代码架构设计

代码架构设计（**一定要自顶向下编程，先做大厅再做关卡。不然，接口对不上，未来的接口设计会相当复杂。**）

1. 总体架构：模块划分图（按照职能划分）

1.1 应用核心

这个模块是游戏的“骨架”，负责最顶层的流程控制和通用管理，不包含具体游戏规则。

- `GameManager`：游戏的真正总入口。负责游戏启动、退出、全局事件分发。它持有其他核心管理器的引用。
- `GameStateManager`：**这是实现全流程设计的核心**。它用一个状态机来管理游戏处于哪个大阶段（主菜单、大厅、对战、设置等）。它只负责状态的切换，不处理具体界面显示。
- `SceneManager`：封装Unity的场景加载、卸载、过渡动画（如Loading界面）。
- `InputManager`：**全局输入处理**。统一管理所有平台的输入（键盘、鼠标、手柄），将原始输入转化为抽象的“游戏动作”（如“打开菜单”、“确认”），并允许在不同游戏状态下响应不同的输入集。
- `SettingsManager`：全局设置的**数据管理**。它负责读取、保存、应用所有设置（音频、画面、键位、语言）。它本身不包含UI。

1.2 游戏逻辑

这个模块是游戏的“大脑”，包含所有自走棋特有的规则和玩法。

- `BattleCore` **(对战核心)**:
  - `BattleManager`：控制单局对战的完整流程（准备阶段 -> 战斗阶段 -> 结算阶段）。
  - `BoardManager`：管理棋盘网格逻辑，处理棋子的放置、移动、格子状态。
  - `ChessPiece`：棋子的逻辑实体，包含生命、攻击等属性和行为。
  - `PieceManager`：管理场上所有棋子的生成、回收和阵营关系计算。
- `MetaGame` **(元游戏)**:
  - `MetaProgressionManager`：管理**科技树**的解锁和升级。
  - `NarrativeManager`：管理**剧情**和**任务**的触发与进度。

- `PlayerProfile`: 代表玩家档案，包含金币、解锁内容等跨局数据。

1.3 用户界面

这个模块是游戏的“皮肤”，负责一切显示和界面交互，它依赖于 `游戏逻辑` 和 `应用核心` 模块。

- **`UIManager`**: UI层的总调度员，负责打开、关闭、切换不同的UI面板。
- **`UI Panels` (各种UI面板)**:
  - `View_MainMenu` (主界面视图)
  - `View_Lobby` (大厅视图)
  - `View_Battle` (对战视图)
  - `View_Settings` (设置视图)
  - `View_Archive` (存档选择视图)
- 每个 `View` 只关心**如何显示**和**接收玩家输入**。当有按钮点击时，它应该去调用 `GameStateManager` 或 `BattleManager` 等逻辑模块的方法，而不是自己处理游戏逻辑。

1.4 数据与配置

这个模块是游戏的“记忆库”，用于存储和定义各种数值。

- **`Data Models` (数据模型)**: 定义核心数据的C#类。
  - `GameSaveData` (存档数据)
  - `SettingsData` (设置数据)
  - `ChessPieceData` (棋子属性)
- **`ScriptableObjects` (配置资产)**: 在Unity编辑器里配置的资产文件，非常适合用来配置棋子属性、关卡数据、技能效果等。
- **`Localization` (本地化系统)**: 管理多语言文本。

1.5 视听资源

这个模块是游戏的“血肉”，管理所有美术和音效资源。

- **`AudioManager`**: 统一播放背景音乐和音效，与 `SettingsManager` 联动控制音量。
- **`Visual Assets`**:
  - 棋子的Sprite和动画控制器。
  - UI的图集、字体。
  - 场景的背景、特效等。

1.6 工具与底层

这个模块是“工具箱”，提供可复用的代码片段。

使用泛型事件总线（EventBus.cs）、服务定位器(ServiceLocator.cs)进行事件的传输以及安全的访问。

- `Extensions`：对C#或Unity内置类的扩展方法。
- `Utilities`：通用工具类，如自定义的数学库、协程管理器、对象池等。
- `Service Locator` / `Dependency Injection`：一个简单的框架，用于解耦模块之间的直接引用，让 `AppManager` 能轻松地将 `SettingsManager` 提供给 `AudioManager` 和所有 `View` 使用。

2. 数据流图

代码如下

- graph TD
  subgraph 输入
  P[玩家输入]
  end

- P --> IM[InputManager
应用核心]

  subgraph 核心引擎
    IM --> GSM[GameStateManager
应用核心]
    GSM --> UM[UIManager
用户界面]
    GSM --> BM[BattleManager
游戏逻辑]
    GSM --> MGM[MetaGameManager
游戏逻辑]
  end

  UM --> |调用显示方法| V_MainMenu[View_MainMenu
用户界面]
  UM --> |调用显示方法| V_Battle[View_Battle
用户界面]
  UM --> |调用显示方法| V_Settings[View_Settings
用户界面]

  V_MainMenu --> |发出“开始游戏”指令| GSM
  V_Battle --> |发出“放置棋子”指令| BM
  V_Settings --> |修改设置数据| SM

  BM --> |请求棋子数据| SD[棋子Data
ScriptableObject]
  BM --> |写入回合结果| PD[PlayerProfile
玩家存档数据]
  BM --> |播放音效指令| AM[AudioManager
视听资源]

  MGM --> |请求科技树数据| TD[科技树Data
ScriptableObject]
  MGM --> |写入进度| PD

  SM[SettingsManager
应用核心] --> |应用音量设置| AM
  SM --> |保存/加载| SDATA[SettingsData
数据]

  AM --> |加载/播放| A_ASSETS[音效/音乐文件
视听资源]
  V_Battle --> |加载/显示| V_ASSETS[UI精灵/动画
视听资源]

  PD --> |序列化/反序列化| SAVE[存档文件
数据]

3. 接口定义（要求全部使用ScriptableObject配置）

一边写代码，一边添加接口。

依赖关系要求

依赖关系只允许单向依赖，不可以循环。

4. 数据模型

- 数据库设计（单机游戏/弱联网：使用文件存储）
- 缓存策略（自走棋中，考虑计算缓存和资源缓存。局外的开放世界探索暂不考虑。）
- 序列化协议（将局外科技树、剧情进度、任务系统、大厅探索进度保存在硬盘上，主要使用JSON协议。）

5. 非功能性需求

- 性能指标（QPS、延迟）（写脚本开始时考虑）
- 容灾方案（写脚本开始时考虑）

6. 具体要求

1. （scriptbleObject来做编辑器内的数据调用）
2. （用事件来进行通信。）。注意分析代码耦合性。
3. 中央协调器来管理系统间的依赖，避免findObjectOfType。
4. A*算法寻路，同时使用添加缓存的方式优化性能。
5. 用状态机提升代码清晰度
6. 配置sprites的导入设置（保证导入的统一）
7. InputSystem进行输入

可选

使用object Pooling System插件创建对象池——当英雄和敌人的数量多、种类少时。否则不推荐用，配置麻烦。

2. 可使用的插件

lean、Dotween、TextMeshPro、Unity UI、InputSystem