用户数据要求有多条，则采用 **KTUV(Key-Type-UId-Value)** 形式进行保存，既兼容了普通数据格式，也支持了多条数据的要求。
- 用户数据(Value)通常情况下使用基础数值类型则满足需求，当然游戏设计者可以扩展数据的存储结构。
- 通过 **KT(Key&Type)** 或 **KTU(Key&Type&UId)** 能得到玩家的具体数据(Value)，用于运算与展示。
- 用户数据的变更以通知的形式下发，支持增量变更与直接更新两种。

### 8.1.1 Common模块设计说明

- 提供数据对象的基础管理功能，并定义了数据对象的最基础的接口。
- 提供最基础的配置数据接口定义和相关的数据结构实现。
- 提供最基础的用户数据接口定义和相关的数据结构实现。
- 提供可选的扩展接口定义。
- 提供用于基础计算的数据结构实现。







#### 8.1.1.1 Common接口设计

- IElement 数据对象基础接口
  - IElementSet 数据集合基础接口，用于管理IElement对象。
  - IUserMaterialData和IUserMaterialData 用户数据相关接口。
  - ISerializeDataBinary、ISerializeDataJson和ISerializeDataString 数据序列化相关接口。
  - IDataRandomable 数量随机相关接口。

### 8.1.1.2 Common数据结构设计

- ElementSet IElementSet的实现类，提供对IElement对象的管理功能，可用于继承或组合。
  - 配置数据相关数据结构
    - MaterialData 配置数据基础，包含Id, 名称、类型、大小约束、描述。
    - MaterialDataDisplay 配置数据显示相关数据，包含图标Id、显示权重。
    - MaterialDataOper 配置数据操作行为设定，包含按位定义值。

### 操作行为设定：

- 假设二进制第1位定义为可使用，第2位定义为可丢弃。
  - 当值为二进制(11)时，代表数据定义即可使用也可丢弃。
- MaterialDataWorth 配置数据价值定义相关数据，包含价值、品质。
- 价值定义：**
- 类型市场商品标出的单价。
  - 用于不同配置间数量比较与计算。
- 用户数据相关数据结构
    - UserMaterialData 用户数据基础，包含Id(对应MaterialData的Id)、类型(对应MaterialData的类型)、数量
    - UserMaterialData1 继承于UserMaterialData,, 增加唯一Id这个属性
  - 计算和通知相关数据结构
    - DataNum
      - Id、类型 用于定位配置数据项与用户数据项
      - 数量 用于覆盖用户数据项中的数量
    - DataOffset
      - Id、类型 用于定位配置数据项与用户数据项
      - 偏移值 用于与用户数据项中的数量相加，结果覆盖用户数据项中的数量
    - UserNotifyData 通过DataNum、DataOffset数据进行更新用户数据后，可计算出UserNotifyData结果，用于通知前端显示效果。
  - 随机相关数据结构
    - DataRandNum 在最小值与最大值间进行均值随机。
    - DataRandWeight 针对万分权值进行随机，只有真和假两个结果。
    - GroupRandAvg 多个数值中随机一个，概率平均。
    - GroupRandWeight 多个数值中随机一个，概率依赖权值。

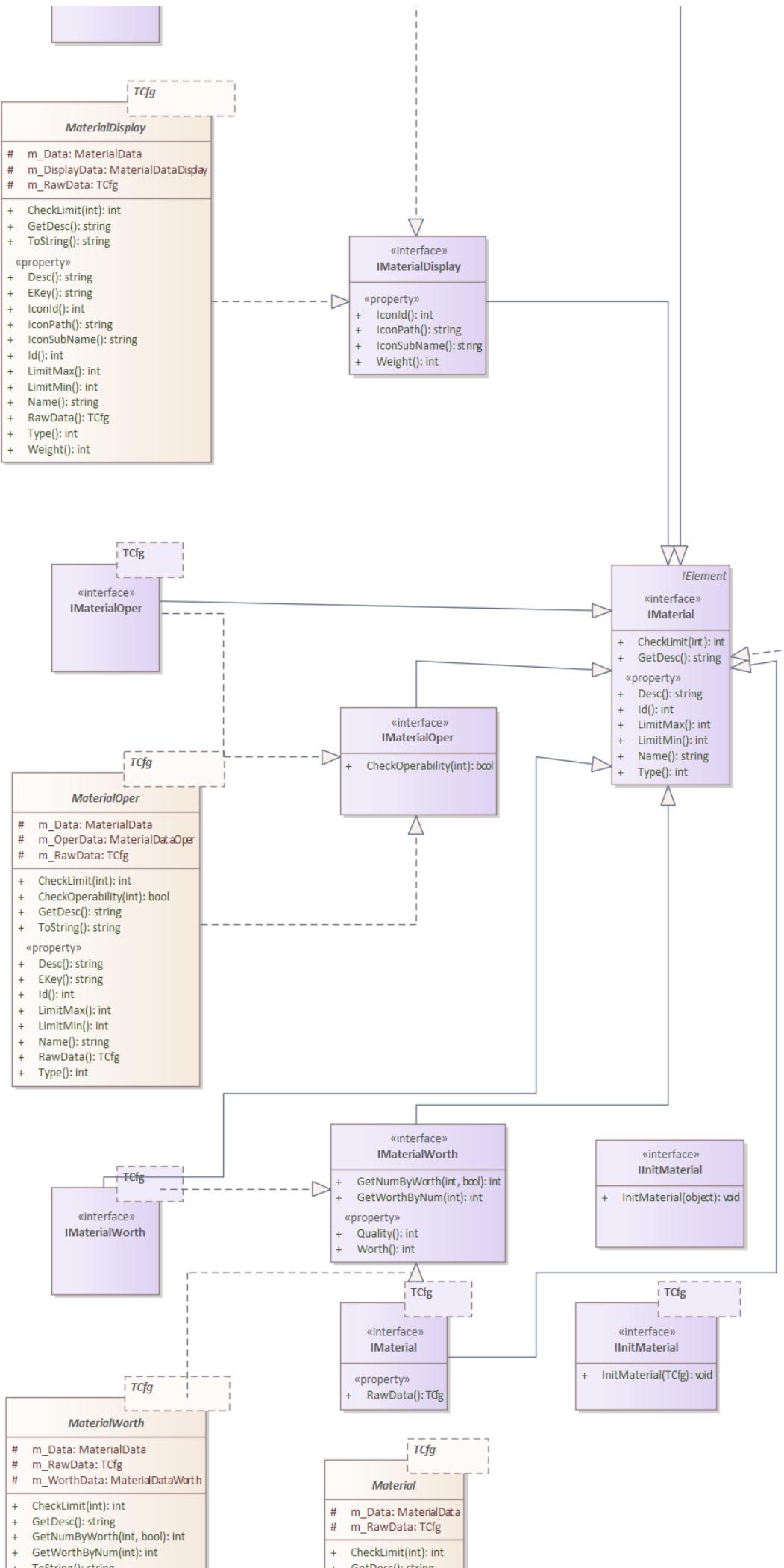
### 8.1.1.3 Common工具类设计

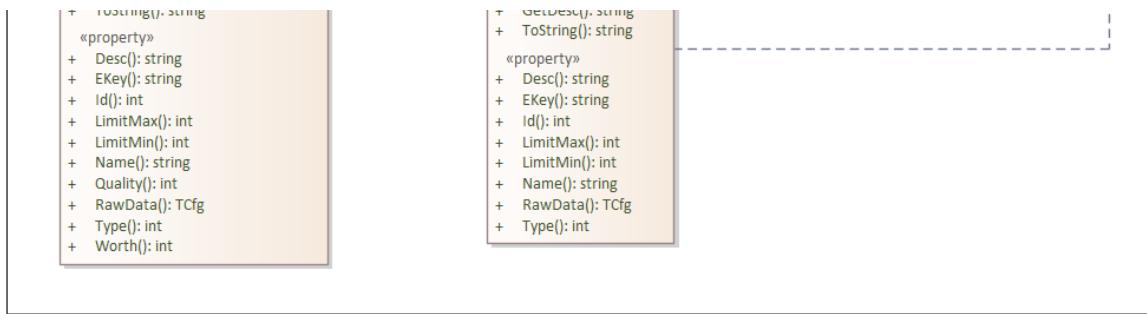
- DataNumUtils 针对DataNum相关的行为逻辑
- DataOffsetUtils 针对DataOffset相关的行为逻辑
- UserNotifyDataUtils 针对UserNotifyData相关的行为逻辑
- DataRandomUtils 针对IDataRandomable接口实现类的随机逻辑

### 8.1.2 Material模块设计说明

- 提供配置数据的常用接口定义和相关实现。
- 使用Common中的接口和数据结构进行组合使用。







### 8.1.2.1 Material接口设计

- IMaterial 继承自 IElement，并定义了配置数据的常用属性与功能。
- IMaterialDisplay 继承自 IMaterial，进一步定义了有显示功能的配置数据的属性与功能。
- IMaterialOper 继承自 IMaterial，进一步定义了有操作行为的配置数据的属性与功能。
- IMaterialWorth 继承自 IMaterial，进一步定义了有价值定义的配置数据的属性与功能。
- IMaterialSet 继承自 IElementSet，并定义了配置数据集合的常用功能。
- IInitMaterial、IInitMaterial<TCfg> 定义配置数据的初始化逻辑接口。

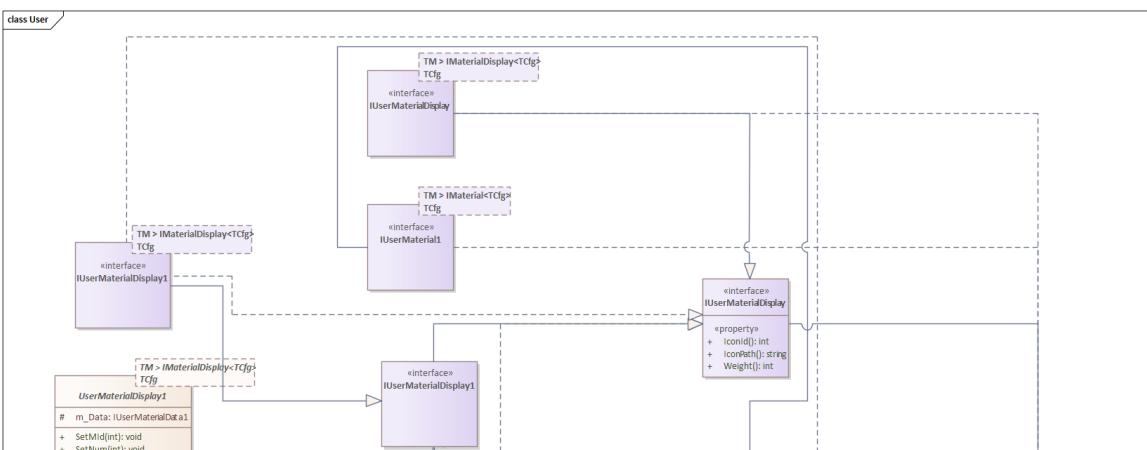
### 8.1.2.2 Material数据结构设计

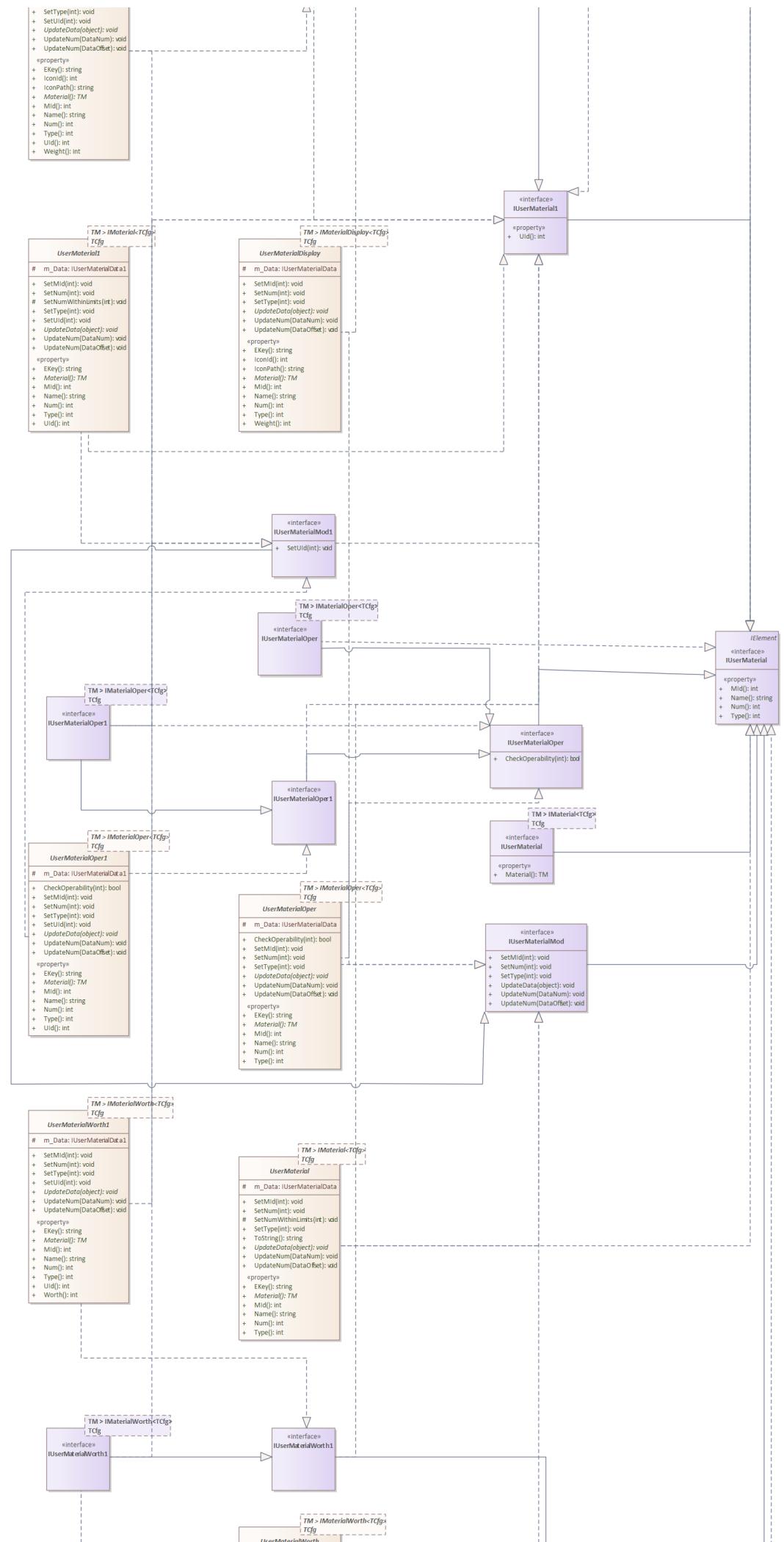
**注意：**以下数据结构的逻辑实现，依赖于Common中的数组结构，使用方式为**组合**。

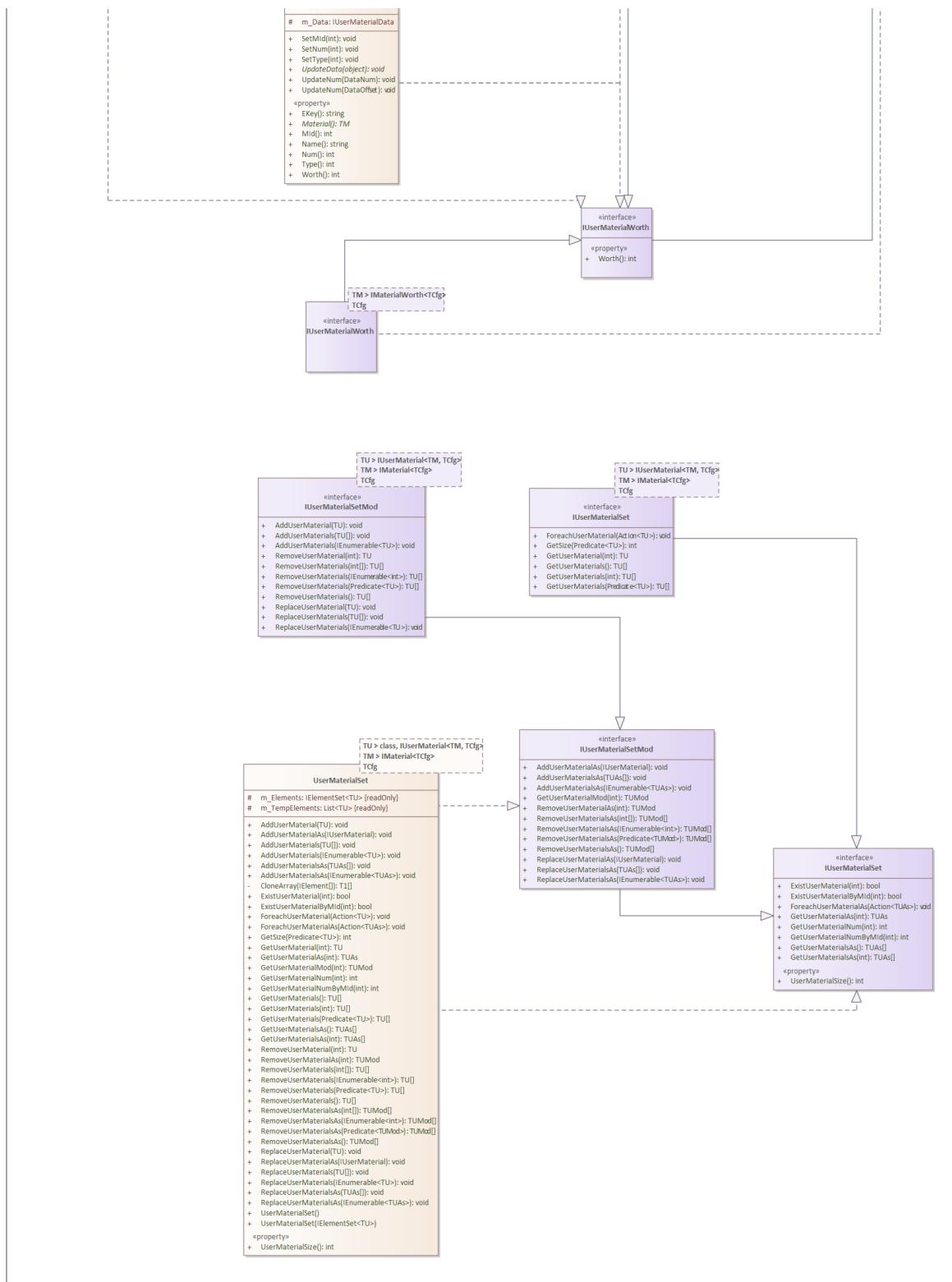
- Material 实现 IMaterial 的抽象类，实现部分功能，子类可重写行为。
- MaterialDisplay 实现 IMaterialDisplay 的抽象类，实现部分功能，子类可重写行为。
- MaterialOper 实现 IMaterialOper 的抽象类，实现部分功能，子类可重写行为。
- MaterialWorth 实现 IMaterialWorth 的抽象类，实现部分功能，子类可重写行为。

### 8.1.3 User模块设计说明

- 提供了普通用户数据功能。
- 提供了带唯一Id的用户数据功能。
- 提供了用于管理用户数据集合功能。







### 8.1.3.1 User接口设计

- IUserMaterial 和 IUserMaterial1
    - IUserMaterial 继承自 IElement，定义了用户数据的属性与功能。
    - IUserMaterial1 继承自 IUserMaterial，增加了唯一Id的属性。
  - IUserMaterialDisplay 和 IUserMaterialDisplay1
    - IUserMaterialDisplay 继承自 IUserMaterial，增加了与显示功能相关的属性与功能。

- IUserMaterialDisplay1 继承自 IUserMaterialDisplay，增加了唯一Id的属性。
- IUserMaterialOper 和 IUserMaterialOper1
  - IUserMaterialOper 继承自 IUserMaterial，增加了与操作行为相关的属性与功能。
  - IUserMaterialOper1 继承自 IUserMaterialDisplay，增加了唯一Id的属性。
- IUserMaterialWorth 和 IUserMaterialWorth1
  - IUserMaterialWorth 继承自 IUserMaterial，增加了与价值相关的属性与功能。
  - IUserMaterialWorth1 继承自 IUserMaterialWorth，增加了唯一Id的属性。
- IUserMaterialMod 和 IUserMaterialMod1
  - IUserMaterialMod 定义用户数据的修改接口
  - IUserMaterialMod1 定义带唯一Id的用户数据的修改接口
- IUserMaterialSet 定义用户数据集合中关于读取、查找相关的接口
- IUserMaterialSetMod 定义用户数据集合中关于修改相关的接口

### 8.1.3.2 User数据结构设计

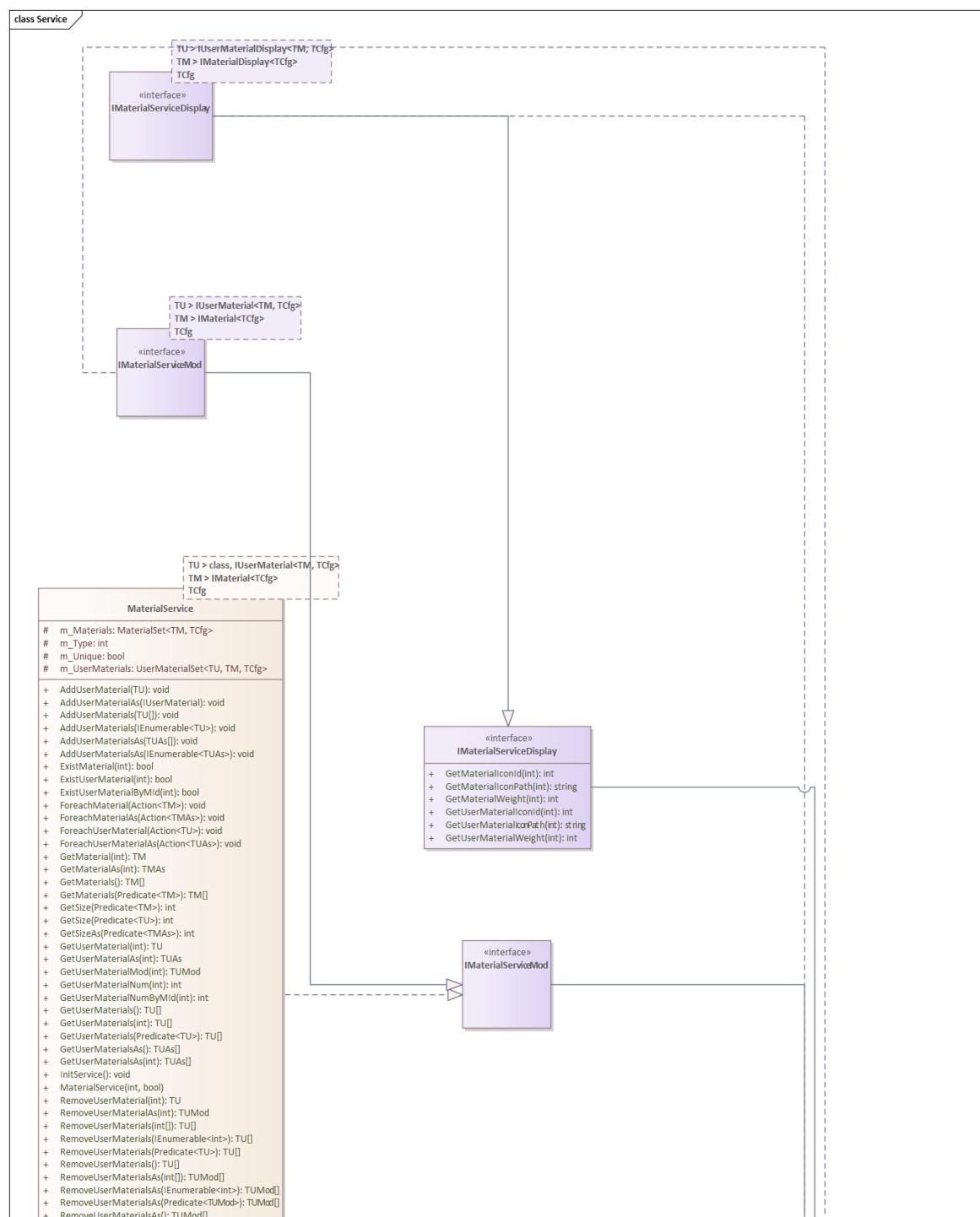
**注意：**以下数据结构的逻辑实现，依赖于Common中的数组结构，使用方式为**组合**。

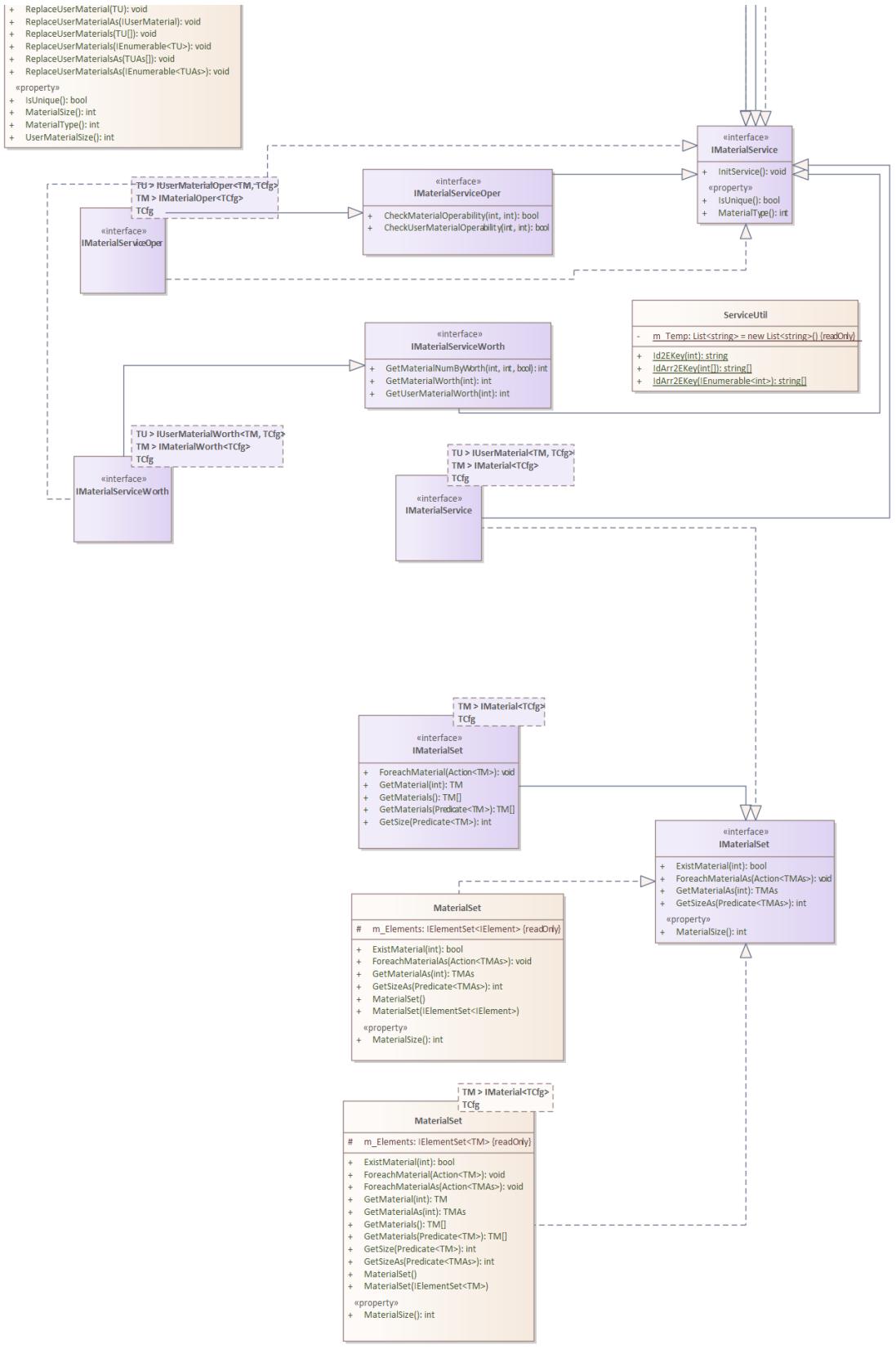
- UserMaterial 和 UserMaterial1
  - UserMaterial 实现 IUserMaterial 的抽象类，实现部分功能，子类可重写行为。
  - UserMaterial1 实现 IUserMaterial1 的抽象类，实现部分功能，子类可重写行为。
- UserMaterialDisplay 和 UserMaterialDisplay1
  - UserMaterialDisplay 实现 IUserMaterialDisplay 的抽象类，实现部分功能，子类可重写行为。
  - UserMaterialDisplay1 实现 IUserMaterialDisplay1 的抽象类，实现部分功能，子类可重写行为。
- UserMaterialOper 和 UserMaterialOper1
  - UserMaterialOper 实现 IUserMaterialOper 的抽象类，实现部分功能，子类可重写行为。
  - UserMaterialOper1 实现 IUserMaterialWorth1 的抽象类，实现部分功能，子类可重写行为。
- UserMaterialWorth 和 UserMaterialWorth1
  - UserMaterialWorth 实现 IUserMaterialWorth1 的抽象类，实现部分功能，子类可重写行为。

- UserMaterialWorth1 实现 IUserMaterialWorth1 的抽象类，实现部分功能，子类可重写行为。
- UserMaterialSet 实现 IUserMaterialSet 的抽象类，实现部分功能，子类可重写行为。

### 8.1.4 Service模块设计说明

- 材料 Service 的设计扩展了[服务框架](#)功能。
- 材料 Service 的分类应该与类型(指配置数据中的类型)一一对应，也就是一种类型创建一个Service。
- 提供了材料 Service 的基本接口定义。
- 提供了可显示材料、可操作材料、有价值材料的基本接口定义。
- 提供了材料service的一个默认实现类，可按需使用。不建议直接修改和继承。





### 8.1.4.1 Service接口设计

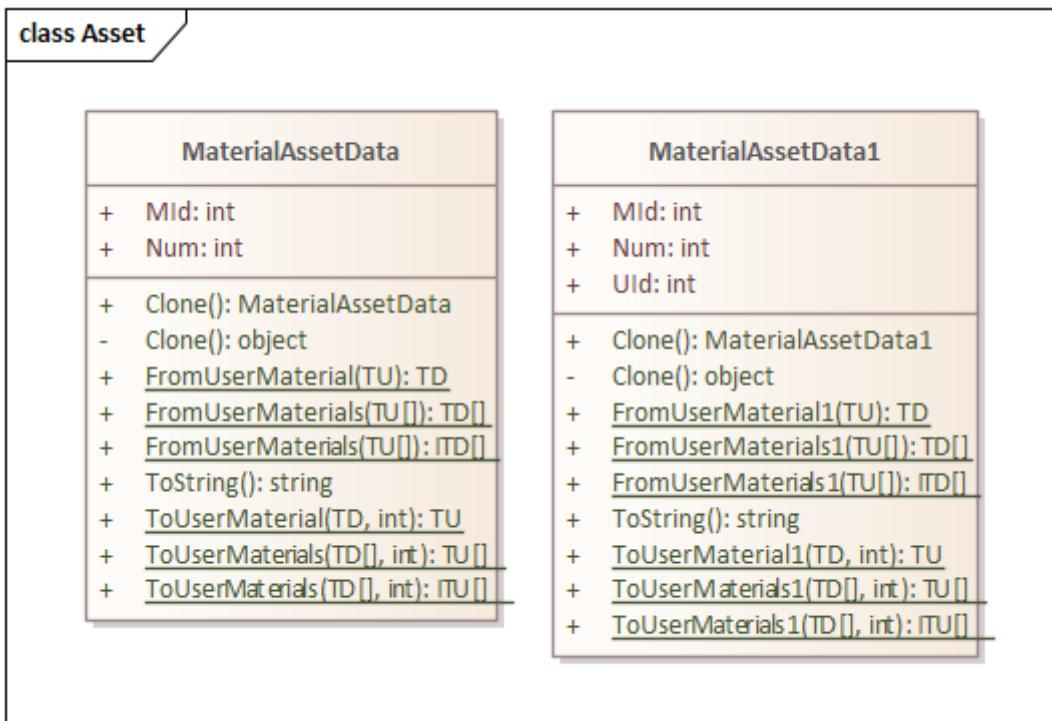
- **IMaterialService** 材料 Service 的基本接口定义。
- **IMaterialServiceDisplay** 可显示材料 Service 的基本接口定义。
- **IMaterialServiceOper** 可操作材料 Service 的基本接口定义。
- **IMaterialServiceWorth** 价值材料 Service 的基本接口定义。
- **IMaterialServiceMod** 材料service的与修改相关的接口定义。

### 8.1.4.2 Service逻辑设计

- MaterialService 一个默认的材料服务实现类，可按需使用。不建议直接修改和继承。  
内部功能由 User、Material、Common 中的功能单元组合而成。
- ServiceUtil 提供一些与 service 相关的基础逻辑函数。

### 8.1.5 Asset模块设计说明

- MaterialAssetData 材料用户数据的序列化结构，用于Unity面板。
- MaterialAssetData1 带唯一Id的材料用户数据的序列化结构，用于Unity面板。



## 8.2 使用

依赖于服务框架，具体使用流程与一般服务一致。

### 8.2.1 材料数据分类

按照业务需求，把材料配置表数据进行分类

### 8.2.2 准备接口

每类材料创建一个服务接口，至少继承 `IInitService`、`IInitDataService`、`IMaterialService`

- `IInitService` 用于初始化材料配置数据
- `IInitDataService` 用于初始化材料用户数据。

**用户数据来源：**

1. 服务器请求

2. 本地缓存文件
3. 其它来源

- IMaterialService 提供一些材料相关的属性与功能。

### 8.2.3 实现接口

实现上一步创建的接口。

### 8.2.4 按照服务框架流程使用

按照[服务框架说明](#)，注册、初始化、调用。

## 8.3 示例

GameDriver/Samples/Service

具体示例服务实现类为ResourceService, 服务名称定义为"ServiceName.Material\_1"

| Name                      | IArgumentService | IAwakableService | IInitService | IInitDataService |
|---------------------------|------------------|------------------|--------------|------------------|
| ServiceNames.ConfigData   | Disable          | Finish           | Finish       | Waiting          |
| ServiceNames.Preload      | Disable          | Finish           | Finish       | Waiting          |
| ServiceNames.Material_1   | Disable          | Disable          | Finish       | Waiting          |
| ServiceNames.DemoService1 | Finish           | Disable          | Finish       | Waiting          |
| ServiceNames.DemoService2 | Finish           | Disable          | Doing        | Waiting          |
| ServiceNames.DemoService3 | Finish           | Disable          | Waiting      | Waiting          |
| ServiceNames.DemoService4 | Finish           | Disable          | Waiting      | Waiting          |