方案自定义

说明

这部分比较类似编程了，不过这里是基于 json 格式实现的，相比较而言，会比编程可读性要强一些，但是没有基础的人可能对一些概念比较模糊。但是如果对这方面极为感兴趣的话，可以试着理解后面的例子，尝试看看。

教程

1. 创建一个新的方案

首先我们从新建一个自定义方案开始。

不过开始这部分前我们先理解下 json 的概念，json 主要 key - val 的形式，即键值对，意味着可以通过 key 获取 val。相当于一个储物柜，你寄存了东西获得了一个条形码（key），可以通过这个条形码获取到你寄存的包裹（val 值）。

一个方案其实需要以 “key-方案” 的形式存在。

> KEY 支持中英文，不过一般情况下在计算机编程中，会尽量避免使用中文，原因除了美观外，还有中文编码不一定得到支持，所以有些程序要求不要中文路径也是这样的原因，哈哈，题外话了。

```
{
    "新方案 1 的 KEY":{
        // 这里写新方案 1 内容
    },
    "新方案 2 的 KEY":{
        // 这里写新方案 2
    }
}
```

上面展示了新建多个方案的写法，值（val 方案）用 `{}` 包裹表示对象（对象的意思是使用键值对的格式，还有数组，这个在后面介绍），多个方案直接以 `,` 隔开，最后一个值后面不要逗号。

Key 可以为任意字符串（用双引号包裹的叫做字符串），只要保证在同级里面是唯一的就行，一般出于规范会使用有意义的英文命名。

> 关于 json 的语法可以参考网上资料，不过还是会疏忽不容易发现，所以这里推荐使用微软的 VSCode 编辑器，支持对 json 语法的检验。

方案的属性

上文说到值也就是方案是一个对象，对象是拥有属性的，这些属性就是键值对的属性，和 LOL 英雄一样，见下面的例子。

```
{
    "狗熊":{
        "类型": "英雄",
        "名字": "不灭狂雷·沃利贝尔",
        "攻击力": 100,
        "防御力": 70,
        "移动速度": 340
    }
}
```

那么我们的方案也是拥有其必要的属性。

name

首先第一个属性是 name，该属性主要是用于UI界面方案名称的显示。

type

第二属性是 type，该属性主要用于方案的分组，可选的值有 `[0 | 1 | 2 | 3 | 4]`。

0 表示公用行动；1 表示刷野自动化行动；2 表示副本自动化行动；3 表示私有方案；4 表示地图跳转。

会被分配到指定的选框去，0 和 3 不会被分组。

schemes

第三个属性是 schemes，该属性则是方案的具体步骤，是一个 json 数组。

上面提到了对象，那么这里的则是数组，数组的话是不需要 key 的，可以理解他有一个默认的 key，就是他所在的位置的下标。数组其实就是数学上的序列啦。

props

其实还有一个属性是 props，这部分涉及要变量，放在后面介绍，除非你要设计一个复杂的方案，才会用到，比如执行一定次数的来回刷野。

实例一：打开地图

```
{
    "OPEN_MAP": {
    "name": "打开地图",
    "type": 2,
    "schemes": [
      { "type": 3, "position": [62, 278, 20, 1200] }
    ]
  }
}
```

在 `path.FB.json` 文件中加入上面的方案后可以在副本的方案库中找到。

OPEN_MAP 是这个方案的 KEY，可以为任意的值，哪怕你打一堆乱码也是有效的，不过不推荐了撒。

后面我们继续来看 schemes 部分是怎么编写的。

schemes：type 属性

schemes 是这个数组，数组的每一个元素都是一个对象。该对象有一个必要的属性就是 type。

type 的可以选值为 `[0 | 1 | 2 | 3 | 4]`，但是需要注意的是，这里的 type 和上面的 type 不是一个东西，这里的是表示行为的类型。比如 3 表示的鼠标事件。鼠标事件支持两个操作，分别是点击和拖拽。

schemes：鼠标事件

当指定 `type : 3` 时，这个元素就会被解析为鼠标事件，程序会读取这个元素的 `position` 属性或 `bias` 属性。

语法如下：

```
{ "type": 3, "position":[x, y, scope, delay], "bias":[x1,x2,x3,x4] }
```

点击事件

当存在 position 属性是，这个事件会被理解成点击事件，position 也是一个数组，它要求必须有四个元素。

```
"position": [x, y, scope, delay]
```

`x` 和 `y` 表示点击位置的坐标，关于坐标的值，可以利用按键精灵的抓抓获取。

`scope` 表示点击的范围，会以当前坐标为中心，在范围内随机点击，这样就拟人的点击（人不可能两次踏入同一条河流吧，欸嘿），一般而言小图标填 5，大图标可以再大点，懒得管统一填 5。

`delay` 表示延迟，也就是点击后会等待的时长，因为UI界面需要一定反应时间，所以这个值越大越安全，太短可能导致界面反应不过来，单位是毫秒，1000 则表示等待 1s 后再执行后续操作。

实例一中表示的就是点击坐标为 (62,278)，范围为 5 的位置，点击后等待 1.2 秒。

拖拽事件

当不存在 position 属性，只存在 bias 属性时，这个事件会被理解成拖拽事件，bias 也是一个数组，它要求必须有四个元素，但与 position 的意义不一样。

```
"bias": [x1, y1, x2, y2]
```

`x1` 和 `y1` 表示拖拽的起点坐标；

`x2` 和 `y2` 表示拖拽的终点坐标。

> position 和 bias 同时存在时只会生效一个。

实例二：从西方世界的时空裂缝前往月影森林

```
"YYSL_GO": {
    "name": "从西方世界的时空裂缝前往月影森林",
    "type": 2,
    "schemes": [
        { "type": 3, "position": [62, 278, 20, 1500] },
        { "type": 3, "position": [600, 204, 20, 400] },
        { "type": 3, "bias": [270, 30, 220, 530] },
        { "type": 3, "position": [588, 608, 5, 400] },
        { "type": 3, "position": [292, 782, 5, 400] }
    ]
}
```

schemes：移动

当指定`type : 1`时，这个元素就会被解析为移动。

> 移动顾名思义就是移动啦，其本质就是拖拽界面，所以，理论上可以使用鼠标事件的 bias 实现，但是因为 API 封装的问题，鼠标事件的拖拽是在 200 毫秒中完成的，而且移动也分为野外移动和城内移动。野外移动又可以分为移动向 BOSS、道中移动、移动向 FEAR 这样的三种情况，另外你也不希望每次输入那么多不知道啥意思的坐标吧。所以这部分是单独提供一个类型用于移动（又是一堆废话.jpg）

mode 属性指定移动模式，默认`"CITY"`

移动通过元素的`mode`属性来设置移动的模式，可选的值有`["CITY" | "WIDE" | "FEAR" | "BOSS"]`，默认为`"CITY"`。

- `CITY`：表示在城内移动，这一个模式下移动的精度最高，但是不会解决战斗，也就是说，这个模式只适合不会遇到怪物的情况。
- `WIDE`：表示在野外移动，这个模式下会通过后面提到的 plan 属性去执行相应的战斗策略去战斗。
- `FEAR`：表示对决 FEAR，基本上和`WIDE`模式类似。
- `BOSS`：表示对决 BOSS，这个模式下在移动到终点后会自动点击对战的确定键进入战斗，其他和`WIDE`模式类似。

direction 属性指定移动方向，必填

移动方向需要使用元素的`direction`属性指定，可选的值有`["UP" | "RIGHT" | "DOWN" | "LEFT"]`，必填。

distance 属性指定移动距离，必填

移动具体使用`distance`来指定，虽然用了“距离”的英文，但这里的距离其实是时间，比如`"distance": 1000`，表示的移动 1 秒钟。所以估摸移动距离自己可以默算下，必填。

这部分对遇怪进行了优化，会自动减去战斗的时间，基本上和不会遇怪用时相似。

> 但是时间计算对于计算机来说本来就是很难控制的，如果出现 200 毫秒左右的偏差，很容易导致跑过头或还没跑到，这个修补问题我会尝试在以后的更新中解决。
>
> 另外还有一个致命的问题是，游戏的卡顿及其容易导致跑图失败，这个问题一般在游戏刚刚启动，第一次跑图时容易出现。关于跑图失败已经有潜在方案，暂时不开放使用。
>
> 这里所有时间的单位都是毫秒，后文不在解释了。

plan 属性指定战斗策略，选填

非 `CITY` 模式下，需要指定该移动的遇怪时的战斗策略，可以不填，不填的话会读取配置中的默认战斗策略，所以，请保证默认策略适合当前的战斗。

这里可填写战斗策略的 KEY，称为 `PLAN_KEY`，请确保在 plan 中存在相应的方案。

实例三：移动至塞雷纳海岸渔场

下面的例子实现了跑到塞雷纳海岸的渔场，这里除了点击事件（type=3）和移动（type=1）以为，还有一个引用（type=0），这个后面会说明。

上面提到 mode 的默认值为 `CITY`，但是如果在野外居多的情况下，可以通过 props 设置默认移动模式，如实例第 4 行。

这样要求战斗策略库中有一个 COMMON 方案，请在 `plan.json` 中添加。

```
"YC_SLNHA": {
    "name": "塞雷纳海岸渔场",
    "“type": 0,
    "props": [{ "name": "MOVE_MODE", "default": "WIDE" }],
    "schemes": [
        { "type": 0, "go": "XF,XD,LD" },
        { "type": 1, "mode": "CITY", "direction": "LEFT", "distance": 7250 },
        { "type": 1, "plan": "COMMON", "direction": "LEFT", "distance": 6500 },
        { "type": 1, "plan": "COMMON", "direction": "UP" },
        { "type": 1, "plan": "COMMON", "direction": "LEFT", "distance": 4300 },
        { "type": 1, "plan": "COMMON", "direction": "DOWN" },
        { "type": 1, "plan": "COMMON", "direction": "LEFT", "distance": 1000 },
        { "type": 3, "position": [390, 580, 5, 600] }
    ]
}
```

> 在这个例子中，可以发现，当两个地图交界处，卡顿十分明显，该方案在第一次执行时，很容易出现无法到达指定位置的问题，在第二次执行时成功率提高。
>
> type=0 在下面就会介绍到。

schemes：引用

当指定`type : 0`时，这个元素就会被解析为引用。

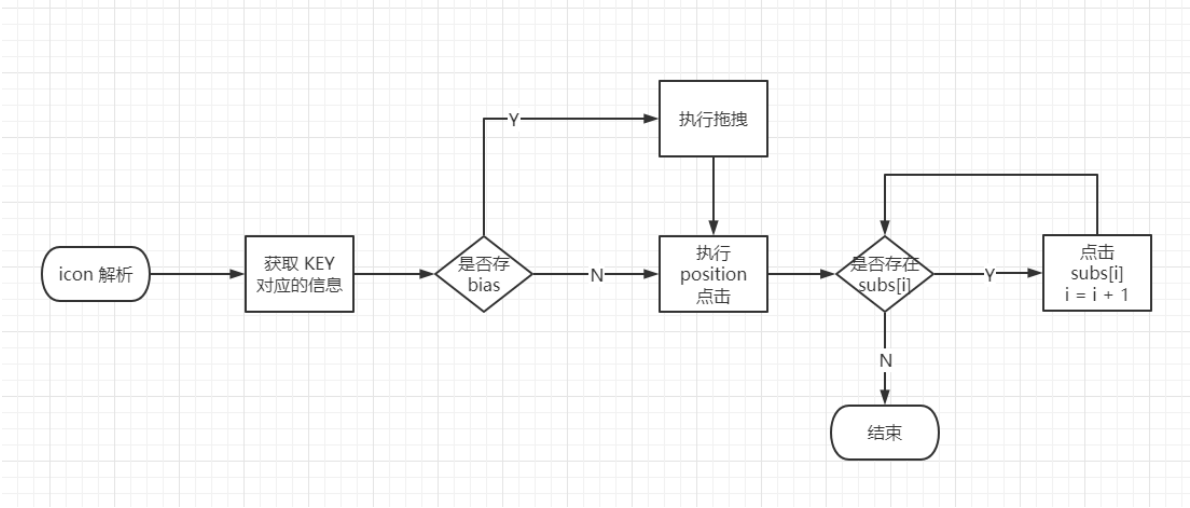
引用是一个重要的概念，基本库中会对大部分操作提供模板，这部分模板不需要通过复制黏贴来重新，直接通过 KEY 值来进行引用。

引用 icon

如 README 中所说，icon 是主要提供界面基本图标的点击功能。

```
{
  "CLOSE": {
    "name": "关闭",
    "type": 0,
    "position": [698, 1248, 5, 400]
  },
  "MAP": {
    "name": "地图",
    "type": 0,
    "position": [62, 278, 20, 1500]
  },
  "GO": {
    "name": "区域跳转-是",
    "type": 0,
    "position": [292, 782, 5, 400]
  },
  "RESTORE": {
    "name": "回复点",
    "type": 0,
    "position": [657, 431, 5, 600]
  },
  "FB_ZX_H": {
    "name": "主线困难副本",
    "type":3,
    "position": [464, 956, 5, 600]
  },
  "FB_ZX_VH": {
    "name": "主线非常困难副本",
    "type":3,
    "position": [373, 956, 5, 600]
  },
```

不过 icon 这部分在大多数情况下不会使用到，所以这部分不进行详细介绍，关于解析流程的话，直接看下面的流程图。



实例四：打开地图-2

```
{ "type": 0, "icon": "MAP" }
```

这样相比于填写坐标要来得更加简单和直观。

> icon 的信息可以在 API 列表文件中查询。

引用 go

这部分可能要使用的多点，因为地图跳转是比较常用的操作，使用这个引用可对做到一些智能化的操作。不过后续这部分也逐渐会由 path 替代，所以就简略介绍下。

go 有三部分组成 `[DF 东方|XF 西方],[XD 现代|WL 未来|GD 古代|???],[地点代号]`，三部分之间使用逗号隔开。

比如 `XF,XD,LD` 表示前往现代西方的林德。

另外还需要说明的一点时，出发点都是次元夹缝，因此，引用 go时，它会自动前往次元夹缝后再前往目的地。

实例五：前往当前大陆的次元夹缝

```
{ "type": 0, "go": "CUR,???,CYJF" }
```

这里有一个特殊的值 `CUR`，表示当前大陆。

引用 path

这个可以类比编程中的函数调用，path 其实就是一个函数，可以被另外一个 path 引用。这可以避免重复写一些相同的路径。比如副本困难和非常困难地图是一样的，这就可以只使用一套方案来实现两个模式的刷图。

引用也很简单，在 path 属性上填写对应的 `PATH_KEY` 就行了。

实例六：引用实例三

```
{ "type": 0, "path": "YC_SLNHA" }
```

> 基础的教程就到这里结束了，细心的同学发现 `"type" : 2` 并没有介绍，这个属于高级教程，感兴趣的可以继续看下去。

高级教程

这部分针对需要更复杂的方案时才会涉及，有编程基础的会理解起来更容易。

变量

上面提到 path 可以类比编程的函数，函数有一个重要组成部分就是函数的参数，函数可以根据参数执行不同的操作。

path 引用时支持参数的，也就是变量。

声明变量

一个自定义 path 的变量在其 `props` 属性中声明，声明支持两种格式，第一种是对象（键值对），第二种是字符串。

`props` 本身是一个数组，因此可以声明多个属性，比如 `"props" : ["arg1", "arg2"]` 表示声明了两个变量。

刚刚使用的是字符串声明，这样声明的两个参数是必须填写的参数，因为它没有默认值，如果需要赋予默认值的话，必须以对象像是声明。

比如 `"props": [{"name": "arg1", "default": 1}]` 表示 arg1 参数在未填写时默认为 1。

变量使用

声明了一个变量后，然后就是使用变量。

变量的使用是比较简单的，比如在移动中，将战斗方案设置为变量，这样就可以由调用者决定要不要使用他自定义的方案。

变量使用需要使用 `$`（美元符号）符号包裹。

实例七：改写实例三

在实例三中，野外移动使用的都是 `COMMON` 这个方案。而下面的实例中，使用的是 plan1 变量指定的方案。

这里默认值使用的是 CUR_WIDE_PLAN，需要说明的地方是，配置中提供了三个默认方案，分别是道中、FEAR、和 BOSS，如果希望变量的默认值为三个默认方案的其中一个，可以分别使用 `CUR_WIDE_PLAN`、`CUR_FEAR_PLAN`、`CUR_BOSS_PLAN`。

```
"YC_SLNHA": {
    "name": "塞雷纳海岸渔场",
    ""type": 0,
    "props": [
        { "name": "MOVE_MODE", "default": "WIDE" },
        { "name": "plan1", "default": "CUR_WIDE_PLAN" },
    ],
    "schemes": [
        { "type": 0, "go": "XF,XD,LD" },
        { "type": 1, "mode": "CITY", "direction": "LEFT", "distance": 7250 },
        { "type": 1, "plan": "$plan1$", "direction": "LEFT", "distance": 6500 },
        { "type": 1, "plan": "$plan1$", "direction": "UP" },
        { "type": 1, "plan": "$plan1$", "direction": "LEFT", "distance": 4300 },
        { "type": 1, "plan": "$plan1$", "direction": "DOWN" },
        { "type": 1, "plan": "$plan1$", "direction": "LEFT", "distance": 1000 },
        { "type": 3, "position": [390, 580, 5, 600] }
    ]
}
```

实例八：在 path 引用中传入变量

实例七展示的是如何创建并使用变量，实例八则是说明如何在 path 引用时，将变量传入。

如果说是引用实例七，在引用 path 中已经介绍过了，传参其实很简单，在这个引用中的属性中写值就行了就可以了。

上面的变量叫做 `plan1`，那么我们就使用 `plan1` 的属性名即可。

```
{ "type": 0, "path": "YC_SLNHA", "plan1": "COMMON"}
```

这里传入的方案任然是 COMMON，但是当希望使用另外一个方案时，只需要修改这里的参数，不需要对引用的方案进行修改。

condition 表达式

condition 表达式支持逻辑判断、计算、图片检验。

逻辑判断

`logic:`为前缀的是逻辑表达式，程序会根据表达式的结果执行后续的操作。

逻辑表达式目前仅支持基本的运算符（`+`、`-`、`*`、`/`），基本逻辑判断（`>`、`<`、`>=`、`<=`、`==`、`!=`），优先运算（`(`、`)`）。

比如`logic:$val$ > 10`表示判断`val`变量是否大于 10。

计算

`calc:`为前缀的是计算表达式，程序会根据表达式的结果执行后续的操作。

计算表达式目前仅支持基本的运算符（`+`、`-`、`*`、`/`），优先运算（`(`、`)`）。

比如`calc:$val$ /(10 - 8)`表示计算变量`$val$`除以 2 的值。

图片是否存在

`img:`为前缀的是图片检验存在的功能。存在返回 ture，不存在返回 false，程序会根据表达式的结果执行后续的操作。

比如`img:['menu.png', [0,0,200,200], 0.8]"`表示检验 menu.png 在矩形范围内是否存在，相似度匹配为 0.8。

> 如果需要在自定义范围内检验自己的图片的话，第一个参数用绝对路径，即`/`开头。
>
> 如`img:['/storage/emulated/0/photo/menu.png', [0,0,200,200], 0.8]`使用自定义路径下的 menue 图片在矩形范围内是否存在，相似度为 0.8。

相似度后面还有一个参数 delay，表示延迟匹配，如`img:['menu.png', [0,0,200,200], 0.8, 3000]`表示 3 秒内会不断匹配，直至匹配成功或超时。这个适合在跳转后 UI 还没刷新是使用。

图片是否不存在

`no-img:`为前缀的是图片检验不存在的功能。不存在返回 ture，存在返回 false，程序会根据表达式的结果执行后续的操作。

其他与`img`类似。

结果执行

当指定`"type" : 2`时，这个元素会被理解为结果执行。

结果执行依赖于刚刚介绍的 condition 表达式。当 condition 表达式为 true 或不为空时，会执行`consequence`约定的结果。

> 因为 calc 的计算结果一定会触发结果处理，所以一般不会在 condition 中使用 calc。

语法如下：

```
{
    "condition": "condition 表达式",
    "consequence": "结果类型，可选值[EXIT | CONTINUE | ASSIGN]，必填",
    "assign": [{"count": 0}]
}
```

consequence：EXIT

当执行结果为 true 时会退出，为 false 时会继续运行。

consequence：CONTINUE

当执行结果为 true 时会继续运行，为 false 时会退出。

consequence：ASSIGN

当执行结果为 true 时会执行赋值操作，为 false 时不会赋值。

赋值操作由 assign 属性决定，assign 属性的值支持使用 condition 表达式。

赋值操作可以理解为创建全局变量（一次执行中，只要唯一的一个）。

循环

循环可以重复执行某一个 scheme（步骤，即 schemes 的元素），也就是说，所有 scheme 都有 for 属性。

> condition 属性可能在后续也对所有 scheme 进行支持，届时，for 对象的 condition 不再使用，目前没有这个需求。

通过 for 属性指定，for 属性可以是对象也可以是数值。

数值表示循环次数：

```
{"for": 10}
```

对象支持使用 condition 来进行约束，下面的表示继续执行的条件为遇怪次数小于 count 变量，同时来回走动次数小于 10。

```
{ "for": { "condition": ["logic:$MONSTER_COUNT$ < $count$", "logic:$ROUND_COUNT$ < 10"] } }
```

> 对象的话也可以使用 count 属性来之间指定循环次数（condition 任然生效）。

实例九：实现来回刷野

这个实例要更加复杂，主要需要实现下面所述的功能：

1. 来回走动刷野。
2. 可以自定义遇怪次数。
3. 来回走动 10 次没有遇怪则认为出 bug 了停止移动。

```
"_SY": {
    "name": "来回刷野-一次来回",
    "type": 3,
    "props": [{ "name": "MOVE_MODE", "default": "WIDE" }, "plan"],
    "schemes": [
```

```
        {
            "type": 1, "plan": "$plan$", "direction": "LEFT", "distance": 1200,
"wait_stop": false,
            "onBattle": { "ROUND_COUNT": 0, "MONSTER_COUNT":
"calc:$MONSTER_COUNT$ + 1" }
        },
        {
            "type": 1, "plan": "$plan$", "direction": "RIGHT", "distance": 1200,
"wait_stop": false,
            "onBattle": { "ROUND_COUNT": 0, "MONSTER_COUNT":
"calc:$MONSTER_COUNT$ + 1" }
        },
        { "type": 2, "consequence": "ASSIGN", "assign": { "ROUND_COUNT":
"calc:$ROUND_COUNT$ + 1" } }
    ]
},
"SY": {
    "name": "来回刷野",
    "type": 0,
    "props": [
        { "name": "plan", "default": "CUR_WIDE_PLAN" },
        { "name": "count", "default": 5 }
    ],
    "schemes": [
        { "type": 2, "consequence": "ASSIGN", "assign": { "MONSTER_COUNT": 0 }
},
        { "type": 2, "consequence": "ASSIGN", "assign": { "ROUND_COUNT": 0 } },
        {
            "type": 0, "path": "_SY", "plan": "$plan$",
            "for": { "condition": ["logic:$MONSTER_COUNT$ < $count$",
"logic:$ROUND_COUNT$ < 10"] }
        },
        { "type": 2, "consequence": "ASSIGN", "assign": { "MONSTER_COUNT": 0 }
},
        { "type": 2, "consequence": "ASSIGN", "assign": { "ROUND_COUNT": 0 } }
    ]
}
```

> 这里可以看到一个个特殊的参数，onBattle 赋值事件：这个事件会在遇怪时执行，实例中表示会遇怪后会对遇怪次数进行 + 1操作。

这个通过 SY 进行引用，该方案接受刷野次数以及战斗方案，战斗方案默认使用道中战斗策略。

首先创建两个全局变量，`MONSTER_COUNT` 遇怪次数、`ROUND_COUNT` 来回次数。

然后 SY 引用 _SY ，并传入战斗方案。循环条件为遇怪次数小于 count 变量，同时来回走动次数小于 10。

_SY 中，首先左移，遇怪时对 MONSTER_COUNT + 1，ROUND_COUNT 重新设值为 0。一次来回后设置 ROUND_COUNT + 1。

最后重新将两个全局变量设置回 0。

结束语

首先感谢看到最后来理解我的逻辑。关于 path 自定义部分应该可以介绍的都介绍完了，我也很期待大家能玩出什么花样来，当然如果觉得这个实在太难了也没关系，你们也可以基于已经写好的模板进行修改。

另外愿意分享自己的方案的同学，表示热烈的欢迎和感谢，感谢你们的努力让作者少测一张图。（幸福.jpg）

虽然这个方案架构经过一次升级（当时大概一半的代码重写了，不过功能更强大了）。

关于更新的事情，emmmm，写这个的时候还没发布呢，不知道你们的反响是啥，目前主要任务还是扩充基本库，新功能还在路上。

最后是自定义方案的建议：

1. 遇怪模式下移动不稳定性大，特别是当你电脑运行多个应用时（用术语讲就是 CPU 资源竞争大时），时间的计算精度缺失越大，所以跑图前先清一波道中最好（所以需要一个无消耗或者时低消耗道中方案，有玛娜就行了.jpg），如果是对战 FEAR 后，可以通过移动至墙角来进行重新定位。
2. 我尽量提供丰富的 API，比如上面的刷野实例其实就是这个脚本刷野的实现，其他关于打开宝箱、对话、等待跳转结束等功能都已经提供，这部分 API 可以参考 PATH API 列表查询（有人愿意在基本库中添加自己封装的泛用方案也是很欢迎的）。

最后对于 PATH 逻辑有更高的兴趣的同学,可以去看看群里的关于 PATH 解析的流程图，应该可以帮助你更好的理解这块内容。