HOI4DEV 钢4MOD开发工具

更新日期：2024.12.10

使用教程

HOI4DEV工作理念

HOI4DEV旨在通过Python程序自动化钢4 mod开发过程，基本原理是：将钢4的脚本语言（Clausewitz scripting language, CCL）翻译为通用的JSON格式，从而允许使用Python进行批量编辑和生成。

与此同时，HOI4DEV也添加了对于DDS图像处理、GFX代码生成、地图编辑、国策树自动摆位、自动本土化等专属HOI4DEV的功能处理。

更进一步，对于新建国家、军队、科技、装备、领袖等各个项目，HOI4DEV实现了许多模板，可以快速通过简单的文件夹格式构建钢4 MOD。

HOI4DEV中的mod开发采用“编译”模式：我们推荐整个mod的所有文件全部动态生成，而开发者只对mod的工程文件进行修改。工程文件主要以JSON与资源文件夹的形式存在。

例如，一个通常的工程中的资源文件夹是这样的：

```
some_resource_folder/
├── info.json
├── default.png
└── locs.txt
```

其中，`info.json` 文件一般将会最终被编译为转为钢4代码，`default.png` 转为GFX图标，`locs.txt` 转为本地化。
例如，如果上述资源文件夹代表一个角色，`info.json` 将是关于角色的特质等代码，`default.png` 为角色肖像，`locs.txt` 包含角色的多语言名称和描述。
如果上述资源文件夹代表一个民族精神，`info.json` 将是精神的修正效果，`default.png` 为精神图标，`locs.txt` 包含精神的多语言名称和描述。
再比如上述资源文件夹代表一个军事装备，`info.json` 将是装备的数值，`default.png` 为装备在生产界面的侧视图，`locs.txt` 包含装备的多语言名称、装备分类、描述等。

从0开发一个完整的mod，其工程文件会近似于：

```
resources/
├── characters/ # 角色
```

```
|   ├── character_1/
|   |   └── ...
|   ├── character_2/
|   |   └── ...
|   └── ...
├── decisions/ # 决议
|   └── ...
├── doctrines/ # 学说
|   └── ...
├── equipments/ # 装备
|   └── ...
├── events/ # 事件
|   └── ...
└── ...
```

同时，这个结构是自由的，可以按照偏好将相关的内容放在一起（例如将角色资源文件夹与所属国家放在一起等）。相比于钢4的组织方式（同一个实体的代码、图标、本地化等属性分离），HOI4DEV工程文件更加易于就近编辑。

在编辑工程文件后，可以用HOI4DEV进行编译来生成一个钢4 mod。

基础功能

钢4脚本CCL语言与JSON格式互相转化

使用 `hoi4dev convert -i <输入文件> -o <输出文件>` 来自动、智能地从一种格式转为另一种格式。

格式识别取决于文件的后缀名，`.txt`，`.gui` 与 `.gfx` 文件会被视为CCL语言，而 `.json` 文件会被视为JSON格式。

例如，从一个CCL语言转为JSON格式，在Python代码内，这等价于：

```
input_file = "<输入文件>"
output_file = "<输出文件>"

ccl_data = ReadTxt(input_file)
json_data = CCL2Dict(ccl_data)

CreateFile(output_file)
SaveJson(json_data, output_file, indent=4)
```

而从JSON格式转为CCL语言则等价于：

```
input_file = "<输入文件>"
output_file = "<输出文件>"

json_data = LoadJson(input_file)
ccl_data = Dict2CCL(json_data)

CreateFile(output_file)
SaveTxt(ccl_data, output_file)
```

(拓展阅读) 钢4脚本CCL语言、JSON格式转化标准

注意到P社的CCL语言语法是一坨屎山，我们希望将它与JSON格式语法对齐。

JSON的语法格式非常简单，只包含列表和字典。一个列表通常使用中括号 `[]` 表示，例如 `["A", "B", "C", "D", ...]`；而一个字典键值总是以 `"<key>": "value"` 的形式，包括在大括号 `{}` 中；无论是列表还是字典中，除最后一个元素外，其他元素用之间必须使用逗号 `,` 进行分隔。

相比之下，CCL语法主要以 `<key> = <value>` 的形式储存键值，并且无论是列表还是字典都是用大括号 `{}` 存储，也没有明确的分隔符号；在众神的黄昏DLC更新后，甚至出现了列表与字典混用的语法。同时，CCL语法中的值并不被引号包括，也增加了识别难度。

因此我们主要基于JSON的字典来转化CCL语法。下面是一个角��的CCL代码例：

```
characters = {
    CHARACTER_STARLIGHT_GLIMMER = {
        name = CHARACTER_STARLIGHT_GLIMMER_NAME
        portraits = {
            civilian = {
                large = "gfx/leaders/CHARACTER_STARLIGHT_GLIMMER.dds"
                small = "gfx/leaders/CHARACTER_STARLIGHT_GLIMMER_small.dds"
            }
            army = {
                large = "gfx/leaders/CHARACTER_STARLIGHT_GLIMMER_army.dds"
                small = "gfx/leaders/CHARACTER_STARLIGHT_GLIMMER_army_small.dds"
            }
            navy = {
                large = "gfx/leaders/CHARACTER_STARLIGHT_GLIMMER_navy.dds"
                small = "gfx/leaders/CHARACTER_STARLIGHT_GLIMMER_navy_small.dds"
            }
        }
        gender = female
        country_leader = {
            desc = CHARACTER_STARLIGHT_GLIMMER_DESC
```

```
            traits = {
                TRAIT_STARLIGHT_GLIMMER_RESISTANCE_LEADER
                TRAIT_STARLIGHT_GLIMMER_FORMER_PRINCIPLE
            }
            ideology = harmonicism_republic
        }
    }
}
```

HOI4DEV转换后，对照的JSON格式：

```
{
    "characters": {
        "CHARACTER_STARLIGHT_GLIMMER": {
            "name": "CHARACTER_STARLIGHT_GLIMMER_NAME",
            "portraits": {
                "civilian": {
                    "large": "gfx/leaders/CHARACTER_STARLIGHT_GLIMMER.dds",
                    "small": "gfx/leaders/CHARACTER_STARLIGHT_GLIMMER_small.dds"
                },
                "army": {
                    "large": "gfx/leaders/CHARACTER_STARLIGHT_GLIMMER_army.dds",
                    "small":
"gfx/leaders/CHARACTER_STARLIGHT_GLIMMER_army_small.dds"
                },
                "navy": {
                    "large": "gfx/leaders/CHARACTER_STARLIGHT_GLIMMER_navy.dds",
                    "small":
"gfx/leaders/CHARACTER_STARLIGHT_GLIMMER_navy_small.dds"
                }
            },
            "gender": "female",
            "country_leader": {
                "desc": "CHARACTER_STARLIGHT_GLIMMER_DESC",
                "traits": [
                    "TRAIT_STARLIGHT_GLIMMER_RESISTANCE_LEADER",
                    "TRAIT_STARLIGHT_GLIMMER_FORMER_PRINCIPLE"
                ],
                "ideology": "harmonicism_republic"
            }
        }
    }
}
```

而当我们引入HOI4DEV的角色模板后，MOD开发者实际上需要编写的代码就只有：

```
{
    "gender": "female",
    "country_leader": {
        "ideology": "harmonicism_republic",
        "traits": [
            "TRAIT_STARLIGHT_GLIMMER_RESISTANCE_LEADER",
            "TRAIT_STARLIGHT_GLIMMER_FORMER_PRINCIPLE"
        ]
    }
}
```

特别地，由于JSON格式标准相对简单，在与CCL的转化中会出现表达能力不足的情况，我们有如下规则：

1. 所有CCL的等式转化为JSON字典的一对键值。例如：`research_speed_factor = 0.5` 会成为 `{"research_speed_factor": 0.5}` 。
2. 所有CCL的不等式，会转化为值为空的键。例如 `value > 3` 会成为 `{"value > 3": null}` 。
3. 所有CCL中的 `yes` 与 `no` 会转换为真、假的布尔值。例如 `is_ai = yes` 会成为 `{"is_ai": true}` 。
4. 所有CCL中的注释 `# This is a comment` 会被忽略。如果想要在JSON中添加注释，请使用 `{"// This is a comment": null}` 。
5. JSON不允许存在重复的键值，因此重复的键值会以 `__D<编号>` 区分。例如 `tag = USA` ， `tag = GER` ， `tag = SOV` 一起作为判断条件时，会成为 `{"tag": "USA", "tag__D1": "GER", "tag__D2": "SOV"}` ，编号顺序取决于其在CCL代码中的顺序，从前向后。
6. 所有CCL中的列表会被转为中括号。例如 `add_ideas = { my_idea_1 my_idea_2 }` 会成为 `{"add_ideas": ["my_idea_1", "my_idea_2"]}` 。因为无法区分，所以一个空的字典也会被视作列表。在众神的黄昏DLC更新后，列表与字典可能混合使用，此时列表中的元素都会转化为值为空的键。

遵循这些规��的JSON文件，也可以被方便地转换回CCL语言。如今，您可以使用 `hoi4dev convert -i <输入文件> -o <输出文件>` 来自动、智能地从一种格式转为另一种格式。

TXT格式的本地化

使用 `hoi4dev loc2json -i <输入文件> -o <输出文件> -s <域>` 可以将TXT或YML，转换为JSON。格式识别取决于文件的后缀名， `.txt` 文件会被视为HOI4DEV定义的本地化文本文件，而 `.yml` 文件会被视为钢4的本地化格式。 `<域>` 只对 `.txt` 文件生效。

在Python代码内，这等价于：

```
input_file = "<输入文件>"
output_file = "<输出文件>"

json_data = ReadTxtLocs(input_file, scope="<域>") if input.endswith('.txt') else ReadYmlLocs(path=input)

CreateFile(output_file)
SaveJson(json_data, output_file, indent=4)
```

与此同时，用 `hoi4dev json2loc -i <输入文件> -o <输出文件>` 可以将JSON文件转换为TXT文本文件。由于转为YML后会是多个文件放置在mod的 `localisation/` 目录中不同地方，需要在mod编译时使用。

在Python代码内，这等价于：

```
input_file = "<输入文件>"
output_file = "<输出文件>"

json_data = LoadJson(input_file)

CreateFile(output_file)
SaveTxtLocs(json_data, output_file)
```

(拓展阅读) 钢4本地化YML格式、TXT格式、JSON格式转化标准

注意到P社的本地化是一坨屎山，为了方便mod开发者进行本地化，而不是一边开发mod一边跑去翻YML文件，HOI4DEV定义了一个TXT纯文本的本地化格式。

本地化文件是一个纯文本文件。通过 `[语言.键值名]` 的方式进行本地化。

- 支持多行文本。开头与结尾的空行与空格会被自动清除。
- 支持不同语言混合，方便进行翻译。
- 支持一个通用的替换域：带有 `@` 的键将用当前作用域名称替换 `@` ，并在名称后加上 `_` （除非以 `@` 结尾）。这意味着许多本地化不需要再写冗长的前缀。例如，一个角色的姓名和描述可以直接用 `@NAME` 和 `@DESC` 表示。

例如，这是按照HOI4DEV格式的一个本地化文件例：

```
[en.key01]
test1
test2
[zh.@key01]

中文测例1
中文测例2
[zh.key01]
中文测例3
中文测例4
```

例如，在 `CHARACTER_ABCDEFG` 替换域下，它将被解析为JSON：

```
{
    "key01": {
        "en": "test1\ntest2",
        "zh": "中文测例3\n中文测例4"
    },
    "CHARACTER_ABCDEFG_key01": {
        "zh": "中文测例1\n中文测例2"
    }
}
```

在钢4的YML格式中，上述本地化等价于两个文件：

```
l_english:
  key01:0 "test1\ntest2"
```

与

```
l_simp_chinese:
  key01:0 "中文测例3\n中文测例4"
  CHARACTER_ABCDEFG_key01:0 "中文测例1\n中文测例2"
```

如今，您可以使用 `hoi4dev loc2json -i <输入文件> -o <输出文件>` 和 `hoi4dev json2loc -i <输入文件> -o <输出文件>` 进行TXT与JSON的转换。由于转为YML后会是多个文件放置在mod的 `localisation/` 目录中不同地方，需要在mod编译时使用。

图像编辑

图像编辑功能允许用户对图像进行缩放和格式转换，支持多种图像格式，包括 `.png` 、 `.jpg` 、 `.tga` 和P社莫名奇妙就特别喜爱的 `.dds` 。

使用以下命令进行图像的最基本缩放、裁剪编辑和格式转换：

```
hoi4dev imgedit -i <输入文件> -o <输出文件> -r <缩放比例> -w <宽度> -h <高度> -b <行为> -f <翻转TGA> -c <压缩>
```

- `-i` 或 `--input` ：输入文件路径。
- `-o` 或 `--output` ：输出文件路径。
- `-r` 或 `--ratio` ：缩放比例，默认为1。
- `-w` 或 `--width` ：目标宽度，默认为-1（保持原始宽度）。
- `-h` 或 `--height` ：目标高度，默认为-1（保持原始高度）。
- `-b` 或 `--behavior` ：缩放行为，默认为 `max` ，可选值为 `max` 或 `min` 。
- `-f` 或 `--flip_tga` ：保存 `.tga` 文件时是否翻转图像，默认为 `False` 表示不翻转（钢4中的国旗都为上下反转的 `.tga` 文件）。
- `-c` 或 `--compression` ：保存编辑后图像时的压缩格式，默认为 `dxt3` 。仅对 `.dds` 有效。

请注意此命令的行为：

1. 图像的比例将始终保持，不会出现拉伸、扭曲。
2. 如果给定比例 `r` ，图像将始终按比例缩放，然后裁剪 / 扩展（填补透明背景）。 `w` 和 `h` 应该要么都不设置，要么都设置，否则会导致错误。
3. 如果没有给定比例，同时既没有给定 `w` 也没有给定 `h` ，图像将不会被修改。
4. 否则，如果没有给定比例，同时只给定了 `w` 或 `h` 中的一个，图像将缩放到给定的大小。
5. 如果同时给定了 `w` 和 `h` ，如果行为为 'max'，图像将缩放直到两个边都大于给定大小，然后裁剪到给定大小；如果行为为 'min'，图像将缩放直到只有一个边等于给定大小，然后扩展到给定大小。
   例如：

- (5, 3) 大小的图像按 `r=2` 缩放将得到 (10, 6)。
- (5, 3) 大小的图像按 `r=2，w=30，h=30` 缩放将得到 (10, 6) 然后扩展到 (30, 30)。
- (5, 3) 大小的图像按 `w=30` 缩放将得到 (30, 18)。
- (5, 3) 大小的图像按 `h=30` 缩放将得到 (50, 30)。
- (5, 3) 大小的图像按 `w=30，h=30，behavior='max'` 缩放将得到 (50, 30) 然后裁剪到 (30, 30)。
- (5, 3) 大小的图像按 `w=30，h=30，behavior='min'` 缩放将得到 (30, 18) 然后扩展到 (30, 30)。

在实际使用中，例如：

```
hoi4dev imgedit -i <输入文件> -o <输出文件> -r 2 -w 800 -h 600 -b max -c dxt3
```

会将输入图像放大两倍后，裁剪为800x600（如果小于这个数值会填补透明背景），且如果保存为 `.dds` 文件会进行压缩。

在Python代码内，这等价于：

```
input_file = "<输入文件>"
output_file = "<输出文件>"
r = 2  # 缩放比例
w = 800  # 目标宽度
h = 600  # 目标高度
behavior = 'max'  # 缩放行为
flip_tga = False  # 不翻转TGA
compression = 'dxt3'  # 压缩格式

img = ImageLoad(input_file)
edited_img = ImageZoom(img, r=r, w=w, h=h, flip_tga=flip_tga, behavior=behavior)
ImageSave(edited_img, output_file, compression=compression)
```

但值得注意的是，在Python代码内， `flip_tga` 默认为真，即默认翻转所有 `tga` 图片文件。

进阶功能

HOI4DEV最大的优势并不是已经提供的现成功能（否则并不是必须将CCL语言多此一举转为JSON语言），而是允许自动化的操作。因此，想要使HOI4DEV工具在mod开发中发挥更重要的作用，我们推荐您了解接触Python编程。

对于HOI4DEV提供的完整的Python支持，可以查看本仓库中 `doc/hoi4dev.html` （google格式自动生成的**pdoc文档**），其中包含所有函数的英文说明。

使用Python方便地处理JSON语言，是HOI4DEV最重要的功能，也是HOI4DEV的扩展性。例如，如果我们希望修改地图，为地图上的每一个地块添加一个民用工厂，在nudge编辑器中添加或者手工添加都需要很长时间，而用HOI4DEV将所有states文件转为JSON，编辑后再转回钢4代码就非常容易。

从0开始创建一个新的mod

```
from hoi4dev import *

root = CreateMod(
    name = "WTF",
    title = "WhatTheFuck",
    version = "v0.1.0",
    hoi4_version = "1.15.1",
```

```
    tags = ["Alternative History", "Gameplay", "Map", "Technologies", "Sound", "National Focuses", "Events", "Ideologies"],
    replace_paths = [
        # 添加替换路径
    ]
)
set_current_mod(root) # 这样我们就可以在HOI4_MODS_PATH下面新建一个mod文件夹了：`<HOI4_MODS_PATH>/WTF/`

InitMod() # 多数文件会从钢4中复制而来

CompileMod() # 编译钢4代码，之后每次修改工程以后就可以使用CompileMod()来预览结果。不过，随着工程越来越大，编译的时间会增长（一般对于中等模组会需要数分钟）。
```

在 `<HOI4_MODS_PATH>/WTF/` 中，我们可以看到一个编译完的mod文件空壳：

```
<HOI4_MODS_PATH>/
    WTF/
    |   ├── common/
    |   ├── data/
    |   ├── descriptor.mod
    |   ├── events/
    |   ├── gfx/
    |   ├── history/
    |   ├── hoi4dev_settings/
    |   ├── interface/
    |   ├── music/
    |   ├── sound/
    |   └── tutorial/
    └── WTF.mod
```

唯一多出的是 `data/` 文件夹，里面将存储所有的 `.json` 文件（我们并不建议直接在 `data/` 中修改文件，而是单独新建一个工程文件夹，然后每次编译前将工程文件复制到 `data/` 中）。

注意到这里包含了许多空路径、和默认复制的路径，开发者可以浏览每个目录，删除自己的mod并不需要修改的文件夹。

在下一部分中，我们将展示一个完整的mod开发实例工程：PIHC（The Pony In The High Castle MOD），高堡奇驹模组。

GitHub链接（暂时为私密，需要联系作者申请访问；我们将很快开放）。

(拓展阅读) 示例#1：地块数据批量编辑

钢4 mod中，地块数据都在 `history/states/*.txt` 中，我们可以写如下Python代码来为每个地块添加一个民用工厂：

```
from hoi4dev import *
import numpy as np # 用于随机
np.random.seed(42)

for state_file in [f for f in ListFiles('history/states/') if f.endswith(".txt")]: # 查看history/states/中的所有地块描述文件
    ccl_data = ReadTxt(pjoin("history/states/", state_file)) # 读取当前地块的钢4代码
    json_data = CCL2Dict(ccl_data) # 将这个地块的代码转为JSON格式，可以开始随意编辑了

    if 'buildings' not in json_data['state']['history']: # 如果这个地块没有建筑代码，那么创建一个空的建筑代码，避免出错
        json_data['state']['history']['buildings'] = {}
    if 'industrial_complex' not in json_data['state']['history']['buildings']: # 如果这个地块没有民用工厂，那么添加1个新的民用工厂建筑代码，否则直接将民用工厂数量+1
        json_data['state']['history']['buildings']['industrial_complex'] = 1
    else:
        json_data['state']['history']['buildings']['industrial_complex'] += 1

    SaveTxt(Dict2CCL(json_data), state_file) # 保存修改
```

举一反三：

```
from hoi4dev import *
import numpy as np # 用于随机
np.random.seed(42)

for state_file in [f for f in ListFiles('history/states/') if f.endswith(".txt")]: # 查看history/states/中的所有地块描述文件
    ccl_data = ReadTxt(pjoin("history/states/", state_file)) # 读取当前地块的钢4代码
    json_data = CCL2Dict(ccl_data) # 将这个地块的代码转为JSON格式，可以开始随意编辑了

    # 如果想要将这个地块的人口减半：
    json_data['state']['manpower'] = int(data['state']['manpower']) // 2

    # 如果想要将这个地块交给苏联：
    json_data['state']['owner'] = 'SOV'

    # 如果想要给20%的地块增加三份铝：
    if np.random.rand() <= 0.2: # 20%概率
```

```
        if 'aluminium' not in json_data['state']['resources']:
            json_data['state']['resources']['aluminium'] = 3
        else:
            json_data['state']['resources']['aluminium'] += 3

    # 如果想要重新分配所有地块的胜利点，给所有普通城市地块随机加一个省份胜利点为10，清除其他胜利点
    if json_data['state']['state_category'] == 'city':
        provs = json_data['state']['provinces'] # 获得这个地块的所有省份
        random_prov = np.random.choice(provs)
        json_data['state']['history']['victory_points'] = [random_prov, 10] # 清除原来的胜利点，将这个省份设为10胜利点（即victory_points = { random_prov 10 }）

    SaveTxt(Dict2CCL(json_data), state_file) # 保存修改
```

更复杂的使用示例：

```
from hoi4dev import *
import numpy as np # 用于随机
np.random.seed(42)

# 如果想要让一个地块影响所有的周围地块（比如临近地块人口都+1），我们需要先根据`map/definition.csv`与`map/provinces.bmp`建立省份之间的相邻关系，然后读取这个相邻关系进行操作。生成相邻关系我们将在工程文件实例中展示。也可以参考pdoc文档中的`hoi4dev.maps.map2graph`部分。
state_mapping = {}
for state_file in [f for f in ListFiles('history/states/') if f.endswith(".txt")]: # 查看history/states/中的所有地块描述文件
    state_id = int(state_file.split('-')[0].strip()) # 获得地块的数值id，因为地块文件名通常为：`1-France.txt`，所以可以用'-'切割。
    state_mapping[state_id] = state_file # 这样我们可以直接从地块数值id获得地块文件名，即从`1`得到`1-France.txt`。

for state_file in [f for f in ListFiles('history/states/') if f.endswith(".txt")]: # 查看history/states/中的所有地块描述文件
    adjacency = LoadJson('state_adjacency_graph.json') # 读取相邻关系，我们这里假设已经用`hoi4dev.maps.map2graph.BuildAdjacencyGraph`完成了预处理
    state_id = int(state_file.split('-')[0].strip()) # 获得地块的数值id
    for adjacent_state_id in adacency[str(state_id)]: # 查看所有相邻地块：
        adjacent_state_file = state_mapping[adacent_state_id] # 读取相邻地块的文件所在位置
        adjacent_ccl_data = ReadTxt(pjoin("history/states/", adjacent_state_file)) # 读取相邻地块的钢4代码
        adjacent_json_data = CCL2Dict(adjacent_ccl_data) # 将这个地块的代码转为JSON格式，可以开始随意编辑了
```

```
18          adjacent_json_data['state']['manpower'] += 1 # 人口+1
19          SaveTxt(Dict2CCL(adjacent_json_data), adjacent_state_file) # 保存修改
```

(拓展阅读) 示例#2：添加一个民族精神

前述内容中，我们提到一个通用的资源文件夹格式。对于一个民族精神来说：

```
my_idea/
├── info.json
├── default.png
└── locs.txt
```

我们只需要简单写一点民族精神的代码 `info.json`，配一个默认图标 `default.png`，写描述 `locs.txt` 即可。

首先，HOI4DEV在工程文件配置中有一个决定民族精神图标大小的参数：

```
{
    ...
    "img_scales": {
        ...
        "idea": [74, 74],
        ...
    },
    ...
}
```

这意味着我们无论使用什么大小、什么格式的、名为 `default` 的图片文件，在转换为民族精神图标过程中都会先缩放至大约74x74大小，然后中心裁剪为74x74，再也不需要进行复杂的dds图片编辑了！同样地，这些图标会自动添加到 `.gfx` 文件中，不需要编写GFX文件代码。

接下来，HOI4DEV的民族精神模板会默认添加一些属性：例如默认为国家精神(`country`)、不可移除、内战保留等。开发者只需要写如下JSON代码：

```
{
    "modifier": {
        "consumer_goods_factor": 0.3,
        "stability_weekly_factor": -0.02,
        "local_manpower": -0.05
    }
}
```

就可以添加一个民族精神了，编译成钢4代码后的效果：

```
ideas = {
    country = {
        IDEA_my_idea = {
            picture = my_idea
            removal_cost = -1
            modifier = {
                consumer_goods_factor = 0.3
                stability_weekly_factor = -0.02
                local_manpower = -0.05
            }
            allowed_civil_war = {
                always = yes
            }
        }
    }
}
```

多语言本地化（精神名称和描述，域 `@` 代表精神名称 `IDEA_my_idea`，因此 `@desc` 代表 `IDEA_my_idea_desc`，就会自动成为精神描述的本地化）：

```
[en.@]
Dark Age
[zh.@]
黑暗时代
[en.@desc]
War, plague, famine and turmoil trouble this country.
[zh.@desc]
战争、瘟疫、饥荒和动乱困扰着这个国家。
```

(拓展阅读) 示例#3：构造一个智能笑话事件集

如果我想要为mod添加一个故事集/笑话集，可能有30个事件，那么我要为30个事件都去写代码么？最理想的自然是只写30份文本，然后共用一份代码。理想中，我们只需要下述资源文件：

```
jokes_txt/
├── joke_001.txt
├── joke_002.txt
├── joke_003.txt
├── ...
└── joke_030.txt
```

让我们来把这些文件转成事件 `JOKES.1`，`JOKES.2`，… `JOKES.30`！

首先，思考一个笑话事件的模板应该是什么样的，我们希望玩家可以浏览上一个笑话、下一个笑话、随机笑话，或者不看了。原本如果直接用钢4写代码，每个笑话事件的代码大概会是这样的：

```
country_event = {
    id = JOKES.3
    picture = GFX_EVENT_JOKES_3
    immediate = { # 标注这个笑话是否看过了，如果看过了，标记一个flag，之后我们在看随机笑话的时候就更少让它出现
        hidden_effect = {
            if = {
                limit = {
                    NOT = {
                        has_global_flag = PIHC_GLOBAL_FLAG_JOKE_003_SEEN
                    }
                }
                add_to_variable = { # 记录看过的笑话数量，之后可以用作成就之类的
                    var = global.VAR_PIHC_JOKE_COUNTER
                    value = 1
                }
                set_global_flag = PIHC_GLOBAL_FLAG_JOKE_003_SEEN
            }
        }
    }
    is_triggered_only = yes
    title = EVENT_JOKES_3
    desc = EVENT_JOKES_3_desc
    option = { # 不看了
        ai_chance = {
            factor = 100
        }
        name = EVENT_JOKES_3_o0
    }
    option = { # 随机笑话，这里我们之后再单独写一个JOKES.9999隐藏事件来触发一个随机笑话
        ai_chance = {
```

```
            factor = 0
        }
        hidden_effect = {
            country_event = {
                id = JOKES.9999
            }
        }
        name = EVENT_JOKES_3_o1
    }
    option = { # 下一个笑话
        ai_chance = {
            factor = 0
        }
        country_event = {
            id = JOKES.4
        }
        name = EVENT_JOKES_3_o2
    }
    option = { # 上一个笑话
        ai_chance = {
            factor = 0
        }
        country_event = {
            id = JOKES.2
        }
        name = EVENT_JOKES_3_o3
    }
}
```

因为HOI4DEV的事件模板会自动处理命名、图标、本地化等元素，我们只需要如下工程资源文件：

```
{
    "immediate": {
        "hidden_effect": {
            "if": {
                "limit": {
                    "NOT": {
                        "has_global_flag": "PIHC_GLOBAL_FLAG_JOKE_<当前笑话的三位id>_SEEN"
                    }
                },
                "add_to_variable": {
                    "var": "global.VAR_PIHC_JOKE_COUNTER",
                    "value": 1
```

```
                },
                "set_global_flag": "PIHC_GLOBAL_FLAG_JOKE_<当前笑话的三位id>_SEEN"
            }
        }
    },
    "options": [
        {
            "ai_chance": {
                "factor": 100
            }
        },
        {
            "ai_chance": {
                "factor": 0
            },
            "hidden_effect": {
                "country_event": {
                    "id": "JOKES.9999"
                }
            }
        },
        {
            "ai_chance": {
                "factor": 0
            },
            "country_event": {
                "id": "JOKES.<下一个笑话的id>"
            }
        },
        {
            "ai_chance": {
                "factor": 0
            },
            "country_event": {
                "id": "JOKES.<上一个笑话的id>"
            }
        }
    ],
    "is_triggered_only": true
}
```

本地化（依次为事件标题、事件描述、事件选项文本）：

```
[zh.@]
```

```
女皇的著作
[en.@]
Empress's Book

[zh.@desc]
和煦光流走进书店，发现书店里面摆满了自己的新书《新谐律主义思想》，非常高兴。
不过和煦光流在临走的时候叮嘱了书店的店长："也不要光摆我的书，也可以稍微摆一点别的种类的书嘛。"
店长回复到："不好意思，和煦光流女皇，别的书都卖完了。"

[en.@desc]
Cozy Glow walked into the bookstore and found her new book, "Neo-harmonism Explained", displayed all over the store, which made her very happy.
However, before leaving, Cozy Glow reminded the store manager: "Don't just sell my books. You can also display a few other types of books for variety."
The manager replied: "I'm sorry, Empress Cozy Glow, but all the other books have been sold out."

[zh.@o0]
不看了
[en.@o0]
I'm Done
[zh.@o1]
随机笑话
[en.@o1]
Random Joke
[zh.@o2]
下一个笑话
[en.@o2]
Next Joke
```

因为这里我们不需要事件图，可以用统一的默认图。

于是，我们就可以用Python批量生成这个工程文件了：

```python
from hoi4dev import *

all_joke_txt_files = [f for f in ListFiles("jokes_txt/", ordered=True) if f.endswith(".txt")]
for i, joke_txt_file in enumerate(all_joke_txt_files, 1):
    joke_locs = ReadTxt(pjoin("jokes_txt", joke_txt_file)) # 读取笑话的文本内容
    joke_folder = pjoin("resources", "events", "JOKES", f"{i:03d}") # 创建JOKES事件空间中的一个事件
    CreateFolder(joke_folder)

```

```
    joke_json_data = LoadJson(SINGLE_JOKE_TEMPLATE_PATH) # 读取我们刚才写的模板
    joke_data["immediate"]["hidden_effect"]["if"]["limit"]["NOT"]["has_global_flag"] = \
        joke_data["immediate"]["hidden_effect"]["if"]["limit"]["NOT"]["has_global_flag"].replace(
            "<当前笑话的三位id>", f"{i:03d}" # 替换模板中的事件id
        )
    joke_data["immediate"]["hidden_effect"]["if"]["set_global_flag"] = \
        joke_data["immediate"]["hidden_effect"]["if"]["set_global_flag"].replace(
            "<当前笑话的三位id>", f"{i:03d}" # 替换模板中的事件id
        )
    joke_data["options"][2]["country_event"]["id"] = \
        joke_data["options"][2]["country_event"]["id"].replace(
            "<下一个笑话的id>", f"{i+1}" # 替换模板中的下一个事件id
        )
    joke_data["options"][3]["country_event"]["id"] = \
        joke_data["options"][3]["country_event"]["id"].replace(
            "<上一个笑话的id>", f"{i-1}" # 替换模板中的上一个事件id
        )

    button_locs = [ # 自动添加四个选项的本地化
        {
            "zh": "不看了",
            "en": "I'm Done"
        },
        {
            "zh": "随机笑话",
            "en": "Random Joke"
        },
        {
            "zh": "下一个笑话",
            "en": "Next Joke"
        },
        {
            "zh": "上一个笑话",
            "en": "Prev Joke"
        }
    ]
    if i == 1:  # 第一个事件，就不需要上一个的按钮
        joke_data['options'].pop(3)
        button_locs.pop(3)
    if i == len(all_joke_txt_files): # 最后一个事件，就不需要下一个的按钮
        joke_data['options'].pop(2)
        button_locs.pop(2)

```

```
    joke_locs += "\n"
    for j, button in enumerate(button_locs): # 追加四个（或少于四个）选项的本地化
        joke_locs += f"[zh.@o{j}]\n{button['zh']}\n[en.@o{j}]\n{button['en']}\n"

    # 保存修改和本地化
    SaveTxt(joke_locs, pjoin(joke_folder, "locs.txt"))
    SaveJson(joke_data, pjoin(joke_folder, f"info.json"), indent=4)
```

接下来我们写好 `JOKES.9999` 隐藏事件，在随机 `random_list` 里面留空：

```
{
    "immediate": {
        "hidden_effect": {
            "random_list": {
                "seed": "random",
                "// 之后每个笑话都会添加到这个地方": null
            }
        }
    },
    "options": [
        {
            "ai_chance": {
                "factor": 100
            }
        }
    ],
    "hidden": true,
    "is_triggered_only": true
}
```

我们希望没看过的笑话在随机中优先出现，让看过的笑话权重为1，没看过的为8：

```
random_joke_folder = pjoin("resources", "events", "JOKES", "9999")
CreateFolder(random_joke_folder)
random_joke_folder = LoadJson(RANDOM_JOKE_TEMPLATE_PATH) # 读取我们刚才写的模板
for i, joke_txt_file in enumerate(all_joke_txt_files, 1):
    # 注意JSON不支持重复键值，所以HOI4DEV了函数`find_dup`来添加重复键值，这里是插入一个概率权重为`1`的事件
    random_joke_data["immediate"]["hidden_effect"]["random_list"][find_dup("1", random_jokes)] = {
        "modifier": {
            "factor": 8,
            "NOT": {
```

```
                "has_global_flag": f"PIHC_GLOBAL_FLAG_JOKE_{i:03d}_SEEN" # 如果没见过这个事件，那么概率权重上升为8
            }
        },
        "country_event": f"JOKES.{i}" # 触发事件
    }
# 保存修改，因为是隐藏事件，反正玩家看不到，可以选择不做本地化
SaveJson(random_joke_data, pjoin(random_joke_folder, "info.json"), indent=4)
```

更进一步，如果我们想要添加一个玩家看完所有笑话的成就，那么成就代码应该大致是这样的：

```
{
    "possible": {
        "difficulty > 1": null,
        "has_start_date < 1002.12.2": null,
        "has_any_custom_difficulty_setting": false,
        "game_rules_allow_achievements": true
    },
    "happened": {
        "hidden_trigger": {
            "check_variable": {
                "var": "global.VAR_PIHC_JOKE_COUNTER",
                "value": "<笑话总数>",
                "compare": "greater_than_or_equals"
            }
        }
    }
}
```

我们只需要稍微修改这个成就文件即可：

```
achievement_data = LoadJson("achievements/READ_ALL_JOKES/info.json")
achievement_data["happened"]["hidden_trigger"]["check_variable"]["value"] = int(len(all_joke_txt_files))
SaveJson("achievements/READ_ALL_JOKES/info.json")
```

从此以后，我们每次想要添加一个笑话的时候，只要新开一个文本文件把笑话写下来就好了，HOI4DEV和刚刚我们这段脚本会自动完成笑话事件的生成、默认配图、多语言本地化、随机、成就更新，添加100个笑话也丝毫不会累！