1. 开始

开始制作一个职业的输出插件时,你的主要目的就是输出一个可以被 AEAssist 识别并加载的 Dll. 该 dll 需要位于和 AEAssistCNVersion 同目录下的 RotationPlugin 中.

具体位置是 RotationPlugin/你的插件文件夹名/你的插件.dll,并且 dll 同文件夹下需要有个 Loader.toml,里面定义了该加载哪个 dll.

具体的参考 RotationPlugin/AE/AE.dll 和 RotationPlugin/AE/Loader.toml

你有两种创建输出插件的方式,因为考虑到你可能没有工程源码,或者懒得打开工程,这里先列举其中一种:

利用你的 ide(主流是 rider/visual studio)创建一个.net6 的 class libary 工程.引用 AEAssistCNVersion 下的 CombatRoutine.dll 和 Common.dll.并设置这两个 dll 的 Copylocal 为反选状态(即不跟随输出).再创建一个 Loader.toml(设置复制到输出目录),并配置输出目录,确保最后输出的位置正确.

如果一切正常,当你编译这个工程时,在 RotationPlugin/你的插件文件夹名/这个路径下已经创建出来了类似的 dll.

2. 入门

AEAssist 在加载你的 dll 后,会扫描你的所有公开类型(即声明为 public 的类),寻找实现了 IRotationEntry 的类型,并实例化.

请确保 IRotationEntry 的实现类没有构造函数或者至少拥有无参构造函数.

```
7 usages  2 inheritors  AnotherEnd  1 exposing API
public interface IRotationEntry : IRotationUI
{
    /// <summary>
    ///     你的名字,长度必须>=2
    /// </summary>
    8 usages  2 implementations  AnotherEnd
    string AuthorName { get; }

    /// <summary>
    ///     针对的是哪个Job
    /// </summary>
    8 usages  2 implementations  AnotherEnd
    Jobs TargetJob { get; }

    /// <summary>
    ///     创建Rotation的入口,只会被调用一次
    /// </summary>
    /// <returns>插件的核心 <seealso cref="Rotation" /></returns>
    /// <param name="settingFolder">
    ///     配置文件的存放目录,包含你的代码存放的文件夹的名字
    ///     推荐配置文件采用<see cref="SystemJsonHelper" />来序列化
    /// </param>
    1 usage  2 implementations  AnotherEnd
    Rotation Build(string settingFolder);

    /// <summary>
    /// 语言切换时的处理,推荐语言的配置文件使用TomlHelper来处理,比json更方便
    /// </summary>
    /// <param name="languageType"></param>
    2 implementations  AnotherEnd
    void OnLanguageChanged(LanguageType languageType);
}
```

相关需要实现的信息我已经注释.
最核心的是 Build 方法返回的 Rotation 实例.
下图是作为示例的机工职业返回的 Rotation 实例

```
public Rotation Build(string settingFolder)
{
    MCHSettings.Build(settingFolder);
    return new Rotation(this, SlotResolvers)
        .AddOpener(new OpenerMCH70(), new OpenerMCH80(), new OpenerMCH90())
        .SetRotationEventHandler(new MchRotationEventHandler())
        .AddSettingUIs(new PhyDpsMCHSettingView())
        .AddSlotSequences(new Burst120Sequence(),new ShortBurst120Sequence());
}
```

Rotation 创建时的 SlotResolvers 是必须元素,每个 SlotResolver 代表着当决策使用 Gcd 技能或者 offGcd 时,该使用哪个.

如图,这就是一个最简单的 SlotResolver.

```
public class MCHGCDBaseCombo : ISlotResolver
{
    0+1 usages
    public SlotMode SlotMode { get; } = SlotMode.Gcd; 1
    0+1 usages  AnotherEnd
    public int Check()
    {
        return 0; 2
    }

    0+1 usages  AnotherEnd
    public void Build(Slot slot)
    {
        var spell = MCHSpellHelper.GetBaseGCDSpell(); 3
        if (spell == null)
            return ;
        slot.Add(spell); 4
    }
}
```

1: 这个 SlotResolver 是在计算该使用哪个 gcd 时参与计算的.

2: 当参与计算时,Check 只要返回了>=0 的值,就代表这个 SlotResolver 的判定条件通过,下面的 Build 方法会被插件调用. (**插件中所有名为 XXCheck 的方法,且返回 int 值的,都是>=0 代表通过,<0 代表不通过**)

3: 这里是在计算当前应该使用基础 3 连中的哪个技能.

4: 计算到的技能加入 slot 中. (Slot 将在后续说明)

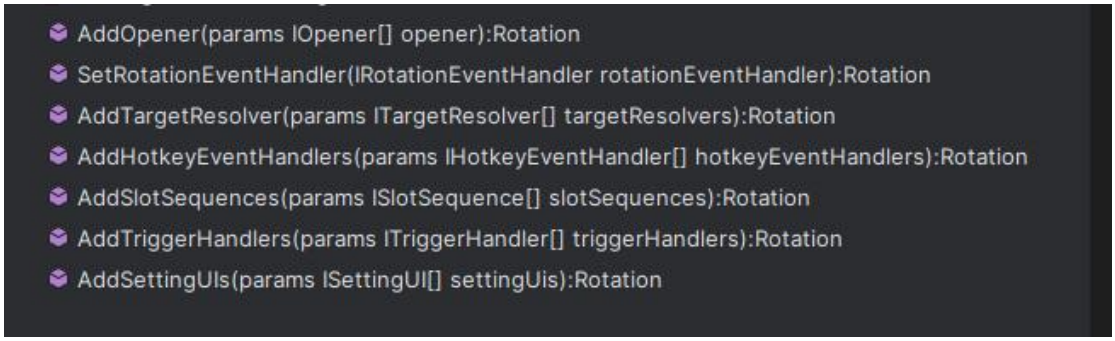
这样一来,当实现了这样一个 SlotResolver 之后,你将它传入 Rotation 的构造参数中,那么你的机工角色就会对着敌人持续进行基础的 gcd 连击了.

关于能力技:

```
public class MCHAbilityReassemble : ISlotResolver
{
    0+1 usages
    public SlotMode SlotMode { get; } = SlotMode.OffGcd;
    0+1 usages  AnotherEnd *
    public int Check()
    {
        //如果整备还没准备好,不打整备,充能技能至少有1层,IsReady才返回True
        if (!SpellsDefine.Reassemble.IsReady())
            return -1;

        // 判断一个gcd的70%时间 (一般是2500*0.7 = 1850ms左右)内,是否至少有一个强威力gcd冷却
        var time = Core.Get<IMemApiSpell>().GetGCDDuration() * 0.7f;
        if (!MCHSpellHelper.CheckReassmableGCD((int)time,out var strongGcd))
            return -2;

        return 0;
    }

    0+1 usages  AnotherEnd *
    public void Build(Slot slot)
    {
        var time = Core.Get<IMemApiSpell>().GetGCDDuration() * 0.7f;
        MCHSpellHelper.CheckReassmableGCD((int)time,out var strongGcd);
        // 找到对应的强威力gcd,接下来的技能使用就是整备+对应gcd
        slot.Add(SpellsDefine.Reassemble.GetSpell());
        slot.Add(strongGcd.GetSpell());
    }
}
```

3. 进阶

Rotation 提供了相当丰富的控制项,用来解决很多输出插件中的细节问题.目前拥有的有:



从上到下分别是:

1. 起手.代表着你的插件提供了多少个起手.每个起手有自己的条件(等级),比如机工 70 级,80 级,90 级的起手都不一样.
2. 设置一些回调处理,比如没有目标时的处理,某技能释放后的处理.

3. 目标选择器.如果你想让角色在战斗中自动选择/更换角色,可以实现这个.
4. 快捷键事件. (没完全测试)
5. 动态技能轴.和起手类似,只不过在实战中发挥作用,常见的做法是利用这个实现一个短期内非常灵活的轴. (机工 120s 的爆发轴,机工过热连)
6. 时间轴触发器的处理器.(没完全测试)
7. 添加设置的 UI.(可以通过设置界面打开的 UI,每个插件都会专门分一栏)

详细的使用参考请查看每个方法的注释和方法参数的注释,以及反编译 AE.dll(或者有源码直接查看源码).

4. 设计细节

1. 关于 Slot

```
/// <summary>
///     通用的Slot,决定接下来一段时间内连续释放的一批技能
///     可以是Gcd+能力技
///     可以是能力技+gcd
///     也可以是单个技能
///     还可以是一连串的组合
///     <seealso cref="Add(CombatRoutine.SlotAction)" />
/// </summary>
75 usages  AnotherEnd  7 exposing APIs
public class Slot
```

Slot 可以认为是单个技能/不可分割的技能连招/一连串的短技能队列. (比如机工的整备+钻头/锚/锯)

如无特殊需求,尽量不要让 Slot 覆盖到 3 个 gcd.因为 slot 中的每一个技能都是预设好的,并不像后续要提到的 SlotSequence 那样,每一步都是实时计算的.

预设好的流程容易产生意外.实时计算的拥有较高容错.后续在 SlotSequence 中会举例说明这个情况.

回到 Slot,通常使用 Slot.Add 方法来添加一个技能进入 Slot 中.添加的顺序决定着使用的顺序.

```
public bool Wait2NextGcd = false;
```

Slot.Wait2NextGcd 代表着该 slot 会阻塞技能使用的计算,直到下一次 gcd 可用前的 300ms.

```
/// 应该是最后一个设置的
/// </summary>
/// <param name="slotSequence"></param>
/// <param name="wait2nextGcd"></param>
10 usages  AnotherEnd  More...
public void AppendSequence(ISlotSequence slotSequence,bool wait2nextGcd)
{
    this.AppendedSequence = slotSequence;
    this.Wait2NextGcd = wait2nextGcd;
    _setEnd = true;
}
```

这个方法可以在 Slot 填充的所有技能使用完毕(包括 Wait2NextGcd)都计算完毕后,附加一个长技能队列.

比如,在机工的 120s 爆发轴中,就通过这个方法实现了野火+过热+过热连的技能队列:

```
static void Step2(Slot slot)
{
    slot.Add(SpellsDefine.Wildfire.GetSpell());
    slot.Add(SpellsDefine.Hypercharge.GetSpell());
    slot.AppendSequence(new HyperChargeSequence(),true);
}
```

2. 关于 ISlotSequence

SlotSequence 代表着一个较长的技能队列.它的底层也是通过实现多个步骤,每一步都填充一个 Slot 来实现的.

1. 如图所示,技能队列里填充的并不直接是 Slot,而是方法的引用(学名函数指针).
2. 动态计算,检测如果 1000ms 内有高威力 gcd(钻头/锚/锯)冷却完毕,就使用对应的高威力 gcd,否则使用基础 gcd.

填充的方法的引用保证队列在执行到第 0 步时,Step0 的逻辑就是根据那个时刻的状态来计算的.

```
1 usage  AnotherEnd
static void Step0(Slot slot)
{
    if (MCHSpellHelper.CheckReassmableGCD(1000, out var strongGcd))
    {
        slot.Add(strongGcd.GetSpell());
    }
    else
    {
        slot.Add(MCHSpellHelper.GetBaseGCDSpell());
    }


    // 等到下个Gcd开始计算再结束
    slot.Wait2NextGcd = true;
}

1 usage  AnotherEnd
static void Step1(Slot slot)
{
    // 上面不等到gcd开始再计算的话，这里的1000就不准
    if (MCHSpellHelper.CheckReassmableGCD(1000, out var strongGcd))
    {
```

如图所示,第 0 步结束时,设置了 Wait2NextGcd,代表着第 1 步是在 gcd 可用的前 300ms 才开始计算.

如果想让第 0 步的时间计算也是依赖于 gcd 可用的前 300ms,可以在 SlotSequence 的 StartCheck 中,加入检查:

```
    // 方便下面的时间计算,这里强制等到gcd开始
    if (!AI.Instance.CanUseGCD())
        return -5;
```