# 第三章 游戏开发工具制作与使用

## 3.1 插件与类库

### 3.1.1DOTween插件

DOTween是一种快速、高效、类型安全的面向对象的动画引擎，基于Unity而制作的。它免费且开源，有着大量先进的功能，非常适用于C#语言的使用者。与其他Unity补间动画的插件相比，DOTween是当中效率最高，使用最方便，最流行的插件。

### 3.1.2 ExcelDataReader库

ExcelDataReader本身是一个用C#语言写的高效且轻量的操作Excel文件的类库。

使用这个类库的目的是为了读取Excel中的数据，用于作品中游戏数据的配置。游戏数据的配置一般是由游戏策划做的，有很多形式，比如JSON、XML。尽管JSON和XML这种类型用于数据配置已经足够清晰了，但是相对于Excel还是棋差一著。

在本次课题中会使用Excel进行游戏数据配置，但是Excel文件本身并不用于游戏过程中的数据读取，只是一个数据存储的中间过程。我将用ExcelDataReader读取Excel的xlsx格式文件，获取到的数据再转换为Unity自己的数据保存类ScriptableObject。至于为什么不用XML或者JSON，是因为ScriptableObject是Unity自带的数据保存类，在Unity的编辑器中可视化非常好。

## 3.2 编辑器扩展

在游戏制作开发过程中，为了工作效率的提升就需要一些工具进行辅助。像上文提及的DOTween插件，这是开发者中已经广泛使用的工具，开发人员不必再次开发，直接添加使用即可。然而每个游戏的需求都各不相同，开发者还是需要自己制作一些工具配合自己的工作，这些工具很多时候仅仅适用于一个游戏项目。

### 3.2.1地图编辑

地图是塔防游戏的灵魂，每个关卡的地图数据如何设置和保存非常重要，因此必须专门设计一个地图编辑工具进行处理。这个地图编辑工具的目的是保存每一个关卡的建塔点数据。接下来将梳理地图编辑工具的开发流程。

目标：地图实例化所有的格子，我用鼠标选择哪些格子成为建塔点，记录下来。

1.数据类的处理：

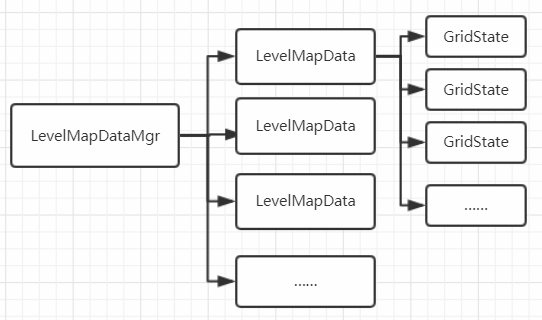
（1）每一个关卡的地图都将被平均划分为9\*12个格子，每个格子需要一个状态，记录格子自身是否是建塔点，是否已经有塔在上面了，就是GridState结构体。

（2）由于加载不同的关卡会布置不同的建塔点，我们需要一个关卡类LevelMapData，记录这个关卡所有的地图数据，也就是对应关卡所有的GridState。LevelMapData还需要一个编号，以便我们加载不同关卡时找到对应的数据。

（3）所有的关卡数据需要一个地方存放在一起，就是LevelMapDataMgr类，继承自ScriptableObject。

|  |  |  |
| --- | --- | --- |
| 数据类 | 成员 | 备注 |
| GridState(结构体)  每个格子的状态 | int id | 所在格子的编号 |
| bool isTowerPoint | 是否是建塔点 |
| bool hasTower | 是否有塔 |
| int towerID | 持有什么塔 |
| LevelMapData  每个关卡的地图数据 | int levelID | 关卡编号 |
| List<GridState> gridStateList | 该关卡所有的建塔点 |
| LevelMapDataMgr  记录所有关卡的地图数据，继承Scriptable Object | List<LevelMapData>levelMapDataList | 记录所有关卡地图数据 |

上述三个类由上到下是被包含的关系。



2.功能类

1. GridPoint

继承Unity的MonoBehaviour类，挂载到格子物体上。捕捉鼠标点击事件，实现通过鼠标点击选择哪些格子成为建塔点的功能。持有数据成员GridState，在格子被选为建塔点后就修改GridState数据。

1. MapMaker

继承Unity的MonoBehaviour类，负责加载所有的格子，在游戏时加载对应的地图数据。

1. MapTool

继承Unity的Editor类，也就是编辑器扩展部分，对MapMaker进行扩展，赋予MapMaker组件几个功能按钮。主要有保存当前关卡数据、读取指定关卡数据、搜集当前关卡数据、重置操作。

### 3.2.2游戏数据

除了上文中的地图数据，本次课题中的大部分游戏数据都是在Excel中配置的，每一种数据都需要转化为Unity的ScriptableObject数据保存格式。这部分的实现也比较简单，就是对应一个数据写一个数据处理类。比如敌人的数据EnemyInfo，跟上文的LevelMapData一样，也需要一个EnemyInfoMgr来存储每一条敌人数据EnemyInfo。EnemyInfoMgr所有的数据最后就是转化存储到Unity的一个asset文件。因此，这部分编辑器扩展的工作其实就是生成这一个asset文件。流程如下：

1. 读取每一行Excel数据
2. 每一行Excel数据生成一个EnemyInfo对象，将行数据对应EnemyInfo的成员变量
3. 将上一步的EnemyInfo对象添加到EnemyInfoMgr的List<EnemyInfo>中。
4. 将所有的EnemyInfo对象添加完毕。
5. 写一个EnemyInfoEditor类，继承Editor，使得能够在Unity菜单中添加一个子项，当点击这个菜单项时，就实现上面的流程，并且将EnemyInfoMgr生成对应的asset文件。

在本次课题项目中，以下的数据类都是这么完成的：

|  |  |
| --- | --- |
| 数据类 | 作用 |
| AchievementInfo | 成就系统的数据 |
| EnemyInfo | 敌人的数据 |
| TowerInfo | 防御塔的数据 |
| ItemInfo | 道具的数据 |
| RoundData | 回合的数据 |
| LevelInfo | 关卡的数据 |
| UIPanelInfo | UI面板的数据 |

### 3.2.3 UI面板代码生成

在第二章中提及的每个UI面板，为了便于管理，面板的变量赋值都是通过代码赋值的，而不是采用Unity拖拽的方式来赋值。因此对应面板的脚本除了需要敲写对应的变量，还要写这些变量的赋值代码。这些代码形式类似，又非常繁琐。例如GamePlayPanel中的UI变量就多达二十个。而本次课题的项目中涉及的UI面板就有十余个，每次重复这样的工作非常浪费时间。看似本次课题的项目UI面板并不多，但是如果遇到的项目非常大，类似《王者荣耀》这类游戏，需要制作的面板可以达到上百个，效率就会被严重拖低。因此我希望能够有一个代码生成器，帮助我生成对应的UI面板代码，生成变量和对变量进行赋值的代码。

代码思路流程如下：

1. 获取选中的UI面板中所有UI物体的名称；
2. 根据命名的规范，提取出需要变量的物体；
3. 编写文件字符，组合整个类的字符结构，根据代码格式添加对应的变量字符；
4. 将字符写入到文件中，生成为.cs文件。

关于命名规范，我的约定是，只有UI物体的名称为Txt\_、Img\_、Btn\_、Trans\_、Raw\_开头的物体才会创造它们的变量。最终效果要做到在Unity的hierarchy界面中选中需要生成代码的面板，点击右键菜单中的GameObject->UITool->CreateScripts就可以生成对应的面板代码。

首先建立一个UIScriptsGenerator类，继承自Editor，这样才能在菜单中创建菜单项。建立一个菜单项点击方法名为CreateScript，方法中先要获取到当前鼠标选中的物体，UnityEditor中提供了Selection.gameObjects可以实现这个要求。这是一个数组，我的需求不考虑多选物体生成代码的情况，因此获得数组第一个对象即可。

获取到选中物体后，需要查找到这个物体下所有的子物体，使用GameObject.GetComponenetsInChildren<Transform>()可以获得一个包含所有子物体的Transform数组。紧接着获取子物体的名称，根据name.Split('\_')可以获得物体名称的前缀。用一个List<string>存下所有的物体名称，同时也要根据前缀存下它们的类型枚举到一个List<UIItemType>中。

public enum UIItemType

{

Txt,

Img,

Btn,

Raw,

Trans,

}

物体的数据收集完就看可以开始拼代码的字符了。

using UnityEngine;

using UnityEngine.UI;

public class 面板名称：BaseUI

{

遍历List<string>、List<UIItemType> 添加对应的变量

public void Init()

{

遍历List<string>、List<UIItemType> 添加对应的变量赋值

}

……

其他常用的方法

}

## 3.3 调试信息输出工具

游戏在制作过程中需要进行调试测试。在编辑器模式下进行测试，遇到问题的时候通过调试信息基本能够比较直观的确定问题所在。但是游戏的测试不仅仅只有在编辑器模式的时候测试，更重要的是真机测试，也就是将游戏打包出来进行测试。然而在真机测试的时候，游戏一旦出现BUG往往只能凭借经验猜测问题可能存在在什么地方。而修复BUG的时候又得回到编辑器模式进行处理，有些BUG仅仅出现在真机测试的情况，编辑器模式难以模拟复现，也就导致查找问题所在变得非常困难也很消耗时间精力。如何在真机测试的时候准确定位到代码的问题所在就显得尤为重要。

我们在查找BUG的时候，最常见的两种方式就是断点调试和通过调试信息定位。在真机测试的情况下，显然我们无法使用断点调试的方式，因此重点就落下调试信息上。调试信息通常是我们为了确保某些代码是否执行而在代码中添加的一些输出信息，也包括代码中的报错。调试信息基本是在控制台输出的，我需要做的就是在真机测试的时候也能查看到调试信息。简单说就是在游戏界面中做一个类似UnityConsole的界面。

首先，获取Unity的调试信息。基于最基本的编程常识，Unity一定有开放调试信息的接口。因为很多游戏有日志保存功能，日志保存的内容其中一部分就是调试信息。因此通过对Unity官方文档的搜索，可以找到调试信息接口:Application.logMessageReceived。这是一个委托事件。它会传三个参数，第一个是调试信息，第二个是调试信息的堆栈信息，第三个就是调试信息的类型。我建了一个对应的结构体来接收这三个信息：

struct Log

{

public string message;

public string stackTrace;

public LogType type;

}

接着开始模拟UnityConsole绘制到游戏界面中。除此之外还有三个重要功能需要模拟，第一是显示堆栈信息，第二是折叠重复信息，第三清空调试信息。

整个工具用一个类来实现即可，建立一个TestConsole类，继承自MonoBehaviour。绘制界面的部分需要借由MonoBehaviour中的OnGUI方法来实现。再者用HandLog方法来接收调试信息，再将每条信息暂存到List<Log>中用于后续绘制时使用。

在OnGUI中先绘制一个GUILayout.Window，在其中再绘制工具栏、Log列表、堆栈信息标签。

工具栏包括一个Clear按钮和一个Collapse勾选。这两个工具实现清空调试信息和折叠重复信息。

Log列表中的每一行有两个组件，一个是显示调试信息的标签，另一个是显示堆栈的按钮stackTrace。点击某一行的stackTrace按钮，堆栈信息标签就会显示该行调试信息的堆栈内容，便于我跟踪代码。

最后需要一个控制显示的布尔变量，我不希望这个界面时时存在与游戏运行过程，指定某个按键来控制界面的显示与隐藏，平时默认为隐藏，在测试出现问题的时候再手动显示这个界面查看调试信息。