



图书馆代码系列教程

题目：动态修正效果显示的自动化解决方案

类型：高级代码

作者：CancerCendil

图书馆编号：HC000002

所属组(选填)：红与白：炼铸钢铁的人们

完成日期：2025/04/08

基于版本：1.16.*

联系方式(选填)：

摘 要

本文主要分为两个部分：**Token-valued variable** 的基本介绍，以及一个实际运用例子：动态修正效果显示的自动化解决方案（也即所谓“字符串变量”的使用例）。

我们都知道（**Dynamic modifiers**）动态修正是个方便动态添加修正（**Modifier**）的好东西，只需要更改其中定义的修正所被赋予的变量的值就可以完成效果修改（这里不再赘述其具体写法和用法）。

但是与民族精神（**Ideas**）相比其缺陷之一就在于需要自行本地化效果变化的提示，而常用的解决方案无论是 `Set_variable` 中的 **Variable Tooltips**，或是使用 `swap_ideas`，都不尽如人意——前者需要自行编写本地化，后者则牺牲了可维护性（试想一下有多种 **ideas** 的情况组合，或是完成编写之后需要往中间加入一个新的修正）。

因此，为了解决这一问题，就可以采用 **Token-valued variable** 的相关特性，结合 `meta_effect`，编写动态修正效果显示的自动化解决方案。

关键词：高级代码；钢铁雄心 4 教程；Data structures 数据结构；Token-valued variable；

目 录

第一部分：Token-valued variable 的基本介绍	1
第二部分：动态修正效果显示的自动化解决方案	3

第一部分：Token-valued variable 的基本介绍

本部分是为没接触过 Token-valued variable 的人准备的，其中绝大部分内容来自于[数据结构 - Hearts of Iron 4 Wiki --- Data structures - Hearts of Iron 4 Wiki](#)，如果已经有所了解可以直接跳转第二部分。

在开始前，我们已经假设了读者对 Variable 有所了解。

Token（标记）可以被理解为是游戏内部的一种令牌或索引，每种可标记数据拥有其唯一的 Token。当使用 `token:some_tokenizable_value` (`token:[某种可标记数据]`) 时，其实际上得到的是一个整数变量（通常具有较大的值），所以可以放入任意的变量中进行自由传递；也可以通过与其他标记变量进行比较，判断相等（即判断为同样的可标记数据，这是由于 Token 唯一确定；此外，判断大于小于实际上没有意义）。

当某一变量（例如此处定义了 `Var_with-Token` 这一变量）拥有被赋予的标记值时，（`civilian_economy` 是民用经济的民族精神名）

例如 `set_variable = { Var_with-Token = token: civilian_economy }`

我们可以通过函数 `GetTokenKey` 得到其标记的字符串值，

例如 `[?Var_with-Token. GetTokenKey]` 实际上得到的是 `civilian_economy`。

此外，我们还可以通过函数 `GetTokenLocalizedKey` 得到其标记对应的本地化键值，实际上是检索与标记的字符串值同名的本地化键的值，

在中文环境下，`[?Var_with-Token. GetTokenLocalizedKey]` 自然得到的是民用经济。

在目前版本(1.16)，已知的可标记数据类型（但可能不限于）：（来自 wiki）
Ideas, 任何在 `/Hearts of Iron IV/common/script_enums.txt` 中列出的内容，
Ideologies/Ideology groups, Technologies, Equipment types, Operations, Characters/Unit leaders, Decisions, Buildings。

由于 Token-valued variable 提供了太多可能性，我们在此无法举例其所有可运用的范围和领域，wiki 上的[数据结构 - Hearts of Iron 4 Wiki --- Data structures - Hearts of Iron 4 Wiki](#) *Advanced use of tokens* 小节大概的讲了一些，感兴趣的可以详细看一下。

我们在这里介绍一种比较直接的运用：使用 Idea Token-valued variable 构造某种意义上的“字符串”变量。

首先为了使用 Idea Token-valued variable，我们必须定义 idea。

例如，

```
Token_valued_IDEA_1 = {#名字自行斟酌
    removal_cost = -1
    allowed = { always = yes }
    modifier = {
    }
    #由于不需要任何效果，所以理论上里面一行都不用填
}
```

然后再编写其本地化，

```
Token_valued_IDEA_1: "variable"
```

此时我们就近似于创建了变量名称为 `Token valued IDEA 1`，变量值为“variable”的字符串类型变量（实际上是常量）。

可以通过之前我们讲过的方式来使用这个虚构的字符串变量，

首先 `set_variable = { Var_with-Token = token: Token_valued_IDEA_1 }`

然后就能通过函数 `GetTokenKey` 得到所谓的字符串变量名称 `Token valued IDEA 1`

```
[?Var_with-Token. GetTokenKey]
```

通过函数 `GetTokenLocalizedKey` 得到所谓的字符串变量值 “variable”

```
[?Var_with-Token. GetTokenLocalizedKey]
```

这样就间接实现了“字符串”变量的构造。

你可能会问这有什么用呢，确实乍一看没什么用，但是只要结合元效果/元触发器（`meta_effect/meta_trigger`），以及动态文本（`scripted_localisation`），就可以任意构造想要的效果和触发器（当然，这也只是其中一种运用思路）。

如果不清楚元效果/元触发器（`meta_effect/meta_trigger`）以及动态文本（`scripted_localisation`）怎么使用的，可以去看 wiki 的相关页面，本文不过多赘述，这亦是第二部分所要重点用到的内容。

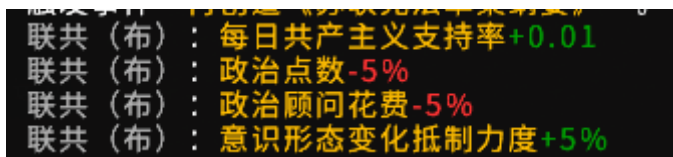
第二部分：动态修正效果显示的自动化解决方案

在开始前，我们假设了读者已经对变量（Variable），标记（Token），元效果（meta effect），动态文本（scripted localisation）有着基本的了解。

为了实现自动化解决方案，我们首先需要对效果显示的本地化进行拆解：

（例如）

【动态修正名】：【修正名】【数值变化】



这只是其中一种拆解方案，可以自行进行构造不同的方案。

这里很明显，前两者是“字符串”变量，最后就是 p 社最常见的 variable：

很遗憾的是，【动态修正】不是可标记数据类型，这意味着它没有可读取的 Token，所以我们只能创建“idea 类型的字符串”作为代替，

我的“联共（布）”动态修正名为 SOV_SUCC_dynamic_modifier

为了可读性，我将对应的 idea 命名为 SOV_SUCC_dynamic_modifier_ik（不建议重名，这样读取本地化键值会冲突），然后将本地化键值填写为“联共（布）”

第二个是 modifier，是可标记数据类型，不需要过多操作。

第三个数值在颜色和格式上的变化属于动态文本和本地化的基本内容，我们在这里不赘述了。

这里我将存储【动态修正名】token 的变量设置为 idea_buff_name

```
set_temp_variable = { idea_buff_name = token: SOV_SUCC_dynamic_modifier_ik }
```

将存储【修正名】token 的变量设置为 idea_buff_type

```
set_temp_variable = { idea_buff_type = token: political_power_gain }
```

将存储【数值变化】的变量设置为 TEMP_CHANGE_var

那么本地化就填写为：（CHANGE_BUFF_TTTT_TEST 是我的本地化名称）

```
[?idea_buff_name.GetTokenLocalizedKey][?idea_buff_type.GetTokenLocalizedKey][?TEMP_CHANGE_var]
```

```
set_temp_variable = { idea_buff_type = token:political_power_gain }
set_temp_variable = { idea_buff_name = token:SOV_SUCC_dynamic_modifier_ik }
set_temp_variable = { TEMP_CHANGE_var = 1 }
custom_effect_tooltip = CHANGE_BUFF_TTTT_TEST
```

（注：这里只能用 temp_variable，不然不能在同一代码块内结算效果！）

显示效果 **联共（布）：political_power_gain1**

然后你会发现怎么 token: political_power_gain 无法得到本地化呢？

由前文可知通过函数 `GetTokenLocalizedKey` 得到其标记对应的本地化键值，实际上是检索与标记的字符串值同名的本地化键的值。

而 p 社的 Modifier 本地化，大部分是这样的

MODIFIER_POLITICAL_POWER_GAIN: "每日获得的政治点数"

而由于 `MODIFIER_POLITICAL_POWER_GAIN` 并不是真正的 modifier，因而没有 token 写

```
set_temp_variable = { idea_buff_type = token:MODIFIER_POLITICAL_POWER_GAIN }
```

实际上会报错提示你不存在这种 token

这里暂时有两种解决方案：

第一种，补充对应的本地化键，即补充

political_power_factor: "每日获得的政治点数"

其实并不难，大部分 modifier 都是被 p 社前面加上了 `MODIFIER`

```
MODIFIER_CARRIER_TRAFFIC: "航母起降能力"
MODIFIER_CARRIER_NIGHT_TRAFFIC: "航母夜间起降能力"
MODIFIER_POLITICAL_POWER_GAIN: "每日获得的政治点数"
MODIFIER_POLITICAL_POWER_COST: "每日消耗的政治点数"
MODIFIER_POLITICAL_POWER_FACTOR: "政治点数"
```

所以只需要复制一个 p 社自己的本地化文档，利用全局搜索去掉 `MODIFIER` 即可

补充后效果 **选择后效果：联共（布）：每日获得的政治点数1**

第二种，使用动态文本

这个提一下思路就行，比如

```
#token本地化转换为中文type
defined_text = {
    name = Get_CN_idea_buff_type
    text = {
        trigger = {
            check_variable = {
                idea_buff_type = token:navy_submarine_defence_factor
            }
        }
        localization_key = MODIFIER_NAVY_SUBMARINE_DEFENCE_FACTOR
    }
}
```

判断对应的 token，写上对应的 p 社的实际本地化键

再把本地化对应的部分改为动态文本即可


```

CHANGE_BUFF_TTTT_TEST: "[?idea_buff_name.GetTokenLocalizedKey]:[Get_CN_idea_buff_type][?TEMP_CHANGE_v
"每日获得的政治点数"

```

效果是一样的

最后再优化一下颜色和数值即可。

在理解上述的基础之上，我们还可以改进，将本地化和**实际效果**（即设置修正的变量数值的过程）融合在一起。

我们知道设置变量数值的过程大部分时候是对某一变量做加法运算（减去就是加负数），所以结合 meta_effect 我们可以得到一个 scripted effect

```

SOV_buff_change_effects = {
  meta_effect = {
    text = {
      add_to_variable = {
        [BUFF_NAME]_var_[BUFF_TYPE] = TEMP_CHANGE_var
      }
      custom_effect_tooltip = REAL_CHANGE_TT
    }
    BUFF_NAME = "[?idea_buff_name.GetTokenKey]"
    BUFF_TYPE = "[?idea_buff_type.GetTokenKey]"
  }
  clear_variable = idea_buff_type
  clear_variable = idea_buff_name
  clear_variable = TEMP_CHANGE_var
}

```

代价是我们相应的需要将动态修正的变量名进行拆解和格式化（不过这本来就是为了可读性而必须要做的），

```

political_power_gain = SOV_SUCC_dynamic_modifier_var_political_power_gain #每日政治点数

```

可以看到变量被拆成了【动态修正名】_ik_var_【修正名】，

你也可以根据自己的习惯来拆，只要你能利用“字符串”完成拼接过程即可。

```

set_temp_variable = { idea_buff_type = token:political_power_factor }
set_temp_variable = { idea_buff_name = token:SOV_SUCC_dynamic_modifier_ik
set_temp_variable = { TEMP_CHANGE_var = 1 }
SOV_buff_change_effects = yes

```

这样 meta_effect 实际执行的代码就是

```

add_to_variable = {
SOV_SUCC_dynamic_modifier_ik_var_political_power_factor = 1
}
custom_effect_tooltip = REAL_CHANGE_TT

```

也是完全对应了我们所设置的变量名

实际效果是在改变了变量的同时，同时拥有数值变化的本地化。