# 代码规范

## 条件控制语句

如果代码块中仅包含一条代码，且没有else分支，则要求不添加花括号。

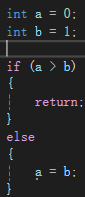
若存在分支语句，即使代码块中仅包含一条代码，也要求添加花括号。

代码块不与条件语句在同一行。

例子1：



例子2：



## 循环语句

不适用do-while语句

while与for的条件语句不与代码块在同一行

即使代码块中只有一条代码，也要求添加花括号

## 变量

采用驼峰命名法，公有与保护变量采用小写字母开头，私有变量则在此基础上在开头添加一个下划线。

例子：



## 方法

采用驼峰命名法，无论共有、私有还是保护方法，均以大写字母开头。

命名时应当以动宾短语为主。

## 类

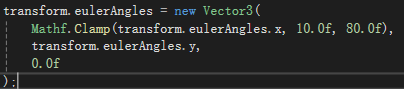
采用驼峰命名法

## 其他

1. 长语句处理

如果在调用方法或者新建实例时，一行代码的长度过长，应当进行换行分割。

例子：

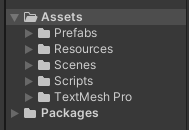


1. 中文与拼音？

原则上来讲，不允许出现任何以中文或者是拼音命名的变量、类或者方法等。但如果实在无法将当前内容转化为英文，可以临时使用拼音。

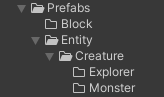
1. （占位符）

# 文件夹结构



项目中的所有文件都保存在Assets下，Package则是保存Unity提供的插件包。而Assets中的TextMesh Pro则是Unity提供的UI支持，不属于我们编写的内容。

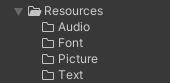
## Prefabs



Prefab表示预制体，在Unity的场景中被创建出的游戏物体可以被单独保存到项目目录下，并用于代码调用生成出相同副本。

Prefabs文件夹则用于存储所有需要被生成的游戏物体。

## Resources



Resources文件夹用于存储各类图像、音频、文本等美术资源。

## Scenes



Scenes文件夹用于存储Unity的游戏场景。

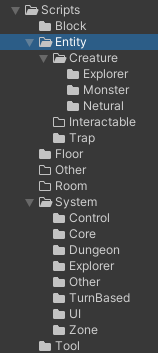
在各位第一次导入仓库中的项目文件时，各位可能会发现编辑器中处在的场景依然是空的，这是因为编辑器还在使用默认场景，各位需要手动前往Scenes文件夹中，通过双击场景文件来切换场景。

Scenes文件夹的根目录下保存的场景文件为核心场景文件，是最后项目导出时会用上的文件，而Test文件夹中则保存着测试场景。其中，StartMenu为主界面场景，Dungeon为游戏场景。

如果需要测试自己的开发的内容，不应当在核心场景文件中测试。建议把核心场景文件复制一份并放入Test文件夹中，然后更改场景文件名字以标识其作用与测试内容，随后在测试场景中进行测试。

## Scripts

Scripts文件夹中存储了项目中的所有代码文件，在其下又额外分支了各个文件夹，用于存储特定类别的代码文件。



目前大多数代码文件都是按照对应功能划分在特定文件夹中的，仅有小部分不足以单独分类的代码文件被放入了Other中。

在开发过程中，各位可以在自己负责的系统的对应目录下自行新增文件夹和代码文件，必要的话，也可以在System或者Scripts等文件夹再新增文件夹分支。但请注意，如果要删改必须要提前通知。

# 系统结构

尽管项目规模实际上并不算大，但我仍然为多种功能分配了专属的管理类作为中转调用。在功能实现上，这些中间层可能并不必要，但通过分层的调用可以一定程度上让我们能够更清晰地切分功能以及任务，方便我们进行协同开发。

下面我会对目前已经存在的部分管理类以及其他抽象类、工具类进行功能简述，并说明仍需要补足的部分以及完善方向。一些类的内容说明可能会十分简略，因为我也还没完善这部分的内容，所以没办法展开讲太多。

## 主界面的管理类

#### MenuManager

该类是主界面中的核心管理类，负责中转调用以及管理游戏进程。

在原本的构想中，实际上是可以将主界面的游戏管理类与游戏场景中的游戏管理类合并使用的，但实际上，考虑到主界面的功能在游戏场景中的复用性较低，因此单独拆分出MenuManager类作为主界面中的核心游戏管理类。

类似MenuManager这样的核心管理类并不需要专门分配给某个人进行维护开发，作为中转站，所有项目成员都可以为其添加方法，以便自己的系统去调用另外的系统。但是，如果涉及到某个非自己编写的方法上的删改，则需要小心，最好提前说明或者通知。

例子：

中转功能：当MenuUIManager收到了来自UI的请求时，它会进一步去调用GameManager中对应的方法。随后GameManager会调用对应方法去向其他类实际实现的方法。

游戏进程的管理：MenuManager在收到退出游戏的请求时，会调用自身的退出游戏方法。

#### MenuUIManager

由于UI的显示与关闭以及文字内容修改操作较为繁琐，同时，为每个单独的UI编写脚本又会导致类爆炸的问题，因此需要一个特定的管理类进行管理。原则上， UI都需要通过一个UIManager来进行实际操作的调用。

在主界面中，负责管理按钮的类即MenuUIManager。如果之后需要编写设置面板，考虑到设置面板中可能会同时存在过多不同类型的操作需求，可能需要单独再分出一个UIManager。

## 游戏场景的管理类

#### GameManager

该类是游戏场景中的核心管理类，负责中转调用以及管理游戏进程。

类似这样的核心管理类并不需要专门分配某个人进行维护开发。作为中转站，所有项目成员都可以为其添加方法，以便自己的系统去调用另外的系统。但是，如果涉及到某个非自己编写的方法上的删改，则需要小心，最好提前说明或者通知。

#### GameUIManager

负责管理主界面中部分UI的显示与关闭以及修改的管理类。

由于UI的显示与关闭以及文字内容修改操作较为繁琐，同时，为每个单独的UI编写脚本又会导致类爆炸的问题，因此需要一个特定的管理类进行管理。原则上， UI都需要通过一个UIManager来进行实际操作的调用。

#### PlayerInputManager

该类用于管理玩家的输入。

考虑到同一个按键操作可能会同时影响或调用多个类的方法，所以采用了Event来管理玩家操作的触发。

如果后续有需要添加的玩家操作，可以在仔细阅读该类的方法中的代码后自行添加。

#### DungeonManager

该类负责存储并管理玩家的资源，并对玩家在地牢建造阶段的操作进行管理。

#### ZoneManager

该类负责初始化地牢中每个区域的地图，并提供区域的切换与调用。

该管理类我已经开发了一部分内容，目前它已经实现在场景中创建存储区域的游戏物体，并用列表进行保存。

#### CameraManager

该类用于管理游戏场景中的摄像机。

考虑到与摄像机有关的操作具有独特性，所以单独分出来一个管理类。

#### ExplorerManager

该类用于管理探险队的生成。

#### TurnBasedManager

该类用于管理处于时间流逝阶段下的各个实体的回合制行动次序。

## 游戏中的实体

#### AbstractEntity

地牢中的所有可以自主行动的物体都需要装载继承了AbstractEntity的组件。在该抽象类中，定义了地牢中的实体应当执行的基本操作，需要子类实现。

为了方便协作，我在AbstractEntity中设置了PreUpdate、MidUpdate、PostUpdate以及PreAwake、MidAwake、PostAwake，它们各自在Update与Awake函数中被按顺序调用，在最终实现时，也可以通过实现这三个函数来区分不同功能的调用时机。

#### AbstractCreature

在地牢中，可以进行移动的物体则被认为是生物，需要装载继承了AbstractCreature的组件。AbstractCreature本身继承了AbstractEntity。在该抽象类中，定义了地牢中的生物应当执行的基本操作，需要子类实现。

#### AbstractExplorer

探险者作为能够自主行动的独立生物类别，单独将其划分出一个抽象类。AbstractExplorer本身继承了AbstractCreature。在该抽象类中，定义了地牢中的探险者应当执行的基本操作，需要子类实现。

#### AbstractMonster

怪物作为能够自主行动的独立生物类别，单独将其划分出一个抽象类。AbstractMonster本身继承了AbstractCreature。在该抽象类中，定义了地牢中的怪物应当执行的基本操作，需要子类实现。

#### AbstractNetural

中立生物实际上与怪物和探险者很相似，但考虑到当前项目的进度问题，并没有直接着手构建中立生物相关的系统。

#### AbstractTrap

陷阱虽然也可以自主行动，但初步认定陷阱应当不可移动，又因为陷阱在玩法中占据相对重要的地位，所以将其单独划分为一类。AbstractTrap类继承了AbstractEntity类

## 游戏中的场景

#### AbstractFloor

考虑到地牢中的地板可能会有多种不同的地板，因此设置抽象地板类，原则上所有作为地板存在的游戏物体都需要装载继承了AbstractFloor类的组件。

当然，实际上如果给每一块地板物体都额外装载一个组件，可能会导致占用过多内存空间，也可以考虑仅给地板物体打上标签来识别不同的地板。

#### AbstractBlock

考虑到地牢中的墙壁可能会有多种不同的墙壁，因此设置抽象方块类，原则上所有作为墙壁存在的游戏物体都需要装载继承了AbstractBlock类的组件。

当然，实际上如果给每一块墙壁物体都额外装载一个组件，可能会导致占用过多内存空间，也可以考虑仅给墙壁物体打上标签来识别不同的地板。

## 其他

#### SceneManager

在Unitiy中，每个场景中的所有GameObject都被存储在Scene文件中，如果游戏系统有需求，可以调用Unity提供的方法进行场景切换。由于这部分代码相对比较独立，因此单独划分一个管理类来调用Unity中管理Scene的方法并切换游戏场景。

Unity的Scene的加载方式有两种。

第一种为直接切换，这样会导致当前场景中的物体全部被销毁，并在理论上移出内存空间，随后再把新的场景中的文件再加载进来。这样会使得我们无法跨场景传输游戏物体，但Unity也提供了相应的方法来应对这种需求，比如将游戏物体直接传输到另一个场景，或者是为某个游戏物体打上特殊标记，使其不会在转换场景时被摧毁。

第二种为叠加场景，原本的场景会被保留，而被加载的场景会叠加在先前的场景上并在Hierarchy面板中以两个不同的场景分别留存其中的游戏物体。

#### DebugManager

DebugManager目前还没有实际开发。

在早先的考虑中，它的作用是为开发者在游戏运行时提供一些必要的调试手段，方便测试，例如能够进行特定功能的调用的UI面板，或者可以输入指令的控制台。不过考虑到目前项目还没有需要调试的内容，因此该管理类还没有实际开发。

#### FetchComponent

在FetchComponent类中只存在一个静态方法GetSpecificComponent。由于Unity本身没办法直接从GameObject上抓取以某个类为基类的组件，所以编写了一个通用的静态方法，通过遍历输入的GameObject的上挂载的所有组件，来找寻以输入泛型为基类的组件，并将遇到的第一个符合要求的组件以Component的形式返回。

使用例子：

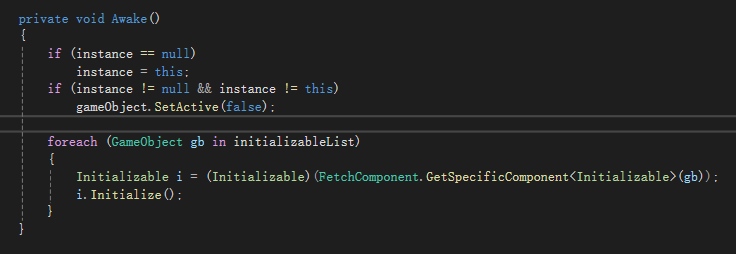


在该处例子中，由于Initializable是一个接口，无法直接通过Unity提供的方法直接获取GameObject上扩展了该接口的组件，所以调用FetchComponent.GetSpecificComponent，并输入Initializable作为泛型变量的值。最后，根据需求，将返回的Component强制转换为Initializable类型。

#### Initializable

在系统初始化以及其他情况下，可能会出现需要考虑先后运行次序的情况。尽管Untiy中的MonoBehaviour类准备了Awake与Start，但仍然不足以厘清所有可能会出现的复杂的的先后次序问题。因此编写了Initializable接口，并让管理类负责按次序统一执行初始化操作。

使用例子：



在该处例子中，考虑到各个系统类之间可能存在需要在初始化阶段先后调用获取信息的情况，因此专门在GameManager中设置initializableList，人工将需要排序的系统类在Unity的Inspector面板中放入，并在GameManager的Awake方法中编写遍历语句，逐个调用列表中的对象的Initialize方法。

该例子相对来讲比较草率，仍然需要人工进行排序，可能会出现由于人工偏差失误所导致的系统初始化错误。

后续可以将其优化为由人工前期对扩展了Initializable接口的类标注优先级，由管理类负责收集这些类并根据优先级进行排序。尽管这样并没办法完全剔除人工上的操作，但至少不需要为每个类单独进行排序。