# 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

场景切换

加载面板

带UIListItem的面板

GamePlayPanel

TowrSetPanel

4.3 游戏玩法逻辑

敌人与子弹移动

防御塔目标设置

攻击

声音处理