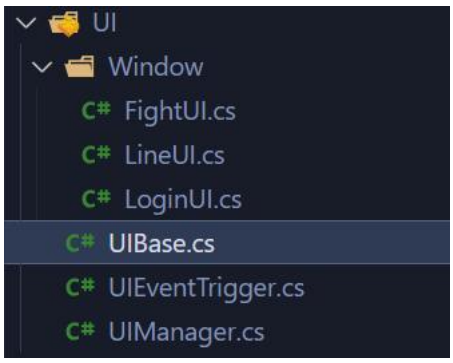
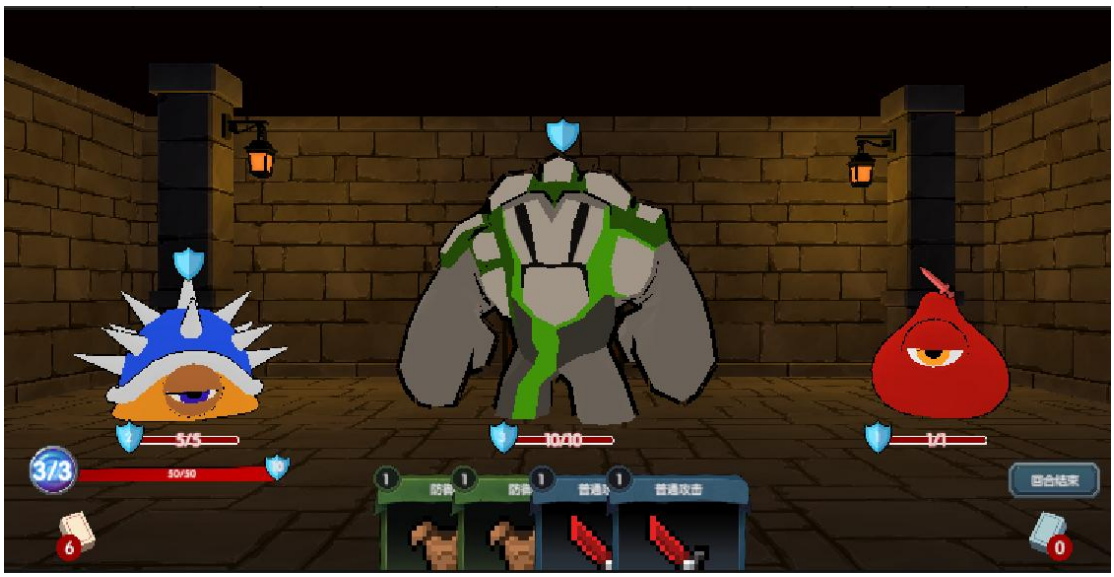
技术设计

一、UI 管理系统

UI 管理系统主要负责游戏中用户界面的实例化、功能实现以及更新，整个系统由若干类共同构成，主要分为 UIManager 类，UIBase 类及其派生类以及 UIEventTrigger 类。



UI 界面示例：



1.UIManager 类

UIManager 类负责管理所有 UI 从资源到游戏内存的实例化，管理 UIBase 类的列表，为 UI 的显示和功能提供了基本条件。其部分功能如下图所示：

```
//ui管理器
29 references
public class UIManager : MonoBehaviour
{
    29 references
    public static UIManager Instance;

    5 references
    private Transform canvasTf;//画布的变换组件

    9 references
    private List<UIBase> uiList;//UI实例列表，存储所有已载入的UI类型


    //关闭界面
    0 references
    public virtual void Awake()…
    3 references
    public UIBase ShowUI<T>(string uiName) where T : UIBase…
    //隐藏UI
    0 references
    public void HideUI(string uiName)…
    //关闭所有界面
    0 references
    public void CloseAllUI()…
    //关闭某个界面
```

其中泛型方法 ShowUI<T>(string uiName)为 UI 实例创建于管理的主要方法，它接受一个 UI 类型的名字如 FightUI，返回并显示该类型 UI。若该类型不存在于管理器的 list 中，则从 Resources 文件夹下加载实例化同名预制体。

```
public UIBase ShowUI<T>(string uiName) where T : UIBase
{
    UIBase ui = Find(uiName);
    if (ui == null)
    {
        //集合中没有 需要从Resources/UI文件夹中更新
        GameObject obj = Instantiate(Resources.Load("UI/" + uiName) , canvasTf) as GameObject;
        //改名字
        obj.name = uiName;
        //添加需要的脚本
        ui = obj.AddComponent<T>();
        //添加到集合进行储存
        uiList.Add(ui);
    }
    else
    {
        //显示
        ui.Show();
    }
    return ui;
}
```

2. UIBase 类

UIBase 类为所有 UI 类的基类，提供了显示、注册事件、关闭等可被子类重写的虚方法，利用多态性实现了 FightUI、LoginUI 等不同的登录界面的不同功能。

UIBase 类实现：

```
public class UIBase : MonoBehaviour
{
    // 注册事件，返回UIEventTrigger对象
    1 reference
    public UIEventTrigger Register(string name)
    {
        // 查找当前UI对象的子对象
        Transform tf = transform.Find(name);
        // 获取或添加UIEventTrigger组件到找到的子对象
        return UIEventTrigger.Get(tf.gameObject);
    }

    // 显示UI，虚方法可以被子类重写
    1 reference
    public virtual void Show()…

    // 隐藏UI，虚方法可以被子类重写
    1 reference
    public virtual void Hide()…

    // 关闭UI，虚方法可以被子类重写
    1 reference
    public virtual void Close()…
```

FightUI 示例：需要在 FightUI 中实现战斗界面玩家血量等 UI 显示更新、卡牌生成与显示、发牌弃牌动画等 UI 逻辑

```
//创建卡牌实体
2 references
public void CreateCardItem(int count)
{
    if(count > FightCardManager.Instance.cardList.Count)
    {
        count = FightCardManager.Instance.cardList.Count;
    }
    for(int i = 0; i < count;i++){
        GameObject obj = Instantiate(Resources.Load("UI/CardItem"), transform) as GameObject;
        obj.GetComponent<RectTransform>().anchoredPosition = new UnityEngine.Vector2(-1000, -700);
        //var item = obj.AddComponent<CardItem>();
        string cardId = FightCardManager.Instance.DrawCard();
        Dictionary<string, string> data = GameConfigManager.Instance.GetCardById(cardId);
        CardItem item = obj.AddComponent(System.Type.GetType(data["Script"])) as CardItem;
        item.Init(data);
        cardItemList.Add(item);
    }
}
```

```
public void UpdateCardItemPos()
{
    // 计算每张卡牌之间的水平偏移量，使它们在水平方向上均匀分布
    float offset = 800.0f / cardItemList.Count;
    // 计算起始位置，使卡牌项在水平方向上居中显示，并且位于屏幕底部
    UnityEngine.Vector2 startPos = new UnityEngine.Vector2(
        -cardItemList.Count / 2.0f * offset + offset * 0.5f, // X 坐标
        -700); // Y 坐标
    // 遍历卡牌项列表，逐个更新它们的位置
    for(int i = 0; i < cardItemList.Count; i++)
    {
        // 使用 DOTween 库中的 DOAnchorPos 方法来动画地更新卡牌项的锚点位置
        cardItemList[i].GetComponent<RectTransform>().DOAnchorPos(startPos, 1.0f);
        // 更新起始位置的 X 坐标，以便下一个卡牌项可以正确地排列在其右侧
        startPos.x = startPos.x + offset;
    }
}
```

```
//将卡牌物体置入弃牌堆
2 references
public void RemoveCard(CardItem item)
{
    AudioManager.Instance.PlayEffect("Cards/cardShove"); //出牌音效
    item.enabled = false; //禁用卡牌逻辑
    //添加到弃牌堆集合
    FightCardManager.Instance.usedCardList.Add(item.data["Id"]);
    //更新使用弃牌堆卡牌数量
    noCardCountTxt.text = FightCardManager.Instance.usedCardList.Count.ToString();
    //从手牌堆中移除该卡牌
    cardItemList.Remove(item);
    //刷新手牌渲染
    UpdateCardItemPos();
    //播放卡牌移动到弃牌堆动画
    item.GetComponent<RectTransform>().DOAnchorPos(new Vector2(1000, -700), 0.25f);
    item.transform.DOScale(0, 0.25f);
    Destroy(item.gameObject, 1);
}
```

3. UIEventTrigger 类

UIEventTrigger 类用于监听鼠标点击事件，挂载于各个 UI 类的实例上，为 UI 点击事件提供了委托注册和处理逻辑。

```
public class UIEventTrigger : MonoBehaviour, IPointerClickHandle
{
    // 事件委托，当对象被点击时触发
    3 references
    public Action<GameObject, PointerEventData> onClick;

    // 静态方法，用于获取或添加UIEventTrigger组件到指定的GameObject
    1 reference
    public static UIEventTrigger Get(GameObject obj)…

    // 实现IPointerClickHandler接口的方法，当对象被点击时调用
    2 references
    public void OnPointerClick(PointerEventData eventData)
    {
        // 如果onClick事件不为空，触发事件并传递当前对象和事件数据
        if (onClick != null)
        {
            onClick(gameObject, eventData);
        }
    }
}
```

二、音效管理系统

由音效管理器 AudioManager 单个类实现，该类为单例模式，在游戏主循环中 start()位置完成初始化与实例化，可被其他脚本在需要播放音乐或音效时调用。

其核心方法为 PlayBGM 与 PlayEffect，两者分别接受需播放音频的名字 string 作为参数，从 Resources 中播放对应音效资源。

```
//播放bgm
2 references
public void PlayBGM(string name, bool isLoop = true)
{
    //加载bgm声音剪辑
    AudioClip clip = Resources.Load<AudioClip>("Sounds/BGM/" + name);
    bgmSource.clip = clip;//设置音频
    bgmSource.loop = isLoop;
    bgmSource.volume = 0.5f;
    bgmSource.priority = 0;
    bgmSource.Play();
}

//播放音效
6 references
public void PlayEffect(string name)
{
    AudioClip clip = Resources.Load<AudioClip>("Sounds/" + name);
    effectSource.priority = 1;
    effectSource.PlayOneShot(clip);//播放音效，和bgm共用一个audioSource对象
}
```

三、信息配置系统

信息配置系统负责从配置文件（由各个 excel 配置表转换而来的 txt 文件）中读取卡牌、敌人、关卡的配置信息，将各个属性的键值对存储在相应字典中，并为其他类提供了查询方法。信息配置系统主要由 GameConfigData 类与 GameConfigManager 类构成。

1. GameConfigData 类

GameConfigData 类为游戏配置表类，卡牌、卡牌类型、敌人、关卡等每一个类型的配置表对应一个配置表类，该类实现了从 txt 中读取配置表信息，并将每一行数据存储到一个字典中，构成一个字典列表，且为其他类提供了根据 Id 查询信息的方法。

配置表示例：

```
Id  Name    Hp  Attack  Defend  Model
Id  名字    血量    攻击力  防御盾  怪物的模型路径
10001   史莱姆  1   1   1   Model/Slime
10002   石头人  10  10  3   Model/Golem
10003   乌龟    5   5   2   Model/TurtleShell
```

```
public GameConfigData(string str)
{
    dataDic = new List<Dictionary<string, string>>();
    //换行切割
    string[] lines = str.Split('\n');
    //第一行是存储数据的类型
    string[] title = lines[0].Trim().Split('\t');//tab切割
    //从第三行下标2开始 开始遍历数据 第二行数据时解释说明
    for(int i=2; i<lines.Length; i++)
    {
        Dictionary<string,string> dic = new Dictionary<string,string>();

        string[] tempArr = lines[i].Trim().Split('\t');

        for(int j=0; j<tempArr.Length; j++)
        {
            dic.Add(title[j], tempArr[j]);
        }

        dataDic.Add(dic);
    }
}

3 references
public List<Dictionary<string, string>> GetLines()···

4 references
public Dictionary<string, string> GetOneById(string id)···
```

2. GameConfigManager 类

该类负责管理所有类型的配置表并为各个类型的配置表提供查询接口。该类为单例模式，便于其他脚本查询游戏配置信息。

```
public static GameConfigManager Instance = new GameConfigManager();

3 references
private GameConfigData cardData; // 卡牌表

3 references
private GameConfigData enemyData;// 敌人表

3 references
private GameConfigData levelData;// 关卡表

2 references
private GameConfigData cardTypeData;// 卡牌类型表

8 references
private TextAsset textAsset;

1 reference
public void Init()···

0 references
public List<Dictionary<string,string>> GetCardLines()···

0 references
public List<Dictionary<string, string>> GetEnemyLines()···

0 references
public List<Dictionary<string, string>> GetLevelLines()···
```

四、战斗逻辑系统

战斗逻辑系统负责回合制战斗的逻辑处理，由 RoleManager，EnemyManager，FightCardManager 以及 FightManager 四个管理器以及与战斗逻辑有关的 Card、Fight 等相关脚本共同构成。它们共同实现了主角、敌人的实例化与初始化，以及双方战斗回合的行动逻辑与功能。

1. RoleManager

功能较为简单，负责管理主角持有的卡牌，维护一个主角目前的牌库，可以进行初始化，添加于删除。

```
public class RoleManager
{
    2 references
    public static RoleManager Instance = new RoleManager();

    12 references
    public List<string> cardList; //存储拥有的卡牌id

    1 reference
    public void Init()
    {
        cardList = new List<string>();
        //四张攻击卡
        cardList.Add("1000");
        cardList.Add("1000");
        cardList.Add("1000");
        cardList.Add("1000");

        //四张防御卡
        cardList.Add("1001");
        cardList.Add("1001");
        cardList.Add("1001");
        cardList.Add("1001");
```

2. EnemyManager 及相关脚本

主要由 Enemy 与 EnemyManager 构成。

其中 Enemy 为敌方个体的管理类，存储相应模型预制体与怪物生命、攻击力等数据，实现了随机动作选择、显示行动意图、攻击受击动画播放与逻辑处理、血量槽与护盾槽更新等功能。

攻击动画播放与逻辑处理部分代码示例：

```
//执行敌人行动
1 reference
public IEnumerator DoAction()
{
    HideAction();
    Debug.Log(transform.gameObject.name + "is doing action " + type);

    //等待某一时刻后的执行行为
    yield return new WaitForSeconds(0.5f);

    switch(type)
    {
        case ActionType.None:
            break;

        case ActionType.Defend:
            //加防御
            Defend += 1;
            UpdateDefend();
            //可以对应播放对应的特效
            break;

        case ActionType.Attack:
            //播放对应的动画
            ani.Play("attack");
            //玩家扣血
            FightManager.Instance.GetPlayerHit(Attack);
```

EnemyManager 为敌方全体的管理类，实现了敌方单位实例化与模型加载、敌方回合行动顺序执行、敌方死亡单位模型移除等管理功能

```
10 references
private List<Enemy> enemyList;
1 reference
public void LoadRes(string id)
{
    enemyList = new List<Enemy>();
    Dictionary<string, string> LevelData = GameConfigManager.Instance.GetLevelById(id);

    string[] enemyIds = LevelData["EnemyIds"].Split('=');
    string[] enemyPos = LevelData["Pos"].Split('=');

    for(int i = 0; i < enemyIds.Length; i++)
    {
        string enemyId = enemyIds[i];
        string[] posArr = enemyPos[i].Split(',');
        float x = float.Parse(posArr[0]);
        float y = float.Parse(posArr[1]);
        float z = float.Parse(posArr[2]);
        Dictionary<string, string> enemyData = GameConfigManager.Instance.GetEnemyById(enemyId);

        GameObject prefab = Resources.Load<GameObject>(enemyData["Model"]);
        GameObject obj = Object.Instantiate(prefab, new Vector3(x, y, z), prefab.transform.rotation);
        Enemy enemy = obj.AddComponent<Enemy>();
        enemyList.Add(enemy);
        enemy.Init(enemyData);
    }
}
```

3. FightCardManager

FightCardManager 为卡牌管理器类，它与卡牌基类 CardItem 及其 AttackCard，DefendCard 等派生类共同管理游戏中的卡牌效果，如战斗开始卡牌初始化、抽牌弃牌逻辑、卡牌功能逻辑以及卡牌选中、拖拽、使用的动画效果以及功能，完善了游戏的卡牌逻辑。

CardItem 类：

提供了所有卡牌类型的通用方法，如鼠标悬停在卡牌上时放大卡牌并高亮边框、拖拽效果等

```
//记录当前元素在父级中的索引
3 references
private int index;
// 当鼠标进入该 UI 元素时触发
2 references
public void OnPointerEnter(PointerEventData eventData)···

// 当鼠标离开该 UI 元素时触发
2 references
public void OnPointerExit(PointerEventData eventData)···

2 references
Vector2 initPos;//拖拽开始时记录卡牌的位置
//开始拖拽
3 references
public virtual void OnBeginDrag(PointerEventData eventData)···

//拖拽中
3 references
public virtual void OnDrag(PointerEventData eventData)···

//结束拖拽
8 references
public virtual void OnEndDrag(PointerEventData eventData)···

//判断当前费用是否满足卡牌使用条件，播放对应音效与提示，刷新文本
3 references
public virtual bool TryUse()···
```

、

FightCardManager 类：
从 RoleManager 中读入玩家牌库实现战斗中抽牌堆初始化、管理抽牌堆与弃牌堆，处理抽牌弃牌逻辑

```
12 references
public List<string> cardList;//卡堆集合

6 references
public List<string> usedCardList;//弃牌堆

//初始化
1 reference
public void Init()
{
    cardList = new List<string>();
    usedCardList = new List<string>();

    //定义临时集合
    List<string> tempList= new List<string>();
    //将玩家的卡牌存储到临时集合
    tempList.AddRange(RoleManager.Instance.cardList);

    while(tempList.Count > 0)
    {
        //随即下标
        int tempIndex = Random.Range(0,tempList.Count);

        //添加到卡堆
        cardList.Add(tempList[tempIndex]);

        //临时集合删除
        tempList.RemoveAt(tempIndex);
    }
```

4. FightManager

FightManager 为战斗管理器类，它维护了玩家血量、能量、护盾等的当前值于最大值，与战斗回合 FightUnit 及其派生类 Fight_EnemyTurn、Fight_PlayerTurn 等共同管理战斗中的回合更替，以及敌我双方回合行动顺序与行动逻辑。
FightManager 类中的回合更替：

```
//切换回合（战斗类型）
6 references
public void ChangeType(FightType type)
{
    switch(type)
    {
        case FightType.None:
            break;
        case FightType.Init:
            fightUnit = new FightInit();
            break;
        case FightType.Player:
            fightUnit = new Fight_PlayerTurn();
            break;
        case FightType.Enemy:
            fightUnit = new Fight_EnemyTurn();
            break;
        case FightType.Win:
            fightUnit = new Fight_Win();
            break;
        case FightType.Loss:
            fightUnit = new Fight_Loss();
            break;
    }
    fightUnit.Init();//初始化
}
```

Fight_PlayerTurn 玩家回合：
弹出提示，回复能量并调用抽卡逻辑、更新 UI

```
public class Fight_PlayerTurn : FightUnit
{
    2 references
    public override void Init()
    {
        Debug.Log("playerTime");
        UIManager.Instance.ShowTip("玩家回合", Color.green, delegate ()
        {
            //回复行动力
            FightManager.Instance.CurrentPowerCount = FightManager.Instance.MaxPowerCount;
            UIManager.Instance.GetUI<FightUI>("FightUI").UpdatePower();
            //卡堆已经没有卡 将弃牌堆卡置入抽牌堆
            if(FightCardManager.Instance.HasCard() == false)
            {
                //直接交换抽排队与弃牌堆的引用
                List<string> temp;
                temp = FightCardManager.Instance.cardList;
                FightCardManager.Instance.cardList = FightCardManager.Instance.usedCardList;
                FightCardManager.Instance.usedCardList = temp;
                //更新卡堆UI
                UIManager.Instance.GetUI<FightUI>("FightUI").UpdateUsedCardCount();
                UIManager.Instance.GetUI<FightUI>("FightUI").UpdateCardCount();
            }
        });
```

Fight_EnemyTurn 敌方回合：
弹出提示并通过协程启动 EnemyManager 中的敌方行动方法，实现多帧的动画播放

```
//敌人回合
1 reference
public class Fight_EnemyTurn : FightUnit
{
    2 references
    public override void Init()
    {
        //删除所有卡牌
        UIManager.Instance.GetUI<FightUI>("FightUI").RemoveAllCards();
        //显示敌人回合提示
        UIManager.Instance.ShowTip("敌人回合", Color.red, delegate ()
        {
            FightManager.Instance.StartCoroutine(EnemyManager.Instance.DoAllEnemyAction());
        });
    }
```

五、卡牌管理系统

卡牌管理系统主要负责具体卡牌的逻辑处理，由 CardItem、AttackCardItem、DefendCard、AddCard 等管理器及卡牌分支脚本构成，具体数量及内容与设计的具体卡牌有关，它们实现了战斗中卡牌相关的动画效果及打出之后的效果处理，除 CardItem 外其余卡牌分支类均继承自 CardItem 类。

1. CardItem

主要负责管理卡牌相关的玩家交互效果处理，如光标悬浮时卡牌放大、离开后缩小至原本大小、拖拽卡牌前后的动画效果。此外它还负责处理玩家尝试打出卡牌时的逻辑，如判断费用是否足够并提示，以及在成功打出卡牌后的特效创建，相关内容均封装于 CardItem 类中。

```
Unity 脚本|8 个引用
public class CardItem:MonoBehaviour, IPointerEnterHandler, IPointerExitHandler, IBeginDragHandler, IDragHandler, IEndDragH
{
    public Dictionary<string, string> data; //卡牌信息

    1 个引用
    public void Init(Dictionary<string, string> data)
    {
        this.data = data;
    }

    //记录当前元素在父级中的索引
    private int index;
    // 当鼠标进入该 UI 元素时触发
    0 个引用
```

2. AttackCardItem

负责处理攻击牌的效果，包括显示攻击指向目标的贝塞尔曲线及其箭头、展示攻击特效，根据攻击结果对应降低敌人生命值等，相关接口均封装在AttackCardItem类中。

```
Unity 脚本|0 个引用
public class AttackCardItem : CardItem, IPointerDownHandler
{
    1 个引用
    public override void OnBeginDrag(PointerEventData eventdata){}

    1 个引用
    public override void OnDrag(PointerEventData eventdata){}

    4 个引用
    public override void OnEndDrag(PointerEventData eventdata){}

    //按下
    0 个引用
    public void OnPointerDown(PointerEventData eventdata)
    {
        //播放声音
```

3. DefendCard

负责处理防御牌的效果，内容较为简单，播放对应特效并对应增加护盾值即可，内容均封装于DefendCard类中。

```
Unity 脚本|0 个引用
public class DefendCard : CardItem
{
    4 个引用
    public override void OnEndDrag(PointerEventData eventData)
    {
        if (TryUse() == true)
        {
            //使用效果数值
            int val = int.Parse(data["Arg0"]);
            //播放使用后的音效
            AudioManager.Instance.PlayEffect("Effect/healspell");
            //增加护盾数值
            FightManager.Instance.DefenseCount += val;
            //刷新护盾文本
            UIManager.Instance.GetUI<FightUI>("FightUI").UpdateDefense();
            Vector3 pos = Camera.main.transform.position;
```

4. AddCard

负责处理抽卡牌的效果，首先判断牌库中是否有卡可以抽取，若有则播放将其抽入手牌的特效，相关内容均封装于 AddCard 类中。

```
Unity 脚本|0 个引用
public class AddCard:CardItem
    4 个引用
{   public override void OnEndDrag(PointerEventData eventData)
    {
        if(TryUse() == true)
        {
            int val = int.Parse(data["Arg0"]);//抽卡数量

            //是否有卡抽
            if(FightCardManager.Instance.HasCard() == true)
            {
                UIManager.Instance.GetUI<FightUI>("FightUI").CreateCardItem(val);
                UIManager.Instance.GetUI<FightUI>("FightUI").UpdateCardItemPos();
                UIManager.Instance.GetUI<FightUI>("FightUI").UpdateCardCount();
```