# 第四章 游戏内容具体实现

## 4.1 游戏框架

以下是游戏框架图，纵向的箭头表示继承关系，横向箭头表示成员控制关系。

GameRoot是继承MonoBehaivour的，在GameRoot的Awake()方法中对UIMgr等成员进行初始化操作，UIMgr中的Update()方法则在GameRoot.Update()中调用。

### 4.1.1 UI框架

UI框架对于游戏来说十分关键，因为程序的流程基本就是通过UI框架控制的UI面板进行的。下图是UI框架的脑图。

UI框架中最主要实现功能的部分在UIFacade中，但UIFacade是由UIMgr调用的，一般外部无法直接访问UIFacade。UIMgr继承单例模板。其次所有第二章提及的面板都是继承BasePanel。

比如要显示一个MainPanel面板的时候，UIMgr.Instance.Show(UIPanelName.MainPanel)，执行这一步后，如果是第一次执行，UIFacade会向UIBusiness获取MainPanel类，然后获取MainPanel的UIPanelInfo信息，得知预制体的保存地址和UI面板的层级，再加载预制体并将预制体与MainPanel类进行关联。显示的面板会添加到panelShowDict字典中，被隐藏时则会从字典中移除，只有在字典中的面板的BasePanel.Update()才会被调用。

当游戏程序更换场景时，旧场景的面板资源就要被释放，就会BasePanel.OnDestory()。

### 4.1.2资源加载与工厂

一般大型的游戏对资源的管理是十分看重的，因为大型游戏对内存的占用非常大，如何合理的运用内存资源就很关键。一旦超出内存导致游戏卡死，或者程序频繁调用垃圾回收机制（GC）引起游戏卡顿，这就会导致玩家体验感极差。本次课题的游戏是小游戏，内存问题一般不会存在，但是管理好内存是一个游戏开发者应该做的。

下图是关于资源加载部分的脑图：

其中FactoryMgr负责外部调用的，比如成就系统列表要加载一个成就的UI，则需要这样调用：FactoryMgr.Instance.GetUI(uiPath)。回收这个UI的时候就是FactoryMgr.Instance.PushUI(uiPath,uiGO)。回收的时候就是将UI加入到对象池中。

在脑图中的UIFactory和GameFactory均继承自BaseFactory，BaseFactory管理需要重复生成的游戏物体的对象池。这里引入了对象池的概念，对象池就是意图通过重用固定池中的对象而不是单独分配和释放它们来提高性能和内存使。用UIFactory管理的比如MainPanel中的关卡按钮一般最多有三十个，可以用对象池管理。GameFactory就比如防御塔、子弹、敌人等。

其他的SpriteFactory、AudioFactory则是继承BaseRescourceFactory，类似Sprite、AudioClip这类资源加载一次即可在游戏进程中一直使用，因此只需要用字典来进行记录，保证不重复加载即可。

### 4.1.3消息机制

一个健壮的程序系统一般要求低耦合、高内聚。我们通常不希望系统中的模块之间有太多的依赖，希望每个模块都尽可能独立完成自己的功能。但是模块之间也不可能毫无交互，甚至有时候一个模块的某个操作需要跟不止一个模块进行处理。为了保证模块的独立性，我们一般把它们的交互交给一个稳定的第三方来处理。设计模式中有一种模式适用于上述提及的情况，那就是观察者模式。在程序中一般用来搭建消息机制，使用一个消息处理中心来达到解耦的目的。

本次课题的消息机制一共就三个文件，EventCenter、EventType、CallBack。

（1）CallBack很好理解，就是泛型的委托，定义了从无参数到五个参数六种泛型委托。

（2）EventType，这是一个枚举。消息号就是EventType里面定义的枚举，每次需要新的消息类型，就在EventType中定义即可。

（3）EventCenter，消息机制处理中心。有三个核心的公共静态方法提供给程序模块使用：

AddListener(eventType,callBack)：注册监听事件；

RemoveListener(eventType,callBack)：移除监听事件；

Broadcast(eventType,……)：消息广播。

在EventCenter中使用字典来记录事件，key值是EventType，value值则是Delegate。Delegate本身就像一个链表，可以通过+-的符号进行叠加和删减。在Broadcast的时候也是直接使用Delegate，不需要进行遍历就可以将全部方法都调用。

### 4.1.4建造者模式

建造者模式也是一种设计模式，目的是将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。建造者模式经常跟工厂模式配合使用，工厂产生零件，建造者负责组装。本次课题里面实现的建造者模式形式十分简单，EnemyBuilder、BullectBuilder、TowerBuilder都继承了一个IBuilder<T>接口，生成对应的方法进行实现。

主要是四个方法：

GetProduct()：返回GameObject，在这个方法中调用FactoryMgr获得游戏物体。

GetProductClass(GameObject gameObjct)：返回T，在这个方法中获取游戏物体的组件进行配置。

GetData(T productClassGo)：GetProductClass返回的T传入该方法进行数据配置。

GetOtherResource(T productClassGo)：个别特殊的实现在这里进行。

在本次课题中如果需要生成一个Enemy，就会通过EnemyBuilder调用GetProduct()，返回的GameObject就是配置好的了，可以直接使用。GetProduct()中的逻辑大致如下：

Public GameObject GetProduct()

{

GameObject go=FactoryMgr.Instance.GetGame(敌人预制体地址);

BaseEnemy enemy=GetProductClass(go);//获得敌人身上的BaseEnemy组件

GetData(enemy);//在GetData方法中对BaseEnemy组件进行配置

}

### 4.1.5玩家数据存档与读取

关于玩家数据的存档有很多种方式，我采取的是二进制文件。二进制文件基本不具备可读性，玩家不能自行修改数据，对开发者来说就可以避免作弊的情况发生。而且二进制文件的读写是最快的。而JSON和XML都是可读性比较强的文件，玩家可以轻易读懂数据的保存形式，这就不是开发者所希望的。实现玩家数据存档和读取需要两个类：

PlayerData：玩家数据类

PlayerDataOperator：玩家数据操作类

PlayerData记录的内容也很简单，主要是完成关卡的情况、成就达成情况、玩家持有的道具、玩家持有的钻石数量。

PlayerDataOperator有三个公共方法：

LoadPlayerData()，加载玩家数据，在游戏启动时调用。

SavePlayerData()，在各种需要存档的时候调用，如游戏结束、购买道具等。

ResetPlayerData()，玩家删除存档的时候调用。

## 4.2 游戏场景与UI面板

### 4.2.1 场景切换

为了场景切换时视觉上的过渡自然，我在场景退出前会先显示一块黑色的遮罩，进入新的场景时又将这块黑色遮罩隐藏。遮罩的出现和隐藏也要尽可能自然，所以使用的DOTween的To方法，在一定时间内通过修改图片的颜色Alpha值达到渐隐或者渐现的效果。

因为遮罩只在场景切换时使用，我将遮罩的管理也放在SceneStateMgr中。当SceneStateMgr被初始化时，遮罩也会被初始化。在SceneStateMgr中带有两个私有方法，一个是ShowMask()，另一个是HideMask()，分别用来控制遮罩的显示和隐藏。

我为每个场景都建立一个场景状态类，这些类都继承BaseSceneState类。它们只有两个公共方法需要实现。一个是EnterScene()，另一个是ExitScene()。在进入场景和退出场景需要做什么工作都在这两个方法内调用。

场景状态类一共五个：StartLoadSceneState、BeginSceneState、MainSceneState、GameLoadSceneState、GamePlaySceneState。

如果要从StartLoadSceneState切换到BeginSceneState，就要执行SceneStateMgr.Instance.ChangeSceneState(new BeginSceneState)。在场景管理类中首先会执行ShowMask()先显示遮罩，当遮罩完全显示时会执行一个回调，我在这个回调中处理场景StartLoadSceneState.ExitScene()和BeginSceneState.EnterScene()。这样一个场景的切换所涉及的工作就全部完成了。

### 4.2.2 UI面板

本次课题一共有13个UI面板，这边主要介绍几个功能比较多、处理时比较困难的两个面板，其他面板的处理方式基本在这两个面板中都能体现。

1. **MainPanel**

MainPanel涉及到ShopPanel、AchievementPanel、HelpPanel、LevelIntroducePanel的显示。与第四个关卡介绍面板LevelIntroducePanel关联的关卡按钮的管理比较复杂。

我在MainPanel类中建立了一个内部类是LevelButton，继承BaseUIListItem。MainPanel在初始化的时候就会实例所有的LevelButton。每个LevelButton在构造的时候会根据自己的编号获取关卡信息和星级信息并显示，然后注册按钮点击事件。ShopPanel、AchievementPanel、HelpPanel中关于道具按钮、成就项、敌人按钮、防御塔按钮都是依照这种方式进行的。

MainPanel中有一个横版地图，所有的关卡按钮会显示在地图上，玩家通过拖拽的方式来查看地图的全貌。“拖拽”如何实现，是一个困扰我的问题。常见的实现拖拽方式都是继承几个Unity的接口，一般是IBeginDragHandler、IDragHandler、IEndDragHandler这三个接口。这三个接口非常方便，但是必须继承MonoBehaviour。我们的面板类并没有继承MonoBehaviour。当然我可以直接建一个MonoBehaviour的子类挂载在横版地图上，然后继承这三个接口，但这不利于程序管理，尤其是迁移项目的时候，挂载在物体上的类容易丢失。

后来通过Unity官方文档，我还是找到了一个方法，那就是EventTrigger组件。实现方法如下：

**EventTrigger** mImg\_MapEventTrigger;

**EventTrigger.Entry** onDragEntry;

onDragEntry = new EventTrigger.Entry();

onDragEntry.eventID = EventTriggerType.Drag;

onDragEntry.callback.AddListener((data) => { OnDrag((PointerEventData)data); });

mImg\_MapEventTrigger.triggers.Add(onDragEntry);

1. **GamePlayPanel**

这个面板主要功能就是显示战场的金币、波次、生命值、暂停按钮。点击敌人或者防御塔的时候还负责显示敌人或防御塔的部分信息。这部分的工作其实最主要的就是消息机制的运用。敌人的死亡或者到达终点都会影响面板的金币和生命值显示。最直接的办法就是每个敌人都持有面板的引用，然后面板将金币和生命值的变量设置为公共变量，提供给敌人修改。但这不是我希望的，也不符合模块独立的要求。

为了解决金币值等更新的问题，就需要使用消息机制。面板显示的时候就注册金币更新的消息。当敌人死亡的时候广播更新金币，并传递增加的金币数。面板接收到消息后就自动更新金币值。

当我们点击某个敌人或者防御塔，要查看它们的信息时也是通过消息机制来处理。敌人被点击时触发OnMouseDown()方法，我在该方法中进行广播敌人被点击的消息，并传递敌人的信息。面板接收到后，显示敌人信息的面板并更新面板上的信息即可。

在该面板中还有关于道具使用的部分，这部分加载道具的逻辑与MainPanel中的LevelButton相同。道具也是按钮。道具被点击后发送消息和传递自身的编号，根据编号接收消息方执行相应的操作。如金币道具，就会让GamePlayPanel更新金币数值。物理加成道具就会广播全部敌人提高物理伤害。

## 4.3 游戏玩法逻辑

### 4.3.1防御塔

关于防御塔有两个基类，一个是BaseTower，还有一个是TowerProperty。

1. BaseTower

BaseTower继承MonoBehaviour，主要功能很简单，就是确认范围内的敌人目标。

在敌人身上挂载了碰撞器之后，就可以通过OnTriggerEnter、OnTriggerStay、OnTriggerExit来获取进入范围内的敌人。我用一个链表保存所有在范围内的敌人，敌人进入时添加到链表，离开时从链表移除。当敌人死亡或者到达终点时，还在范围内的敌人也要从链表中移除。

BaseTower在初始化的时候会注册敌人移除的消息事件。当敌人死亡或到达终点时会向所有防御塔广播。接收到消息的防御塔就会从链表移除这个敌人。

1. TowerProperty

TowerProperty负责攻击、冷却计时、塔的转动。

因为本次课题的游戏有四种防御塔，不同的防御塔有不同的需求。我通常是再建立一个类继承TowerProperty，重写里面的部分方法来实现的。比如慢速塔和法师塔并不需要防御塔转动瞄准，重写的时候就把防御塔转动的方法置空就可以了。

只要防御塔攻击目标不为空且不在冷却期间，防御塔就会执行攻击方法。

private void Update()

{

if (isBeginCD)

{

if (timeVal >= baseTower.towerInfo.CD)

{

timeVal = 0;

isBeginCD = false;

}

else

timeVal += Time.deltaTime;

}

if (target == null || GameController.Instance.isPause == true)

return;

TransformRotate();

if (isBeginCD == false)

Attack();

}

### 4.3.2敌人

在本次课题，敌人就一个基类BaseEnemy，没有其他的子类。对于敌人的最大需求就是按照指定的路径行走到终点为止。敌人移动的实现是使用插值运算，已知敌人当前位置和目标位置，调用Vector3.Lerp来实现，它有三个参数，头两个是起始位置和目标位置，第三个参数是移动的比例。在使用插值运算方法时，如果移动比例设置为一个固定值，敌人越接近目标点，移动的速度就会越慢。因为每次移动都是按照当前距离的一个比例来移动。而距离是缩短的，这就导致每次移动的距离也在缩短。

为了达到移动的匀速进行，我就把移动距离单位化，使得每次移动都按照单位距离移动。

position = Vector3.Lerp(起点, 终点,1/Vector3.Distance(起点, 终点) \* Time.deltaTime \* enemySpeed);

解决了移动的问题之后，第二问题就是敌人的转向。这个主要是根据当前位置与目标位置的XY差值来处理。2D角色的转向表现是通过播放不同的动画实现的。因此根据差值的正负判断播放对应的动画即可。

if (xOffset < 0)

animator.Play("Right");

else if (xOffset > 0)

animator.Play("Left");

else if (yOffset > 0)

animator.Play("Front");

else if (yOffset < 0)

animator.Play("Back");

### 4.3.3子弹

子弹基类为Bullect当防御塔执行攻击方法时就会调用子弹的建造者产生子弹在防御塔位置，然后将攻击目标赋给子弹。

子弹移动方式并不是物理的方式，跟敌人移动是一个逻辑，不再赘述。

当子弹与敌人的碰撞器接触时，子弹就会向敌人发送伤害的消息。这边使用的不是上文提及的消息机制。而是Unity自己的SendMessage方法。因为游戏框架中的消息机制是广播性质的，没有针对个体的，当然这是可以实现的，但是既然Unity已经有了，就直接拿来用。火炮塔和弓箭塔的子弹基本就是走上述逻辑，慢速塔和魔法塔则又不同。

慢速塔的子弹严格来说不是“子弹”，只是一个圆形动画，在防御塔基础上播放，因此没有移动的逻辑。慢速塔的攻击是范围攻击，也就是遍历攻击前面防御塔提到的敌人链表中的每个敌人。慢速塔也会发送攻击消息，但同时还会再发送一个减速消息。

魔法塔的子弹是线性闪电，所以要重写移动的方法。线性闪电有两种实现方式，一种是使用LineRender组件，还有一种是在目标点和起始点中间生成与距离等长的闪电。我采取的是第二种方式。