ESave 文档 v1.1.0

2024 年 6 月 10 日

时尚埃斯佩尔

文件版本： 1.1.0

目录

在线文档....4
安装....4
资产商店....4
Github....4
设置示例....4
组件....4
文件结构....4
创建菜单项....5
保存文件设置....5
节省存储空间....7
储蓄....7
1. 获取保存文件....8
2. 保存数据....8
装载....8
1. 获取保存文件....8
2. 加载数据....8
获取数据....8
特殊方法....9
保存和加载示例....9
运行时保存创建....10
重要信息....10

创建保存文件.....11
未加密保存.....11
加密保存.....11
1. 存储 AES 密钥和 IV.....11
2. 创建设置数据.....12
3. 创建保存文件.....12
背景保存文件.....12
无限节省.....13
先决条件.....13
创建用户界面对象.....13
无限保存脚本.....14
步骤 1：创建脚本.....14
步骤 2：其他脚本成员.....15
第 3 步：实例化现有保存文件.....16
步骤 4：递增时间.....16
步骤 5：保存和加载数据.....16
步骤 6：新建保存.....17
了解后台保存和加载.....19
与业务部门合作.....19
获取操作.....19
等待完工.....20
获取帮助.....21

小提示.....21
帮我帮你.....21
错误.....21
错误报告.....21
功能要求.....22
选项.....22

在线文档

ESave 文档可能会偶尔更新。要查找最新版本的 文档，请使用此链接。

安装

资产商店

您可以通过此链接在资产商店中找到最新版本的 ESave。

安装步骤：

1. 从资产库中获取 ESave。
2. 打开 Unity 项目，进入窗口 > 包管理器，切换到我的资产
   从左上角的软件包下拉菜单中，然后在搜索栏中输入 `ESave`。
3. 下载并安装软件包。
4. 完成后，会弹出软件包安装程序窗口。单击安装 `Newtonsoft JSON`。

Github

您可以通过此链接在 GitHub 上找到 ESave 的最新版本。

安装步骤：

1. 从版本页面下载一个 `.unitypackage` 文件。
2. 打开 Unity 项目，双击下载的 Unity 软件包。
3. 单击导入。
4. 完成后，会弹出软件包安装程序窗口。单击安装 `Newtonsoft JSON`。

设置示例

您可以在以下文件中找到可与任何渲染管道配合使用的示例场景

`Assets/StylishEsper/ESave/Examples/Any RP Example`。
如果您想尝试 URP 示例，可以从

`Assets/StylishEsper/ESave/Examples` 中的 `URP_Example.unitypackage`.

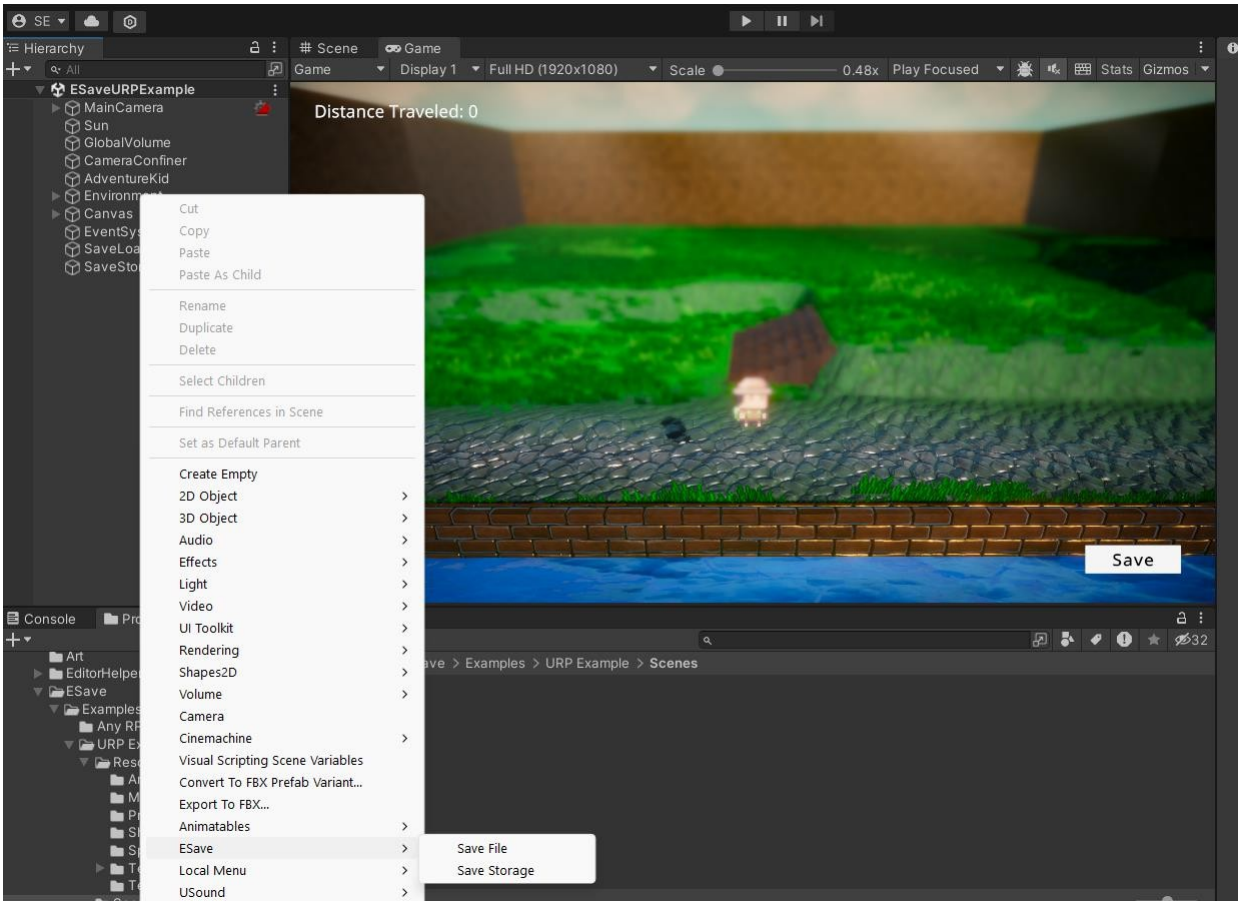
组件

文件结构

保存数据通常存储在不同的文件中，每个保存状态一个文件，玩家设置一个文件。ESave 就是围绕这一理念构建的，因此这是一个需要牢记的重要概念。

建议将保存的数据分开。将玩家的所有数据保存在一个文件中是可能的，也很常见，但这不是必须的。例如，您可以在一个保存文件中

一个用于存储玩家的库存数据，一个用于存储玩家的进度，还有一个用于存储玩家对环境造成的影响。

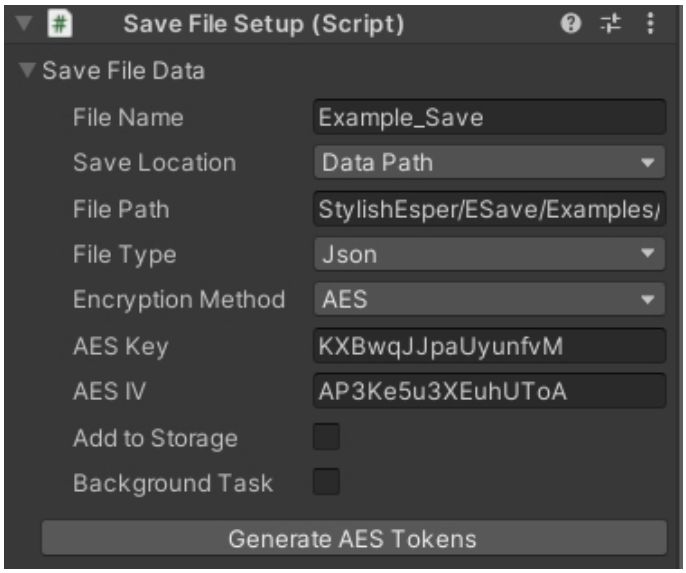


创建菜单项

右击层次结构并导航至 ESave，即可轻松创建 ESave 的保存文件设置和保存存储组件。

保存文件设置

保存文件设置是您将使用的主要组件。该组件的工作原理与它的名称完全相同；它将在用户系统中创建一个保存文件，以便保存和加载数据。



文件名称

将保存在玩家系统中的文件名。

保存位置

有两个常见的位置可以存储播放器文件。

1. **持久数据路径：**一个目录路径，用于存储在运行之间需要保存的数据。
2. **数据路径：**目标设备上游戏数据文件夹的路径。

一般建议使用持久数据路径。数据路径可能无法在所有平台上使用。

文件路径

这是保存位置路径之后的额外路径。例如，在上图中，`URP_Example_Save` 文件将保存在游戏数据文件夹中的 `StylishEsper/ESave/Example` 路径下。

文件类型

文件类型决定了保存的数据格式和文件扩展名。目前只支持 JSON 格式。

加密方法

目前只支持 AES 加密算法。默认选项为不加密。

AES 密钥和 IV

密钥和 IV 用于 AES 算法。您可以单击 "生成 AES 令牌 "按钮，为两者生成随机令牌。

添加到存储

如果选中添加到存储空间，则保存文件将被添加到存储空间。

背景任务

如果选中后台任务，保存和加载操作都将在后台线程中执行，而不是在主线程中。如果保存的文件较大，这将是一个很好的选择。

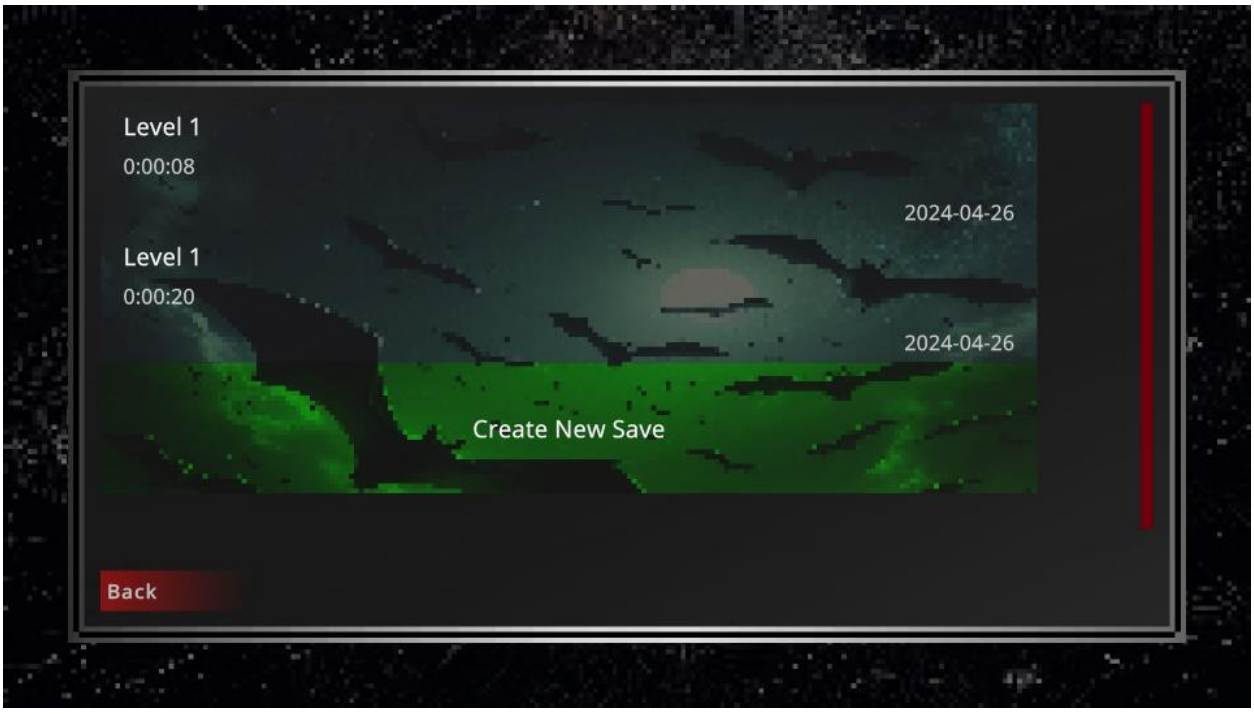
如果您需要在保存和加载完成后执行某些操作，并且 "后台任务 "被选中，请确保先等待操作完成。

节省存储空间

保存存储是一个单例组件，没有检查器属性。保存存储 "会自动存储选中 "添加到存储 "的每个保存文件的路径。它需要一个保存文件设置组件来存储路径。

确保 "保存存储 "的保存文件设置组件的 "添加到存储 "框未被选中。

使用该组件，您可以通过代码根据文件名找到任何保存（已选中 "添加到存储 "值）。



保存存储并不是一个必须的组件，但拥有它可以大大简化访问玩家所有保存内容等任务，尤其是对具有保存列表界面的游戏非常有用。

节约

编写脚本时，请记住使用 `Esper.ESave` 命名空间。

1. 获取保存文件

保存数据的第一步是获取保存文件。这可以通过首先获取保存文件设置组件来完成，该组件将使您可以通过 `SaveFileSetup.GetSaveFile` 访问保存文件。

```
saveFileSetup = GetComponent<SaveFileSetup>();
saveFile = saveFileSetup.GetSaveFile();
```

建议在 "唤醒 "或 "开始 "中使用此代码。

2. 保存数据

保存文件中的每个数据都有一个 ID（字符串）。这样就可以通过 ID 检索特定数据。使用 `SaveFile.AddOrUpdateData` 只需一行即可完成数据保存。

```
saveFile.AddOrUpdateData("DataID", data);
```

第一个参数是数据的 ID，第二个参数是数据本身。任何可序列化的数据类型都可以保存。不过，大多数 Unity 类型无法保存。ESave 可以保 存 常见的 Unity 数据类型，包括（但不限于）Vector2、Vector3、Quaternion、Color 和 Transform。

加载中

1. 获取保存文件

与保存一样，加载数据时也需要引用保存文件。

```
saveFileSetup = GetComponent<SaveFileSetup>();
saveFile = saveFileSetup.GetSaveFile();
```

2. 加载数据

每个加载数据的 ESave 方法只接受一个参数，即数据的 ID。

获取数据

加载数据的主要方法是 `SaveFile.GetData`。该方法可接受任何类型的参数。

```
// 其中 T 是数据类型
T data = saveFile.GetData<T>("DataID");
```

您还可以检索同一类型的数据列表：

```
// 其中 T 是数据类型
List<T> dataList = saveFile.GetData<T>("DataID", "DataID2",
"DataID3");
```

特殊方法

ESave 具有检索某些 Unity 类型数据的特殊方法。

```
Vector2 v2 = saveFile.GetVector2("DataID");
```

向量2

向量3

```
Vector3 v3 = saveFile.GetVector3("DataID");
```

```
四元数 q = saveFile.GetQuaternion("DataID");
```

四元数

颜色

```
颜色 c = saveFile.GetColor("DataID");
```

```
// 返回一个可保存的变换
SavableTransform st = saveFile.GetVector2("DataID");

// 使用 Transform.CopyTransformValues 设置来自一个
// 将 SavableTransform 转换为 Transform
transform.CopyTransformValues(st);
```

变革

保存和加载示例

该脚本会在退出游戏时保存玩家的变换值，并在启动时加载变换值。

使用 ESave 保存变换时，只保存位置、旋转和缩放属性。

```
使用 UnityEngine;

public class SaveLoadExample : MonoBehaviour
{
    // 常量数据 ID
    private const string playerTransformDataKey = "PlayerTransform";

    [序列化字段]
    私有 Transform playerTransform;

    private SaveFileSetup saveFileSetup;
    private SaveFile saveFile;

    私人 void Start()
    {
        // Get save file component attached to this object
        saveFileSetup = GetComponent<SaveFileSetup>();
        saveFile = saveFileSetup.GetSaveFile();
```

```
        // 载入游戏
        LoadGame();
    }

    /// <summary>
    /// 载入游戏。
    /// </summary>
    公 void LoadGame()
    {
        // 检查文件中是否存在数据
        如 (saveFile.HasData(playerPositionDataKey))
        果
无法保存的数据// 从特殊方法中获取 Vector3，因为 Vector3 是
        {
            var savableTransform =
saveFile.GetTransform(playerPositionDataKey);
            playerTransform.CopyTransformValues(savableTransform);
        }

        Debug.Log("Loaded game.");
    }

    /// <summary>
    /// 挽救游戏。
    /// </summary>
    公共 void SaveGame()
    {
        saveFile.AddOrUpdateData(playerPositionDataKey,
playerTransform);
        saveFile.Save();

        Debug.Log("Saved game.");
    }

    私人 void OnApplicationQuit()
    {
        SaveGame();
    }
}
```

运行时保存创建

重要信息

在开始之前，需要注意的是，`new SaveFile()` 并不总是在用户系统中创建新的保存文件。只有在指定路径下没有指定文件名的保存文件时，它才会创建新的保存文件。

例如，如果我们为 2 个 SaveFile 实例使用相同的 SaveFileSetupData，用户系统中将只创建 1 个文件。2 个保存文件实例只是在编辑同一个文件。

```
SaveFileSetupData saveFileSetupData = new SaveFileSetupData()
{
   fileName = "保存文件"、
   saveLocation = SaveLocation.PersistentDataPath,
   filePath = "Example/Path"、
   encryptionMethod = EncryptionMethod.None,
   addToStorage = true
};
// 这两个文件将编辑同一个保存文件 SaveFile saveFile1 = new
SaveFile(saveFileSetupData); SaveFile saveFile2 = new
SaveFile(saveFileSetupData);
```

此处仅供参考，**不**建议这样做。

创建保存文件

我们知道，您可以使用 "保存文件设置 "组件创建保存文件。另外，您也可以通过代码来创建保存文件。

未加密保存

创建保存文件需要保存文件设置数据。下面的代码创建了保存文件设置数据。

```
SaveFileSetupData saveFileSetupData = new SaveFileSetupData()
{
   fileName = "保存文件"、
   saveLocation = SaveLocation.PersistentDataPath、
   filePath = "Example/Path", // 持久数据路径之后的路径（可留空）
   fileType = FileType.Json,
   encryptionMethod =
   EncryptionMethod.None, addToStorage =
   true
};
SaveFile saveFile = new SaveFile(saveFileSetupData);
```

不过，这段代码不会创建加密保存文件。

加密保存

当时唯一支持的加密方法是 AES。要创建 AES 加密保存文件，我们需要 AES 密钥和 IV 。它们用于 AES 算法。AES 密钥和 IV 都是一串随机字母数字字符。建议长度至少为 16 个字符。

1. 存储 AES 密钥和 IV

保存文件的密钥和 IV **不得**更改，因为加密和解密时都 需要使用相同的密钥和 IV。因此，我们应该创建每次都 要使用的常量。

```
private const string aesKey = "randomkey1234567";
private const string aesIV = "randomiv12345678";
```

2. 创建设置数据

这将与前一个类似，只是我们将加密方法改为 AES，并提供密钥和 IV。

```
SaveFileSetupData saveFileSetupData = new SaveFileSetupData()
{
    fileName = "保存文件"、
    saveLocation = SaveLocation.PersistentDataPath,
    filePath = "Example/Path"、
    fileType = FileType.Json、
    encryptionMethod = EncryptionMethod.AES,
    aesKey = aesKey、
    aesIV = aesIV、
    addToStorage = true
};
```

3. 创建保存文件

使用下面的代码后，您就成功创建了一个加密保存文件。

```
SaveFile saveFile = new SaveFile(saveFileSetupData);
```

背景保存文件

要创建一个能在后台保存和加载数据的保存文件，请设置 `backgroundTask`

值为 true。

```
SaveFileSetupData saveFileSetupData = new SaveFileSetupData()
{
    fileName = "保存文件"、
    saveLocation = SaveLocation.PersistentDataPath,
    filePath = "Example/Path"、
    fileType = FileType.Json、
    encryptionMethod = EncryptionMethod.None,
    addToStorage = true、
    backgroundTask = true
};
SaveFile saveFile = new SaveFile(saveFileSetupData);
```

您可以在运行时随时设置该值，保存文件将相应地使用后台线程或主线程。

```
saveFile.backgroundTask = false;
```

无限节省

从 v1.0.1 开始，在

`Assets/StylishEsper/ESave/Examples/Any RP Examples`。

本教程将参考 ESaveInfiniteSavesExample 场景中的用户界面设置。

先决条件

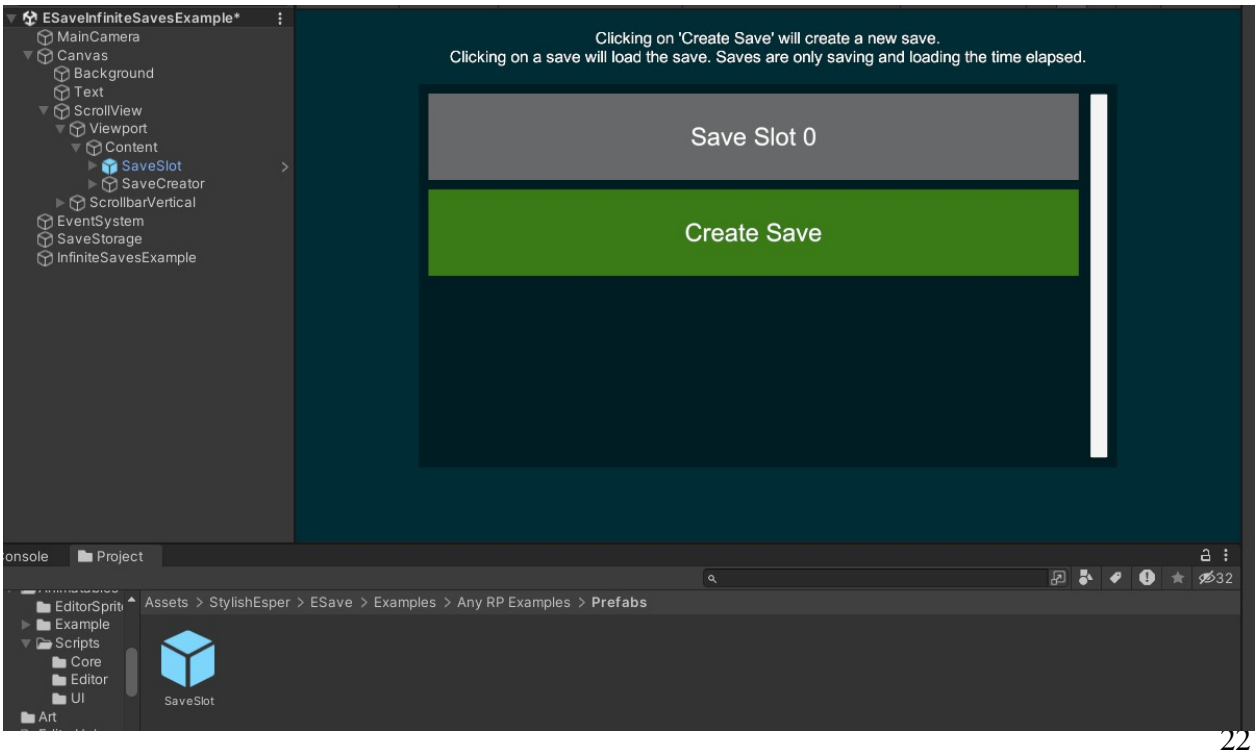
您需要某种可支持无限保存的用户界面设置。一般来说，这种情况下需要一个滚动视图。滚动视图中将填充保存槽按钮和一个保存槽创建按钮。

您需要一种方法来切换保存窗口的模式。在本例中，我们将使用切换器。

创建用户界面对象

我们将首先创建一个 UI 按钮，它既可以加载数据，也可以覆盖数据。这将是一个预制件。您可以通过将对象拖入项目窗口来制作预制件。

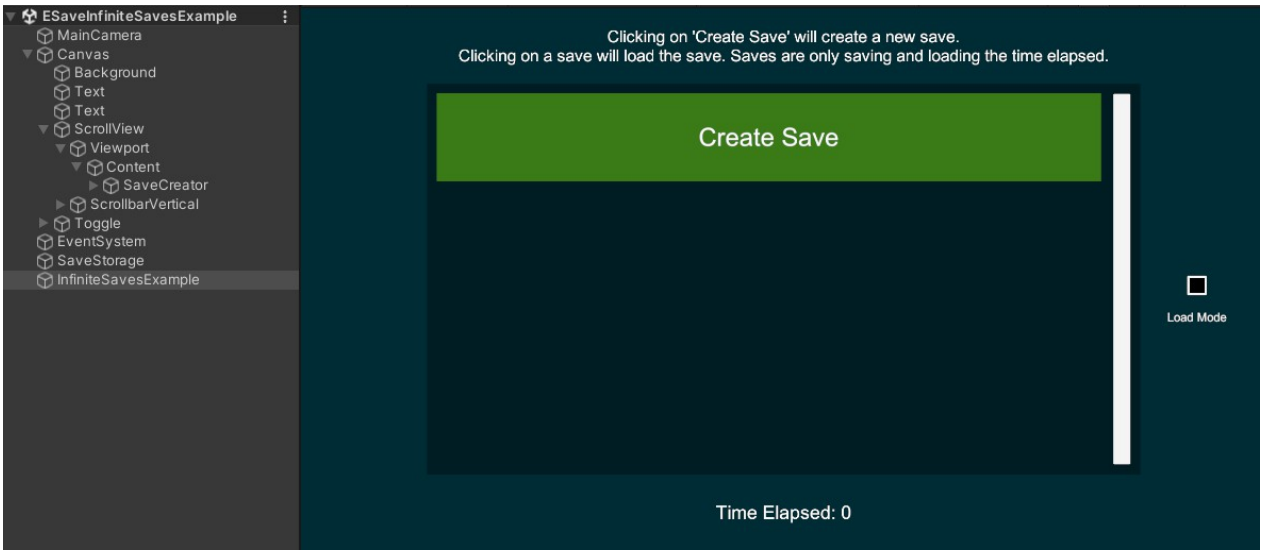
然后，创建一个可以创建保存槽的按钮。这并不需要是一个预制件。这两个按钮都将是滚动视图内容的子按钮。



从层次结构中删除保存槽按钮，因为它将在运行时实例化。

在本例中，我们将使用一个切换按钮来改变保存窗口的模式，该窗口位于滚动视图的右侧。

我们要保存的数据只是经过的时间。因此，滚动视图下方有一个文本对象，它将显示当前经过的时间。



无限保存脚本

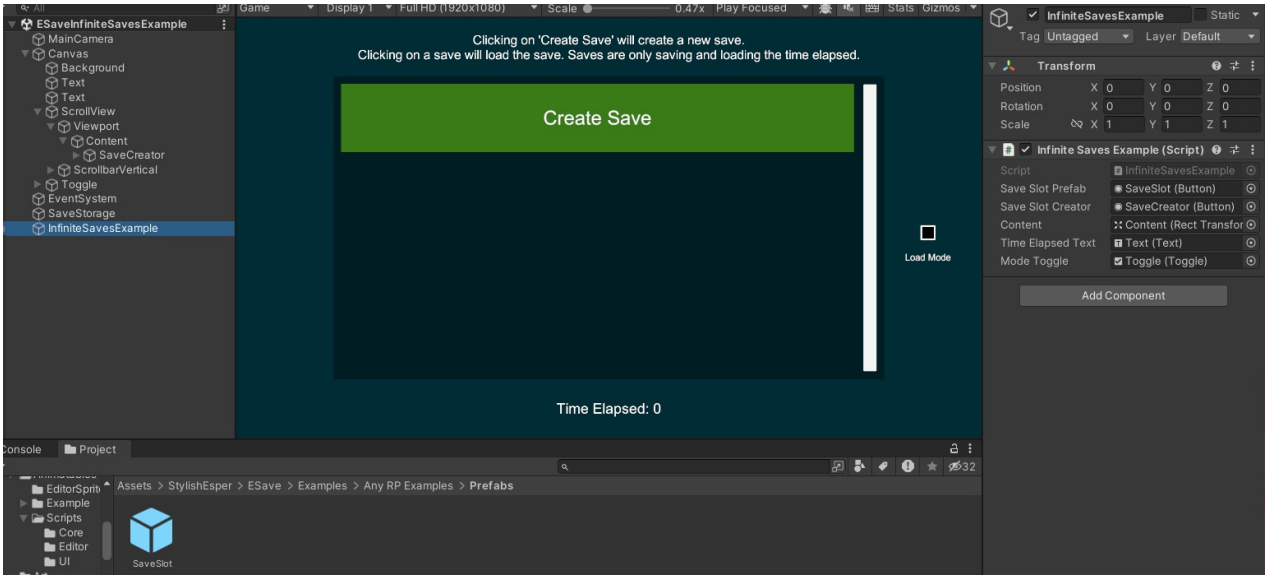
步骤 1：创建脚本

创建一个或多个脚本来处理所有保存菜单功能。在本例中，我们将创建一个名为 `InfiniteSavesExample.cs` 的脚本。

在脚本开始时，添加可在检查器中设置的成员变量。

```
public class InfiniteSavesExample : MonoBehaviour
{
    [序列化字段]
    私用按钮 saveSlotPrefab;

    [序列化字段]
    私用按钮 saveSlotCreator;

    [序列化字段]
    私下转换内容;

    [序列化字段]
    私人文本 timeElapsedText;

    [序列化字段]
    私用 Toggle modeToggle;
}
```

记得在场景中添加一个 GameObject，并将此脚本添加到该 对象中。至 此，我们应该已经具备了填充检查器属性所需的一切。



属性

保存插槽预制件：保存插槽按钮预制件。

保存槽创建器：创建新保存的按钮。

内容：滚动视图的内容对象。

所用时间文本：显示所用时间的文本对象。

模式切换：在加载模式和保存模式之间切换。

步骤 2：其他脚本成员

我们将创建其他在检查器中不可见的成员。

```
public class InfiniteSavesExample : MonoBehaviour
{
        private const string timeElapsedKey = "TimeElapsed";

        [序列化字段]
        私用按钮 saveSlotPrefab;

        [序列化字段]
        私用按钮 saveSlotCreator;

        [序列化字段]
        私下转换内容;

        [序列化字段]
        私人文本 timeElapsedText;

        [序列化字段]
        私用 Toggle modeToggle;

        private List<Button> slots = new();
```

```
        私人 float timeElapsed;

        private bool loadMode { get => modeToggle.isOn; }
}
```

timeElapsedKey：时间流逝的键（或 ID），用于保存。 **slots：**按钮列表

，用于存储所有实例化保存槽。 **timeElapsed：时间**流逝的浮点数。

loadMode：切换开关的 isOn 值。

如果加载模式为真，我们将使保存槽加载一个保存。如果为假（保存模式），我们将使保存槽覆盖一个保存。

第 3 步：实例化现有保存内容

进入游戏模式后，我们需要首先加载玩家在之前的会话中建立的任何现有保存。

我们可以使用 "开始 "方法来做到这一点。

```
私人 void Start()
{
    // 为现有保存设置实例化插槽
    foreach（SaveStorage.instance.saves.Values 中的 var save）
    {
        创建新保存插槽（保存）；
    }
}
```

步骤 4：递增时间

在 "更新 "方法中，我们将递增所经过的时间。这将是本示例中唯一需 要 保存和加载的数据。

```
私人 void Update()
{
    // 每帧时间递增 timeElapsed +=
    Time.deltaTime;
    timeElapsedText.text = $"Time Elapsed: {timeElapsed}";
}
```

步骤 5：保存和加载数据

我们将创建 3 个用于保存和加载的方法。第一个方法将从保存文件中加载数据。

```
/// <summary>
```

```
/// 加载保存。
/// </summary>
/// <param name="saveFile">保存文件。 </param> public
void LoadSave(SaveFile saveFile)
{
    timeElapsed = saveFile.GetData<float>(timeElapsedKey);
}
```

第二个选项将保存（或覆盖）保存文件中的数据。

```
/// <summary>
/// 重写保存。
/// </summary>
/// <param name="saveFile">保存文件。 </param> public
void OverwriteSave(SaveFile saveFile)
{
    // 保存经过的时间 saveFile.AddOrUpdateData(timeElapsedKey,
    timeElapsed); saveFile.Save();
}
```

第三个将用于保存槽，并根据模式加载或覆盖数据。

```
/// <summary>
/// 根据活动模式加载或覆盖保存内容。
/// </summary>
/// <param name="saveFile">保存文件。 </param> public
void LoadOrOverwriteSave(SaveFile saveFile)
{
    如果 (loadMode)
    {
        加载保存（保存文件）；
    }
    不然
    {
        OverwriteSave(saveFile);
    }
}
```

步骤 6：创建新的保存

我们有需要保存文件的方法，但尚未创建保存文件。按下 "创建新的保存 "按钮时，应创建保存文件，同时创建保存槽。

因此，让我们为此创建一些方法。第一个方法将创建一个新的保存文件。

```
/// <summary>
/// 创建新的保存。
/// </summary>
```

```
公共 void CreateNewSave()
{
    // 创建保存文件数据 SaveFileSetupData
    saveFileSetupData = new()
    {
        文件名 =
$"InfiniteExampleSave{SaveStorage.instance.saveCount}",
        saveLocation = SaveLocation.DataPath、
        filePath = "StylishEsper/ESave/Examples/Any RP Examples",
        fileType = FileType.Json、
        encryptionMethod = EncryptionMethod.None,
        addToStorage = true
    };

    SaveFile saveFile = new SaveFile(saveFileSetupData)；

    // 保存经过的时间
    // 技术上讲，由于这是一个空的保存文件，因此在这个阶段没有任何内容被覆盖
    OverwriteSave(saveFile)；

    // 为该数据创建保存槽
    CreateNewSaveSlot(saveFile)；
}
```

最后调用 CreateNewSaveSlot。该槽尚未创建。CreateNewSaveSlot 方法将实例化一个新的保存槽，编辑保存槽的文本，并为保存槽提供一个点击事件，该事件将调用 LoadOrOverwriteSave。

```
/// <summary>
/// 为保存文件创建保存槽。
/// </summary>
/// <param name="saveFile">保存文件。 </param> public
void CreateNewSaveSlot(SaveFile saveFile)
{
    // 安装保存槽
    var slot = Instantiate(saveSlotPrefab, content);
    var slotText = slot.transform.GetChild(0).GetComponent<Text>();
    slotText.text = $"保存插槽 {slots.Count}";

    // 将保存创建器移动到底部 saveSlotCreator.transform.SetAsLastSibling();

    // 为加载添加点击事件
    slot.onClick.AddListener(() => LoadOrOverwriteSave(saveFile));

    slots.Add(slot);
}
```

我们仍然需要保存槽创建按钮有一个创建新保存的单击事件。这可以通过更新我们的 "开始 "方法来实现。

```
私人 void Start()
{
    // 为现有保存设置实例化插槽
    foreach（SaveStorage.instance.saves.Values 中的 var save）
    {
        创建新保存插槽（保存）；
    }

    // 保存插槽创建器的单击事件
    saveSlotCreator.onClick.AddListener(CreateNewSave)；
}
```

完成！现在可以在游戏模式下进行测试了。

了解后台保存和加载

在 后台保存和加载数据不需要做任何额外的操作。只要将 `backgroundTask` 设为 true，就会发生这种情况。不过，在处理保存文件时，您可能需要稍微改变一下编码方式。

与业务部门合作

通常情况下，如果您希望在保存或加载完成后发生一些事情，您只需遵守操作顺序，在保存或加载代码后执行您的代码即可。

如果您从 "保存文件设置 "组件中启用了 "后台任务"，或通过代码将 `backgroundTask` 设置为 true（本质上是相同的），则此规则不适用。

相反，您必须等待保存或加载操作完成。这可以通过 `SaveFileOperation` 类轻松实现。

获取操作

`SaveFile.Load` 和 `SaveFile.Save` 返回一个 SaveFileOperation 对象。这可用于确定保存或加载何时完成。无论是保存还是加载，使用操作对象的方式都没有区别。

重要的是，保存必须在加载之前完成，而加载必须在保存之前完成。这就是保存文件一次只能运行一个操作的原因。

从负载中获取操作

```
SaveFileOperation operation = saveFile.Load();
```

从 Save 获取操作

```
SaveFileOperation operation = saveFile.Save();
```

调用保存或加载后获取操作

保存文件会在内存中存储最近的保存文件操作。如果最近调用过 `SaveFile.Load` 或 `SaveFile.Save`，可以使用 `SaveFile.operation` 来获取操作。如果加载或保存操作未被调用，该操作将为空。

```
SaveFileOperation operation = saveFile.operation;
```

等待完工

有两种方法可以检查保存文件操作是否完成。

1. 检查国家

`SaveFileOperation` 有一个名为 `OperationState` 的枚举。该枚举将随着操作状态的改变而更新。

共有 5 个州：

1. **无**：操作尚未开始。
2. **进行中**：当前正在执行操作（保存或加载）、
3. **已完成**：操作已成功完成。
4. **取消**：取消操作。只能使用 `SaveFileOperation.Cancel`.
5. **失败**：操作过程中遇到错误并已中止。错误信息应显示在控制台中。

您可以用它来检查当前操作所处的状态。

```
如果 (operation.state == SaveFileOperation.OperationState.Completed)
{
    // 做点什么...
}
```

如果您想在保存或加载完成后立即执行某些操作，则**不建议**使用此方法，因为您必须在每一帧都使用此检查。

2. 操作结束事件

`SaveFileOperation.onOperationEnded` 将在操作结束时调用（当操作状态为完成、取消或失败时）。下面是一个示例，说明如何使用它在保存完成后执行一些代码：

仅对将在后台保存和加载的保存文件使用 `onOperationEnded` 事件。

```
私人 void Start()
{
    // 一些代码

    // 从保存文件中获取加载操作
```

```
    var operation = saveFile.Load();

    // Add on-ended event
    operation.onOperationEnded.AddListener(LoadGame);
}

私人 void LoadGame()
{
    // 您的加载代码在这里...
}
```

如果要确保操作成功完成，可以使用类似这样的事件：

```
// Add on-ended event
operation.onOperationEnded.AddListener(() => 添加
结束事件操作。
{
    如果 (operation.state == SaveFileOperation.OperationState.Completed)
    {
        LoadGame();
    }
    不然
    {
        // 做点别的...
    }
});
```

获取帮助

需要帮助?

小提示...

我是一名独立开发者，试图通过创建有用的 Unity 工具来简化他人（和我自己）的游戏开发。我不是一个大团队的成员，也不是一个小团队的成员，只有我自己。如果我收到您的信息，我会尽快回复（给我一两天时间即可）。

帮我帮你

错误

如果您遇到错误，请在留言中列出错误，并解释导致错误的步骤。

错误报告

如果您要报告错误，请在报告中回答这些问题：

1. 你想做什么？
2. 你预计会发生什么？
3. 究竟发生了什么？

功能请求

我一直在不断改进我的产品。如果您希望添加特定功能，请随 时 告 诉 我。请在留言中说明该功能对您的使用案例有何帮助。

选项

您可以通过以上任何一种方式与我联系。

1. Discord 服务器
2. 网站联系表
3. 发送电子邮件至 developer@stylishesper.com
4. 在 X（推特）上给我留言