前言

这是关于 Paradox Interactive （P社） 旗下 Paradox Development Studio 工作室使用 Clausewitz 引擎的游戏的语言教程（注： Jomini 引擎是图像引擎，游戏主体的脚本代码仍然是 Clausewitz ）
P语言即指 Clausewitz 引擎游戏的脚本代码，相当偏向自然语言。
~~一部分P社员工的文化水平是速度和路程都搞不清楚的小学生，不自然语言怎么活~~

从本文中，你也许能学到：

- 制作模组的基础知识
- 解决模组冲突问题
- 模组排序思路
- 自己动手修复P社的奇怪问题

本文不会包含：

- 逐条的文档解读或翻译，请自己输出文档（推荐）或查Wiki
- 某个特定游戏的完整内容编辑教程

希望本文能为玩家扫盲起到一定作用~~完全不看好~~。
本人能力有限，若发现错误请及时指正，感激不尽。

作者：Steam@Saltfish，欢迎探讨技术问题，但请确保你看完了这篇文章。

基本道德

笔者认为，**知识**不应收费，任何人只要愿意学习都应能得到相关资料，尤其是这种完全明码的游戏本来就是旨在便于玩家修改。
——虽然某些Modder利用知识壁垒牟取暴利，某些Modder制作恶作剧模组来恶心玩家，某些Modder剽窃代码并故意影响原模组兼容性、盗取代码据为己有
但相信不作恶的人比作恶的人更多。如果人人都会写，上述问题自然不成问题。

尽量做到：

- 注明自己的设计思路来源于他人的内容
- 为所有没有提及使用许可的内容向作者请求使用/修改许可
- 在共享协议和使用许可没有要求的情况下注明引用来源的出处和作者
  至少要做到：
- 遵守写明的共享协议和使用许可
- 不破坏引用内容的正常功能和兼容性——这在P社引擎机制下具有严重的实际影响，要改至少重命名一下
- 不将他人作品据为己有
  保持礼貌根据不同人的观点可能排在上述任意一处。
  *请*做不到至少遵守*协议*和*许*可的*读*者立即*删*除本文件

预备知识

近年的P社游戏作品：

注意，不同作品的语法和脚本功能之间存在一定差异，本文会尽可能注明不同游戏特有的特性差异。

1. CK2/十字军之王2/Crusader Kings II
2. EU4/欧陆风云4/Europa Universalis IV
3. 群星/Stellaris
4. HOI4/钢铁雄心4/Hearts of Iron IV
5. IR/统治者：罗马/Imperator: Rome
6. CK3/十字军之王3/Crusader Kings III
7. Vic3/维多利亚3/Victoria 3

上述游戏中，笔者没有玩过IR。 部分游戏因为年代久远，细节已经忘却。 本文**主要以EU4、CK3和Vic3为例**进行讲解。

根据语法和游戏功能，笔者将现下流行的P社游戏分为数个世代

- 古代：Vic2、HOI3及更早的游戏
- 中间世代：CK2、EU4、群星、HOI4
- 新世代：IR、CK3、Vic3

各世代的典型特征如下：

- 古代游戏存在高度素材复用，代码功能极其受限，尤其是筛选器的功能很糟糕。
- 中间世代游戏的代码功能明显增加，从EU4开始使用yml本地化。其中CK2和EU4没有原生双字节支持，需要转码和补丁实现双字节语言。
- 新世代游戏的语法有较大改动，且增加了 **Jomini** 引擎管理美术和本地化等资源，下文（Jomini）即指使用Jomini引擎的游戏。

模组需要的编程习惯

这一节是给完全没有编程经验的读者的，已有充分基础的读者可以跳过

缩进

代码缩进是**非常重要**的习惯，缩进可以让你的代码层次结构一目了然。

理论上你可以把所有P语言代码写在一行，但你一定是硅基生物。

P社员工喜欢用制表符`/t`来缩进，表示为 ，这种字符会自动调整长度使其结尾成为固定长度的整数倍，根据平台不同通常是4或8个空格。
目前流行的是4字符缩进，但如前所述，平台不同显示效果不同，因此使用空格是较好的做法
~~但混用比用制表符更糟糕，且跨平台只是难看不影响运行，所以不必过分在意~~

有缩进：

```
event = {
    trigger = {
    }
    option = {
```

```
    }
}
```

无缩进：

```
event = {
trigger = {
}
option = {
}
}
```

可以想见代码复杂之后无缩进有多么难以阅读。**愿智慧离开不缩进者。**

不过，极其简短的或连续而高度重复的代码可以直接写进一行，注意操作符之间仍然要保留空格。

另外，现代代码编辑器通常以缩进来判断代码折叠，因此对于多段内容相似的代码块，可以通过下一节所说的**注释**功能将相似代码/单一功能块写在一起方便折叠：

```
blabla = {
    ### block 1 ###
        if = { limit = { trigger1 = yes } effect1 = yes }
        else_if = { limit = { trigger2 = yes } effect2 = yes }
        else_if = { limit = { trigger3 = yes } effect3 = yes }
    ### block 1 end ###
}
```

CK3的活动类型中，活动背景和实体动辄几百上千行，可达全文件四分之一篇幅，却不使用这种方式折叠，可见新员工业务水平之差。
相反，CK3的人物模型就使用了这一技巧，将各部分内容分开，方便查阅。

注释

#后的内容不会被识别，用于解释代码功能，或暂时屏蔽代码。引擎**只支持单行注释**，一些编辑器会提供多行注释快捷键。

为复杂功能和封装代码写清楚注释，免得日后忘记，也方便他人阅读。

修改原版文件应该在开头和修改处注释修改的内容方便速查。

作为良好编程习惯的一部分，#前后不应与非空白字符相连。

命名

为你的模组起一个缩写名称，然后在这个模组加入的**所有**文件名和命名空间、有名称的条目等的开头写上这个缩写，以便在报错日志中快速定位并避免同名冲突。

文件名应明确指出此文件的功能，并以文件分类结尾，如一个封装指令（scripted_effect）文件应以`_effects.txt`结尾，以方便排查错误。~~CK3 1.13起因为报错机制更新，这个结尾标志变得没有那么必要了~~

无论引擎是否支持其他字符，所有的命名应该**严格按照C语言变量命名标准**进行（只包含英文字母、数字和下划线，且不以数字开头，文件扩展名的'.'除外）。**不要学习P社员工使用连字符代替下划线！**

使用下划线分隔单词，或使用驼峰法（每个单词首字母大写，首个单词首字母大写为大驼峰，否则为小驼峰）。
原版的风格是下划线（Vic3原版甚至用了连字符，千万不要学）

保留字

传统语言的保留字指的是编程语言内部定义的词（字，word），不能再用于函数名、变量名等。
P语言中也有许多这样的保留字，不过某些场合下并非严格地不能使用。
安全起见，不建议使用任何看起来像是原版会用作基础语句的词语和词组。

字符

绝对不要在本地化之外的地方使用任何非ASCII字符。除非你在填写本地化，否则输入法永远保持英文状态。

其他常识

要看懂本文，你需要：

- 基本的逻辑能力，能够理解**与或非**和分支结构（如果满足某条件则执行……），能了解德摩根定律者更佳
- 小学英语词汇量，或至少能使用搜索引擎查询单词含义、阅读P社文档
- 了解ASCII码字符排序逻辑，理解字符串排序和数值排序的区别
- 理解权重概念，会计算加权随机事件发生概率
- 理解字面值、整型int、浮点型float、字符串string的概念，虽然P社引擎并不明确使用这些概念，但本文尽可能严谨地列出了使用场景
- 理解树状结构，理解一个对象的直接父子关系
- 理解编程中函数的概念

准备工作

推荐环境：
建议电脑内存在8GB以上，实际运行中16GB物理内存+适量虚拟内存（笔者使用了60GB）可以支持代码编辑器+游戏本体同时流畅运行。

代码编辑器

推荐使用微软的免费软件VSC Visual Studio Code
注：VSC的中文界面也是一个插件。
下载Paradox Syntax、CWTools等插件获得对P语言的支持
内存不足不要使用CWTools，这个插件会载入整个工作区以提供补全代码，占用巨大内存。

插件的代*码检查*不一定是*对*的，以*实际为*准

绝对不要使用记事本之类的文本编辑器

项目管理工具（高级向）

如果你希望制作大规模的模组，或多人协作：

- 推荐下载Git。
  因众所周知的网络问题，国内推荐使用Gitee而非GitHub，但两者的操作本质上没有区别，只是连接地址不同、语言不同（P.S.Gitee也有英文版本）
- 在VSC中配置Git
- 将VSC设置为Git的默认代码编辑器
- 为Gitee（或GitHub等其他平台）配置SSH密钥
- 推荐使用VSC插件码云（Gitee）常用功能，在VSC设置中输入`@ext:hbybyyang.gitee-vscode-plugin`，将私钥填入设置
- 在文件夹下依次输入

```
git init
git remote add origin 远程仓库地址（SSH：  git@gitee.com:xxx/yyy.git）
git pull --rebase origin master
git push -u origin master
```

以创建本地仓库并连接远程仓库。
关于Git的使用此处仅列出VSC环境配置所需步骤，具体使用和指令请参考Git教程，或与其他协作者一起学习。

启动器

对于EU4及之后的游戏，P社使用统一的Paradox V2，这是一款**质量极差的垃圾软件**，旨在推广广告，且经常搞坏模组引导文件，对于国内用户更是经常因为无法连接P社论坛导致无法正常使用——是的，这款单机游戏的启动器需要联网才能使用一部分功能，尤其是当你能连创意工坊却连不上论坛的时候居然 不 能 上 传 模 组 ！。
推荐使用Irony Mod Manager，可以自动修复损坏的引导文件、简便且低资源占用。
只需点击红色加号创建新合集，其他操作非常明显。

如果有绕开P社启动器上传工坊的需求，可以尝试使用SteamCMD，此处不再赘述。

P语言总论

严格意义上，P语言**分为三个部分**，游戏运行相关的Clausewitz脚本、显示相关的Jomini代码和GUI代码。 EU4起、Jomini前的游戏虽有自定义UI，但功能参差不齐。
自Jomini引擎起，GUI的非布局部分和本地化脚本使用了相同的语法，且功能更加多样化，本文姑且称作Jomini代码，但部分逻辑在Jomini前的游戏已在使用。

作为入门教程，本文主要讲解前者，其特征是使用大量的 `= { }`。~~GUI明明也用~~
在花括号前的等号同时具备了作用域切换、复杂条件和指令的调用等诸多功能。

关系运算

对于有相关背景的读者，这里需要特别强调，P语言的等号重载了比较和赋值。
Jomini引擎之前的游戏在作为数值比较时相当于`>=`。
（Jomini）语法已改为规范的比较（`> >= = <= < !=`），但仍然不使用`==`。

文件与条目

文件的定义不必赘述，但注意严格意义上文件名是包含其路径的，在不同相对路径中，如`/common`下和`/events`下的同名文件实际上**不是同名**，不会互相覆盖。

Windows下，P语言的文件格式是`UTF-8-BOM`，（不确定其他系统是否也是），而流行的编辑器默认格式是`UTF-8`，这通常只会导致每个文件产生一行报错，而不会导致运行问题。
Windows下，`CRLF`和`LF`换行符均可以正常生效而不产生报错或实际问题。

P语言以大量内核指定的路径来识别对应文件应当被作为何种文件读取，显然游戏文件放在错误的目录就不能正确生效。

Clausewitz脚本文件的扩展名均为`.txt`

注意，Windows系统使用`\`作为路径分隔符，其他系统使用`/`，两者均可在游戏引擎中正常读取，故推荐使用`/`，因为前者兼作转义符，需要反转义，否则可能引起错误。
在文件排序中，子文件夹总是排在母文件夹的非文件夹文件前。

条目是代码文件中的部分，或理解为覆盖/抢占机制中的单位，例如一个事件文件中的每一个事件是一个条目。
显然，在不同类型的文件中，条目的粒度会有巨大的差异。

一个略有些反直觉的例子是CK2和EU4的文化和宗教，两者分别属于文化组和宗教组。
可以通过将新的文化/宗教直接贴在原版文件定义过的文化组/宗教组下，从而实现追加或覆盖已有文化/宗教，如：

```
# 原版文件
宗教组1 = {
    宗教组1信息
    宗教1 = {
        宗教1信息
    }
    宗教2 = {
        宗教2信息
    }
    ...
}
宗教组2 = {
    宗教组2信息
    宗教3 = {
        宗教3信息
    }
    宗教4 = {
        宗教4信息
    }
}
```

```
# 载入顺序靠后的模组文件
宗教组1 = {
    宗教3 = {
```

```
            宗教3信息
        }
    }
```

如此，则宗教3被移入宗教组1。

模组结构

CK2以前的游戏，模组管理较难实现，经常以整合包的形式发布。

自CK2开始则有较好的模组生态~~但是P语言兼容性天然一坨~~。

Vic3以前的模组的结构包括：

- 一个.mod文件描述模组的名称、路径和其他信息，关键在于路径。一般称作描述文件，但笔者认为此文件称为引导文件为宜。 这种文件必须放置在模组目录，如文档/Paradox Interactive/游戏名/mod，创意工坊订阅的模组也会自动在这个目录创建引导文件。
- 一个文件夹包含具体的模组内容，且其中还有一个descriptor.mod文件，是实际的描述文件，启动器会自动将此文件中的描述同步到前一个.mod文件中。 这个文件夹理论上可以处于任何路径，只要有对应的引导文件。 按上述形式打包的压缩文件也可以被识别为模组。

Vic3开始不再需要外部引导文件，取而代之的是在模组文件夹/.metadata/metadata.json中以json语法描述模组，因此必须放在指定的载入路径才可以识别。

无论何种模式，引导文件或描述文件都可以：

- 声明模组名称、版本和支持的游戏本体版本，其中本体可以用`*`作为通配符替代最后一段版本号。
- 指定模组的前置模组以指定载入顺序。
- 声明独占任意个文件夹，阻断游戏引擎对对应原版文件的读取。

模组是对游戏的修改，文件组织格式（主要是目录名）必须同原版一致。
例如，你必须将事件文件放在`/events`下才能正确识别。

在较早的游戏中，原版文件不在最终文件夹下建立分级文件夹，而是直接放在一起。
而在较新的游戏（Jomoni）中，原版普遍使用分级文件夹以改善可读性，**请注意区分一个文件夹是内核的读取路径还是为方便分类文件建立的子文件夹**。
另外，`/common`下常有二级内核读取路径，如CK3的`common/artifacts/`下的各文件夹，请务必注意。 ~~强烈谴责P社员工风格不一致，一会扁平化一会分级目录~~

相反，在较新的游戏（Jomoni）中，`/gfx`下的文件有许多是非内核的，但在需要访问图像文件处不指定时，常有一默认路径来抓取图像。

模组的载入逻辑

本质上，引擎载入模组的方式是先载入原版文件，然后按顺序载入各模组的文件——如果出现同名就会覆盖先前的文件。 因此，许多模组需要调整排序才能兼容。

但，更多的问题即使排序也无法解决，必须制作兼容版本/兼容补丁。

文件同名覆盖——模组载入顺序的选择

在P社游戏中，游戏文件载入的逻辑是： 读入**所有**原版文件->根据模组载入顺序依次读取每个模组的**所有**文件

其中，读入是读入到一个虚拟文件空间，**因此遇到同名文件时，后载入的模组文件覆盖先载入的文件**。

模组的引导文件可以指定前置模组，从而固定在其后载入，可用于避免手动排序出错。

条目同名冲突——文件名的选择

在所有文件读入到虚拟空间后，进入加载阶段：

每个内核文件夹，按照文件名顺序（ASCII）读取，因此可以看到原版有很多以`00_`或`zz_`开头的文件，用以控制载入顺序。

如果遇到已经有同名条目载入的情况下再次读取到同名条目，根据具体内容不同会有不同的逻辑：

- （Jomini）绝大部分非GUI脚本采取后覆盖前，所有GUI脚本（含本地化）采取先抢占后。
- （Jomini前）事件等采取后覆盖前，`/common`中的内容则有许多是先抢占后。
- 少数文件采取追加机制，用此法处理会产生多个对象叠加。常见于地图相关的内容中，如EU4的贸易节点地图遇到同名省份多次出现会导致一个省份属于两个节点。

利用这种机制，可以通过补丁方法修改部分条目，避免覆盖整个文件，从而降低维护成本、提高兼容性。
（CK3）不能使用此法处理事件，P社引擎会直接随机取其中一个版本，**无视顺序**。

避免让补丁文件的头部和被覆盖文件的头部相同，P社的某种缓存优化似乎会直接忽略掉名字靠后、与已载入文件头部相同的文件。

其他

此外，某些功能存在匹配顺序，同样也是根据载入顺序判定——先匹配到则完成匹配退出，典型例子是EU4的国家理念National Ideas，每个国家初始化时按顺序检查每组国家理念，选取第一个满足条件的国家理念，因此TAG限定的理念优先级最高，其次是文化组、政体和缺省理念。

根据各功能的相关性，游戏引擎似乎对不同文件夹有一个固定的读取顺序，以确保游戏正确加载。

务必尽可能使同一个类型的文件不要出现循环依赖，并尽可能按正确的顺序排列存在依赖的同类条目
例：（CK3）某特质B附带有增加其他特质A经验获得速率的效果，此时该特质B即依赖于特质A，必须置于特质A的定义后，否则游戏无法正确识别。 ~~垃圾游戏没有函数原型~~

条件与指令 trigger and effect

- Trigger，字面译作触发器，但此处译为条件更合适
  用于判断是否成立（是/非），如是否是玩家。其结果是一个布尔类型（是或不是）的返回值。 **若条件为空或不写可选条件代码块，则相当于判定通过，返回是**
- Effect，字面译作效果，笔者认为效果不能反映其代码特性，故称作指令
  用于实际产生效应，如增加金钱。

初学者很容易混淆这两者，这里以一个if分支结构举例： if = { limit = { 条件 } 指令 } 满足`limit`中条件的情况下，`limit`后的指令才会被执行。 如果将条件和指令混淆，则不能正常触发效果。

另外一种分支条件是 trigger_if = { limit = { 条件1 } 条件2 } 这项功能最迟于EU4引入（**但EU4中此功能被重载到`if`中，不以`trigger_`加以区分，下文不再赘述**）用于简化显示。只有满足条件1的情况下才会显示需要满足条件2，因此在复杂条件的可视化中很有用。

由于分支结构和循环结构的存在，指令语句中可能包含条件语句，但条件语句中**不可能**包含指令语句。

无效的指令不会产生任何效果，而无效的条件总是返回真

内容的可见性 / visible

P社游戏的许多内容都有类似的结构：

- 可见条件，只有满足此条件才能看到相应内容
  - 在早期游戏中通常是`potential`
  - （Jomini）几乎全部统一为`is_shown`
- 有效条件，只有满足此条件才能执行相应内容（如决议）
  - 在早期游戏中通常是`allow`或`trigger`
  - （Jomini）几乎全部统一为`is_valid`
- 效果，即触发后执行的指令
  - 基本上都是`effect`

作用域 scope

作用域是P语言中最重要的概念，某些方面可以类比于对象，但P社游戏里几乎不存在对象的继承和多态性。

除了极少数作用于全局的代码，几乎所有的**条件**和**指令**都必须在正确的作用域才能生效。
（Jomini） `always = <scope>`可用于检测`scope`是否存在。

举例来说，CK中的角色是角色Character作用域，EU、Vic、HOI、群星中的国家是国家Country作用域，这两种作用域都是**完整**的作用域（死去的角色可以被记录但不能存储变量，直到CK3 1.12版本起引入了死者变量）。
与此相对的，Vic3的人群Pop作用域是**不完整**的。

- **完整**的作用域可以存储变量，可以被记录
- **不完整**的作用域则不能，只能现取现用

动态作用域

~~没有找到P社文档里对这些东西的定义，姑且沿用Wiki的说法。~~

这些作用域在代码的几乎所有地方都能使用，但指代对象则因具体情况而不同。

ROOT

最基础的作用域，表示当前执行内容所来源的作用域。

例如，事件的ROOT是触发这个事件的作用域，action的ROOT是触发这个action的作用域，决议的ROOT是执行决议的作用域（角色或国家，取决于游戏）

FROM

在HOI4及以前存在，是事件链或其他触发链的上一环。

如一个事件引发另一个事件，则后一个事件的FROM是前一个事件的ROOT。
互动常以FROM为发起者、而ROOT则是互动对象。

action等通常也会有FROM，详见其注解。
可以使用形如FROMFROM的多层回溯，上限层数不明。

PREV

上一层作用域，用于作用域跳转。

例如
此处是条件代码

```
scopeA = {
    scopeB = {
        PREV = scopeA
    }
}
```

返回：真 此时PREV等同于scopeA
在HOI4及以前，还可以使用形如PREVPREV的多层回溯，上限层数不明。

THIS

当前的作用域，主要用于带括号的复杂指令或条件，或者比较两个作用域是否相同
此处是条件代码

```
scopeA = {
    `this = scopeA`
}
```

作用域的功能类别

作用域可分为3类：

- 条件作用域
- 指令作用域
- 两用作用域 顾名思义，条件作用域实际上是一种条件，最终总要返回是或者非，而指令作用域是指令的一部分。
  访问条件作用域的语句通常以any_开头，而访问指令作用域的语句常以every_或random_开头。

两用作用域则既可以在条件中使用，也可以在指令中使用，其功能取决于处于代码的何处。通常是一些固定跳转的作用域，如省份/军队单位等的所有者、国家的首都等。 不过，也存在一些指向多个同类两用作用域的跳转指令，来自较早（CK2？）的历史遗留，~~是P社幼儿园员工没有学会英语时的黑历史~~。

控制台 & 开发模式 / console & develop or debug_mode

~启动，部分游戏需要在游戏外增加启动参数进入开发模式才能开启控制台，对于调试很有帮助，尤其是群星以后的游戏可以直接在控制台中执行脚本（Vic3除外）。 另外，最迟从CK3开始，开发模式可以**动态载入**大部分文件，即无需重启游戏就能生效修改。

最迟CK3开始，可以在控制台输入`script_docs`获得游戏指令文档，生成在文档/Paradox Interactive/游戏名/logs (CK3)或文档/Paradox Interactive/游戏名/docs (Vic3)
文档中包含了所有指令、条件、actions等的介绍和用例。

事件 event

最常见的指令执行载体。

在不同游戏中差异较大，主要体现在

- 事件类型 不同的游戏支持的事件作用域不同，如EU4支持省份事件和国家事件，CK支持人物事件，Vic3支持国家事件和地区事件
- 事件语法 旧引擎为

```
<事件类型> = {
    id =
}
```

Jomini引擎开始为

```
id = {
}
```

变量 variable

变量名和变量值的键值对。

和正经编程语言的变量相似，但操作要更麻烦。

旗标 flag （~CK3）

简化版的变量，只有变量名，也可以指定有效时间。由于变量语法的改进，在新引擎中已不常用。

时间概念和语句

时间概念广泛用于各种修正的持续时间、延迟触发的等待等。
截止EU5推出前，P语言使用以下时间尺度：

- `years` 年
- `months` 月
- `weeks` 周
- `days` 天
- `hours` 小时（HOI）
  需要注意，即使只有1个时间单位，也需要使用复数形式。
  （Jomini）所有出现的时间单位都可以随意混用
  （Jomini前）某些代码固定地要求特定的时间单位

下文所述的时间语句即形如`<尺度> = <时长>`的单句语句，如`days = 1`，P社文档中使用`mtth`表示时间语句即~~使是在删除这个功能的CK3里也这么叫，太屎出子~~

延时触发

在支持延时的代码块（如触发事件等）中增加一个时间语句即可，如
`trigger_event = { id = event_id.123 days = 1 }` 也可以包含随机因素，如下文代表在2~3日后触发。
`trigger_event = { id = event_id.123 days = { 2 3 } }`

修正 modifier

常说的buff和debuff，对特定作用域产生影响。 修正中通常包含一条或多条**实际提供buff或debuff的原子修正**，虽然也可以是只用于显示或记录的空修正。

例如，修正A使一个角色的健康+1、收入增加10%，其中健康和收入增加是原子修正。 修改原子修正的功能在EU4以后才引入，且CK3的修改实质上只影响显示格式，仍然只能使用内核生成的原子修正。

本地化 localization/localisation

注意，不同游戏中这个单词的拼写不同！！！
本地化是一种键值对映射，即在代码中写“键”（key），在本地化文件中写“值”（value），游戏引擎会自动在本地化文件中寻找对应键的值显示出来。

（~EU4）本地化没有原生双字节支持，需要双字节补丁+汉化MOD（字库）才能运行中文，并且需要专门转码工具，以原版的`english`身份借壳上市。

（~CK2）本地化文件的格式是键值和按顺序排列的多种语言本地化版本，以分号隔开。
（EU4~）使用（不严格的）yml格式，文件名必须以`_l_<语言代码>.yml`结尾，如`loc_l_simp_chinese.yml`。
开头首行是`l_<语言代码>:`，如`l_simp_chinese:`
其后的内容是每行一条`<键>:<版本号> "<本地化内容>"`，如`key:0 "值"`，版本号在较新的游戏（最迟CK3起）中无效，可以不写。
按照yml的标准，首行之后的内容是需要缩进的，且没有版本号一说，虽然实测部分游戏并不需要缩进，但建议按照规范写。
缺乏当前语言本地化值的键会直接显示~~懒鬼P社就不能弄个默认显示第一种语言~~

逻辑运算

Clausewitz引擎支持的逻辑运算如下：

- `AND` 与，其中的所有条件都为真时返回真，否则返回假。 没有特别指明时，所有的条件作用域都以这种模式运行。
- `OR` 或，其中的任一条件都为真时返回真，否则返回假。
- `NOT` 非，其中的条件为真时返回假，为假时返回真。 如前所述，没有特别指明时默认为`AND`模式，因此`NOT`中包含多条语句时**所有条件为真返回假，任一条件为假返回真**，使用`NAND`或`NOT = { AND = { 条件 } }`代替
- `NOR` （EU4等没有，~~因为挪威的TAG也是NOR~~） 全为假，其中任一为真则返回假，全为假则返回真。
- `NAND` （群星~？） 不全为真，实际上等价于`NOT`，但为了代码阅读无歧义建议使用。

分支结构

基本分支结构

如下：

```
if = {
    limit = {
        条件1
    }
    指令1
}
```

满足条件1时才执行指令1。

在其后，可以跟上任意数量的

```
else_if = {
    limit = {
        条件2...
    }
    指令2...
}
```

如果前一个`if`或`else_if`的条件不满足，则判定后一个`else_if`（若有）的条件，直到某一个`else_if`的条件满足则执行其指令

和最后的至多一个

```
else = {
    指令
}
```

如果此前的所有`if`和`else_if`的条件均没有满足，则执行`else`中的指令

位于同一层且连续的`if`、`else_if`和`else`才会被正常判定，`else_if`和`else`的前一个代码块（花括号）必须属于`if`或`else_if`

（EU4~）类似地，`trigger_if`只是将其中的指令换成进一步需要满足的条件：

```
trigger_if = {
    limit = {
        条件1
    }
    条件2
}
```

只有在满足条件1时才会显示条件2，并且可读性比使用复杂的逻辑运算要好。 需要特别注意的是，如果使用了`trigger_else_if`，则最后必须写`trigger_else`，否则会报错。

嵌套分支

`if`是一种指令，也可以写在`if`的指令部分中。
（EU4~）同理`trigger_if`是条件，也可以写在`trigger_if`的条件部分中。
这种操作称为分支结构的嵌套，如

```
if = {
    limit = {
        条件1
    }
    if = {
        limit = {
            条件2
        }
        效果2
    }
}
```

当然，此例完全可以将条件1和条件2用`AND = {}`连接起来，无需两次嵌套。

脚本代码分论

事件

绝大多数人的模组历程开始于不满足控制台的作弊功能，于是写了一个只能从控制台触发的作弊事件来实现更复杂的功能。不过作为一个全面的教程，事件其实没有太多可讲之处，毕竟关键在于事件内的指令和条件的组合。

事件文件位于游戏（对Jomini引擎游戏，是指/game文件夹下而非游戏根目录）或模组根目录下的/events文件夹中

命名空间

在事件文件的开头`namespace = 命名空间`。
最迟在HOI4起支持一个文件声明多个命名空间，多个命名空间时，每个命名空间的声明应该在前一个命名空间的事件结束后，该命名空间的事件开始前声明。
命名空间在游戏启动时被映射到一串数字前缀中，可以在`logs/game.log`中看到各命名空间在游戏**启动**时映射到的数字前缀。
游戏引擎实际处理的是映射后的纯数字ID，但使用命名空间极大地提高了可读性，并有助于防止ID冲突。
事件ID的格式是`命名空间.数字`，虽然较早的一些游戏有无命名空间的纯数字ID，但这是**非常非常糟糕的行为**。

没有声明命名空间而直接使用会导致**事件无法正常读取**。

事件结构

旧引擎事件的基本结构：

```
事件类型 = {
    id = <事件ID>
    # hidden 和 标题+描述 选择其一
    hidden = yes
    title = 标题
    desc = 描述
    # is_triggered_only 和 mean_time_to_happen 选择其一，如果两者均无则相当于
mean_time_to_happen为1天，疯狂触发
    # mean_time_to_happen即mtth，发生一次事件等待时间的数学期望
    # 只能被触发（不会随机发生）
    is_triggered_only = yes
    # 平均发生一次的时间
    mean_time_to_happen = {
        <时间语句>
        modifier = {
            # 此处应该不能使用add代替factor
            factor = 修正系数
            条件
        }
    }
    # 只触发一次，在EU4中是全局只触发一次。
    fire_only_once = yes/no
    trigger = {
        可选，触发条件
    }
    immediate = {
        可选，触发时立即执行的语句
    }
    # hidden的事件建议不写选项而在immediate执行
    option = {
        name = 选项
        选项效果
        # 可选，ai选择这个选项的倾向，加权随机，若所有选项均未填则等概率随机
        ai_chance = {
            factor = 基础倾向
            <modifier = {}>
        }
    }
    after = {
        # 可选，选择选项后，事件结束时执行的效果
        # 部分游戏没有，EU4在最近的版本才引入
    }
}
```

新引擎事件的结构：

```
<事件ID> = {
    # hidden 和 标题+描述 选择其一
```

```
    hidden = yes
    title = 标题
    desc = 描述

    #  (~CK3) 可选对同一个触发主体的冷却
    #  (CK3) 其实质是在触发时为当前作用域设置一个名字等于事件ID的变量，有效期即冷却。
    # 可以通过直接设置相应变量的方式来屏蔽事件，但会刷warning报错。
    cooldown = { days/months/days = xxx }

    trigger = {
        # 可选，触发条件
    }
    weight_multiplier = {
        # 可选，在用on_action随机触发时对基础权重的修饰
        base = <script_value>
        <modifier = {}>
    }
    on_trigger_fail = {
        #  (CK3) 被指定触发但触发条件不满足时执行的代码，Vic3不确定有没有
    }
    immediate = {
        # 触发时立即执行的语句
    }
    # hidden的事件建议不写选项而在immediate执行
    option = {
        name = 选项
        选项效果
        # ai选择这个选项的倾向
        ai_chance = {
            factor = 基础倾向
            <modifier = {}>
        }
    }
    after = {
        # 选择选项后，事件结束时执行的效果
    }
}
```

（本地化键不属于事件脚本的一部分）除事件ID之外，事件框架的所有数字都可以使用全局脚本值替代。 不同的游戏还有一些细节内容，如EU4的跳转`goto`、CK3的主题`theme`、Vic3的显示地点`placement`等，以原版文件为准。
其中`title`、`desc`、`name`都可以使用复杂的套娃本地化，见下文本地化一节。

作用域

关联作用域的切换

以现实为例，拜访一个人之后可能想拜访其邻居，或者在其家中稍作休息。
同样地，访问一个作用域之后可能需要访问与之相关的其他作用域——当然也可能是直接跳转到另一个无关的作用域。
如从一个国家访问其拥有的所有省份，或从省份访问其所有者。

在较新的（Jomini引擎起？）游戏中，可以直接使用.来连续跳转单一指向的作用域，如CK3中，`ROOT.liege.liege = { 代码 }`访问ROOT的领主的领主，无需写成

```
ROOT = {
    liege = {
        liege = {
            代码
        }
    }
}
```

不过，若要访问其全部封臣的封臣，则必须写成

```
ROOT = {
    every_vassal = {
        every_vassal = {
            指令代码
        }
    }
}
```

因为`every_vassal`会返回（遍历）不止一个作用域。

- 固有关联作用域（两用）可以在游戏输出文档的event_targets.log中看到
- 指令作用域可以在effects.log中看到，这个文件还包含了其他指令的说明
- 条件作用域可以在triggers.log中看到，这个文件还包含了其他条件的说明

访问单一关联作用域

如前所述，绝大多数作用域**根据其类型**都存在一些固有的**单一**关联作用域，但对于具体的作用域个体，这些关联作用域**不一定存在**。
单一关联作用域是**两用**的。
例如CK中，父、母是角色的单一关联作用域，且一个人显然只有一个父亲和一个母亲，相反孩子则不是单一关联作用域，因为一个人可以有任意个孩子，所以寻找一个角色的孩子要使用下一节的访问不定关联作用域。
然而，在游戏中，一个角色并**不一定有**明确的父母——毕竟不可能一路考据到猿人。

所以使用时必须确保想要切换的作用域**存在**，注意通常被消灭的国家也是**不存在**的（HOI4的是特例，总是存在），但死去的角色是**存在**的，只是功能受限。
以访问角色的父亲为例：

- 可以使用 `father ?= { }` 来安全访问作用域（Jomini），此时引擎会先检测`?=`**左侧**的作用域是否存在。
- 在此前的游戏中则可以加一句 `exists = father` 进行确认，前者本质上是这种方案的代码糖。
  注意如果条件需要被显示（如决议的执行条件），则前述写法**仍然会引起报错**，因为需要显示的条件会被全部判定，而非按照正常逻辑在确定不满足之后停止判断。（Jomini~？）这种场合应该使用

```
trigger_if = {
    limit = {
        exists = father
    }
    father = {
        条件
    }
}
trigger_else = {
    exists = father
}
```

注意部分游戏没有`trigger_if`，应使用重载的`if`代替。

访问不定关联作用域

通常也作迭代器，变量列表也是迭代器。
继续之前的例子，一个角色可能有多个孩子，因此不能直接以`child = {}`来访问，此处不定指数量不定。

相应地，可以使用`any_` `all_` `every_` `random_` `ordered_`等前缀，不同游戏的具体支持不同：

- any_
  条件域，访问的任一作用域满足其中条件则返回真。
  （Jomini）可以使用`count = 数值/all` **或** `percent = [0,1]浮点数`来实现对一定数量或比例的要求，支持完整的数值大小比较，但只能使用字面值。
  （Jomini）可以使用`filter = { 条件 }`代码块增加额外条件，只有满足`filter`中条件的才会被计入`percent`语句的比例分母。
- all_ （EU4~HOI4？）
  条件域，访问的所有作用域均满足条件则返回真。
- every_
  指令域，对所有访问的关联作用域执行。
  （CK2？）两用，包含了`all_`的功能
- random_
  指令域，随机选取一个访问的关联作用域执行。
  （Jomini？）可增加`weight = <script_value>`代码块调整随机权重，不填为等概率随机。
- ordered_ （Jomini）
  指令域，在`order_by`中指定排序关键词（脚本量script_value，见后文脚本量一节），然后选取第`position`（字面值）大的，或选取最大`max`（脚本值）个，再弃去其中前`min`（字面值）个。

指令域可以增加`limit = { 条件 }`代码块来限定范围。
（Jomini）可以在`limit`后加任意数量的`alternative_limit = { 条件 }`，在前一组限定条件不满足时依次判定后一组限定条件，直到找到满足的或判定完全部限定条件。

要判定是否存在至少一个不定关联作用域：
正确的做法：可以在`any_`访问后接`always = yes`，如

```
any_child = {
    always = yes
}
```

错误的做法：

```
exists = any_child
any_child = {}
```

注意避免过多层级/大范围的遍历或大规模运算，以免因内存不足而导致游戏崩溃

非关联作用域的切换

在某些场合，可能需要从一个作用域跳转到完全无关的另一个作用域。
例如，我们希望在长江发岩浆时把岩浆倒到斯德哥尔摩（EU4）。
显然，长江沿线省份和斯德哥尔摩没有任何关联。
所以，代码应该这样写：

```
1 = {
    倒岩浆
}
```

其中，1是斯德哥尔摩的省份ID，这是较早一些游戏的语法。

在Jomini引擎的游戏中，不再以固定的键值来访问作用域，而是以作用域类型:作用域键的方式来访问，这种模式有更好的可读性和扩展性，如：

- `title:e_roman_empire` (CK3)
- `cu:han` (Vic3)

所以，如果我们希望在Vic3中往斯德哥尔摩倒岩浆，则应该写成

```
s:STATE_SVEALAND = {
    random_scope_state = {
        倒岩浆
    }
}
```

注意这里还使用了一次关联作用域访问，因为Vic3中区域state_region中可能含有多个属于不同国家的地区state。

注意，非关联作用域也可能**不存在**，因此同样需要安全访问或确认，这个问题尤其常见于直接访问不存在的国家，P社自己都犯过十万甚至九万次。

原则上，访问非关联作用域必须以作用域类型:作用域键的格式访问，但仍有少部分语句直接访问**非关联作用域的键值而非作用域**，如Vic3中的历史初始化使用的create_building与create_pop均只接受键值（建筑key或文化key）。
特别地，CK3中的特质trait截止1.14版本仍未完成作用域化相应的标准化，此类作用域在切换时需要以正确格式访问，但在语句判定时既接受纯键值输入，也接受传入作用域，即has_trait = trait_key与has_trait = scope:trait都是合法的。

作用域的类别

如前所述，作用域有不同的类别，根据游戏不同可能有角色、头衔、国家、省份、地区、建筑、军事单位、星球等等。
在不匹配的作用域上，指令和条件不会正常生效。
特别需要注意的是CK的头衔title和省份province、Vic3的区域state_region和地区state是**不同**的作用域，这是非常常见的错误来源。

EU4**没有**人物作用域，对君主、继承人、配偶、顾问的访问本质上是国家域内互相独立的条件/指令。

非永久保存作用域

在连锁的执行/处理中可以将完整的作用域用一个别名保存，以供下游脚本使用。
每一次触发下游脚本的新执行都会复制一份**当前**已有的非永久保存作用域信息。
此处的连锁是指彼此有触发关系，并且显然不可能先使用后记录。
当所属事件链或其他触发链结束时，一个非永久保存作用域的**生命周期**就结束了，因此也就无法再通过这个别名访问此作用域，但不代表这个作用域也会被销毁。
注意： 将一个作用域保存为当前处理中已经存在的别名会报错且不会被执行。

保存作用域是**两用**的。

注意保存作用域也可能需要安全检查。

在Jomini引擎中：

- 指令域保存：save_scope_as = xxx或save_temporary_scope_as = xxx
- 条件域保存：save_temporary_scope_as = xxx
- 使用：scope:xxx
  save_temporary_scope_as = xxx与save_scope_as = xxx的主要区别：
- save_temporary_scope_as = xxx可用于条件域
- save_temporary_scope_as = xxx保存的作用域会在当前部分结束后销毁（如事件的整个条件trigger部分），而非触发链结束时。

许多action、互动等内核调用功能也会自动生成一些这样的作用域以方便处理，这取代了旧引擎中的FROM/FROMFROM。

在较早的游戏（EU4、CK2等）中：

- 保存：save_event_target_as = xxx
- 使用：event_target:xxx

永久保存作用域

保存作用域是**两用**的。

注意保存作用域也可能需要安全检查。

使用变量保存作用域（Jomini）

可以通过将作用域作为变量值的方式永久地保存作用域以便访问

- 保存：`set_variable = { name = xxx value = scope:actor }`
- 使用：`var:xxx`
  不过，这样就需要注意变量记录在了哪个作用域上。

当然，也可以用全局变量代替，这样就无需考虑这个问题:

- 保存：`set_global_variable = { name = xxx value = scope:actor }`
- 使用：`global_var:xxx`

使用全局事件对象保存作用域（CK2~群星？）

- 保存：`save_global_event_target_as = xxx`
- 使用：`event_target:xxx`

印象中CK2的语法略有不同，以实际为准。

数值型作用域（Jomini）

- 指令域保存：`save_scope_value_as = {}`
- 条件域保存：`save_temporary_scope_value_as = {}`
- 使用：`scope:xxx`

相当于一般编程语言的**常量**，代码块内的语法同Jomini变量，同样支持布尔型、数值型和flag型，调用仍然和普通作用域一致，和普通作用域一样，只能被删除再次保存，不能直接在原值基础上运算修改。

条件/指令的脚本化 （封装） scripted_trigger/scripted_effect

严格来说，我们编写的模组代码都是脚本，注意不要混淆。
P社游戏并没有函数这一概念，只能通过事件或“scripted”（脚本，照着稿子念）的条件和指令来封装一系列代码。
封装的意义在于：

- 减少代码量
- 提高可读性
- 保障一致性，修改封装内容即相当于修改了全部调用处，而无需考虑是否漏过修改某处
- 提供接口，便于其他模组的兼容和联动
- 在涉及多个模组之间内容时抑制报错

脚本化基础

位于`/common/scripted_triggers`和`/common/scripted_effects`下，最迟于CK2出现。
例：

```
# /common/scripted_triggers 中的文件
scripted_trigger1 = {
    trigger1 = yes
    trigger2 = yes
}
# /common/scripted_effects 中的文件
scripted_effect1 = {
    effect1 = yes
    effect2 = yes
}
# 在绝大多数文件调用
if = {
    limit = {
        scripted_trigger1 = yes
    }
    scripted_effect1 = yes
}
```

在较新的游戏（HOI4~？）起，也可以在事件文件中以`scripted_trigger` 和`scripted_effect` 作为前缀以声明局部脚本化条件/指令：

```
# /events 中的文件
scripted_trigger scripted_trigger1 = {
    trigger1 = yes
    trigger2 = yes
}
```

调用同上，但这种写法**只在此文件中生效**，且（CK3确认，其他游戏不明）即使在其他文件再次定义也无法生效。

参数传递 （Jomini，EU4~，其他不确定）

通过传递参数，封装条件和指令可以更加方便地实现类似函数的功能，当然也可以通过变量和作用域间接实现更完整的参数功能。
P语言的参数传递**本质上是字符串替换**，只能直接使用**字符串**类型的入参，不能直接使用出参。
习惯上，参数名使用全大写
~~EU4的传参功能是半路出家比较野，P社还在用小写~~ 不同游戏的处理逻辑有巨大差异，原则上在接受参数的封装语句中使用`$`包裹输入参数名表示引用该入参，本地化的引用也遵循类似逻辑。

- （Jomini）使用`$PARAMETER$`，只能使用**不含空白字符（空格、制表符、换行符等）的连续字符串**
- （EU4）使用`"$PARAMETER$"`，参数本身可以跨行
  以下是一个简单例子，其中`IN_PARAMETER`是传入的一个参数，显然应该是一个作用域的**名称**，注意调用并提供参数的一侧（左值）不需要加额外的标识符。

```
# 定义
scripted_effect_name = {
    liege = {
```

```
        $IN_PARAMETER$ = { # 此处$IN_PARAMETER$相当于一般编程语言的形式参数
            <effect>
        }
    }
}
# 调用
scripted_effect_name = {
    IN_PARAMETER = ROOT # 此处IN_PARAMETER相当于一般编程语言的形式参数，ROOT相当于一般
编程语言的实际参数
}
# 等价于：
liege = {
    ROOT = {
        <effect>
    }
}
```

避免在传递作用域名称时使用liege等单一关联作用域或PREV、THIS等基于上下文的动态作用域（ROOT在一次执行之中不会发生变化，因此不受影响）。
由于传递的是字符串，如果在相关封装中切换了作用域，这些字符串被填入之后便可能不再是调用时的相应作用域。
如上例，若将传入参数由ROOT改为liege，则实际执行时作用域便会变为调用处的领主的领主，而非直接领主。
需要传递此类作用域时应使用保存作用域代替。
作为入参的字符串可以被任意拼接，也可再次作为参数调用下一级封装，如下

```
# 定义
scripted_effect_1 = {
    set_variable = {
        name = var_name
        value = flag:$IN_PARAMETER$
    }
}
scripted_effect_2 = {
    scripted_effect_1 = { IN_PARAMETER = $IN_PARAMETER$ }
}
# 调用
scripted_effect_2 = {
    IN_PARAMETER = PREV
}
```

（EU4）参数传递带有一些特殊功能：

- 可以通过always = $IN_PARAMETER$的形式（注意此处无需"包裹）检查是否填写了对应的入参
- 可以通过[[IN_PARAMETER]<语句>]的形式根据是否有对应参数实现分支结构
  下面两种写法是等价的：

```
scripted_effect_1 = {
    if = {
        limit = {
            always = $IN_PARAMETER$
        }
        <effect>
    }
}
scripted_effect_2 = {
    [[IN_PARAMETER]
        <effect>
    ]
}
```

注意：（Jomini）所有入参必须在调用时填满，否则整条调用无法正确识别。
在模组间互动时，应当在占位符中无实际意义地调用封装的入参，以免因参数不匹配而被忽略封装的占位符，以下是简单的解决方法：

- （EU4）通过`always = $IN_PARAMETER$`检测的方式
- （Jomini）对于条件类，先按照正常逻辑设置恒为真或恒为假，再用`save_temporary_scope_value_as`将入参作为不可能被执行的赋值结果
- （Jomini）对于指令类，使用`if`代码块，条件设为恒为假，再用`set_variable`或类似语句将入参作为不可能被执行的赋值结果

显示内容的自定义和隐藏

- `hidden_trigger = {}` 内的条件不会被显示
- `hidden_effect = {}`内的指令不会被显示
- `custom_tooltip = <loc_key>`显示一条本地化。
- （Jomini） `custom_tooltip = {}`和`custom_description = {}`内的条件和指令不会显示，改为显示这两种代码块指定的一条本地化。

```
custom_tooltip = {
    text = loc_key
    ...
}
# custom_description 还有 custom_description_no_bullet 这一变体，区别在于前者会自动在本地化前增加一个`·`以模仿原版本身的本地化风格
custom_description = {
    text = <effect/trigger>_loc_key # 此处并非本地化键，见下文本地化一章中的条件与指令的本地化。
    subject = <scope> # 可选，默认为当前作用域
    object = <scope> # 可选，提示中的另一个作用域
    value = <script_value> # 可选，显示的数字
    ...
}
```

在最终的本地化中，`subject`指向的作用域使用其作用域类型（全字大写），`object`在前者基础上在前面加上`TARGET_`，而`value`指向的数值则使用`$VALUE$`引用。

脚本值 script_value (Jomini)

脚本值相当于一种**数值型**静态变量，这里先讲脚本值的原因是在Jomini游戏中变量的运算也使用脚本值的处理逻辑。
脚本值的性能比变量好，所有运算有条件都应该优先使用脚本值。
脚本值文件位于`common/script_value`，注意与其他script开头的文件夹不同，脚本值**不是**`scripted`。
需要注意，脚本值**不限于**此文件夹，这一逻辑实际上涵盖了Jomini游戏的Clausewitz代码中几乎所有的数值型内容，但某些修正值和Vic3的历史类指令目前还不支持。
如果没有特别说明，本文提及需要脚本值处均可使用数值型字面值代替，在非Jomini游戏中可用的语句提到<script_value>时即只能使用数值型字面值
在游戏文档中会列出script_value或int，代表此处可用脚本值或只能使用字面整型

全局脚本值

```
my_script_value = 1
my_script_value_2 = {
    value = 0
    add = {
        value = 1
        multiply = 2
        desc = entry_desc
    }
    every_living_character = {
        subtract = var:abcd
    }
    if = {
        limit = {
            always = yes
        }
        multiply = my_script_value
    }
    divide = define:NCharacter|XXXXX
    modulo = 233
    min = 1
    max = 10
}
```

如上所述，脚本值可以是简单的定值（相当于全局常量），也可以经过一系列运算访问包括变量、其他脚本值、定义值define在内的其他数值，支持赋值`value`、加`add`减`subtract`乘`multlipy`除`divide`取模`modulo`运算和最大`max`最小`min`值限制。
脚本值的运算实质上与其他代码相同，从上到下逐条执行，也可以使用分支和作用域跳转。
运算从`value`赋初值开始，无初值默认为`0`。
可以出现任意多的`value`，通常用于简单逻辑下直接根据条件选择值。
每一条运算语句中都可以像上述代码中的`add = {}`代码块一样再次嵌套运算。
每一条/层运算语句都可以带有`desc = loc_key`，在需要拆分显示数值

（CK3:`GetScriptValueBreakdown('script_value')`，Vic3：`SCOPE.GetScriptValueDesc('script_value')`）时便可以展示每一条数值的具体来源。

脚本值的作用域

需要特别注意，封装在脚本值文件中的脚本值**属于一个作用域**，并且在不同的游戏中所属作用域的逻辑不同。
可以通过`<scope>.script_value`的方式访问不同作用域的脚本值，就像访问作用域的固有属性一样。

- 在截止1.12版本的CK3中，脚本值属于调用它的局部作用域，即脚本值的ROOT相当于它的THIS
- 在截止1.7版本的Vic3中，反直觉的是，脚本值属于调用它的整个代码段的作用域，即脚本值的ROOT相当于调用它的代码的ROOT
  这种区别导致Vic3中的脚本值必须在调用和编写时仔细确认作用域来源，以免发生读取错误。

```
# common/script_value/xxx_values.txt
my_scripted_value = {
    value = {
        if = {
            limit = {
                is_ai = no
            }
            value = 1
        }
        else = {
            value = 2
        }
    }
}

# 假设后续代码在任意以玩家为ROOT的指令代码中触发
# 我们知道Vic3的角色并没有金钱这项数据，这段代码旨在区分Vic3和CK3对脚本值的不同处理逻辑。
add_gold = my_scripted_value
every_living_character = {
    add_gold = my_scripted_value
}
```

在两种逻辑下，前一句将得到相同的结果，即玩家得到1金币
但后者则会得到截然不同的效果：

- 在CK3式逻辑下，每个角色分别计算这一值，所有存活的AI角色得到2金币，而存活的玩家角色得到1金币
- 在Vic3式逻辑下，所有角色统一使用触发此代码的玩家的值，因此所有存活的角色得到1金币

局部脚本值

相当于一般语言的局部变量，只在声明局部变量的文件中可以访问。
大部分脚本文件都可以声明，一般写在文件头部。
局部脚本值以@开头，使用与全局脚本值不同的逻辑，在一行之内运算完成，如：

```
@poor_tax_base = 0.25
@poor_tax_tier_1 = @[poor_tax_base]
super_poor_building_tax_tier_1 = @[poor_tax_tier_1 / 2]
```

可见，局部脚本值通过`[]`可以实现引用和简单运算，并且可以局部脚本值和全局脚本值可以互相引用，只要当前文件中声明了相应的局部脚本值。
与封装的条件和指令不同，同名局部脚本值可以在不同文件中出现。

匿名脚本值

在这种替代中，脚本值实际上成为了匿名的，即没有名称而直接被代码访问。
下方的代码展示了一个简单的匿名脚本值，可以看到和全局脚本值的写法是一致的。

```
random = {
    chance = {
        value = 0
        add = 1
        ......
    }
}
```

脚本值的导入

可以使用`"value(<scope>)"`格式读取须取目标作用域的复杂条件的条件值，作为脚本值使用。
如：

```
"task_contract_location.squared_distance(root.location)"
# 此代码取root.location与task_contract_location的距离平方，由如下条件衍生而来：
squared_distance = {
    target = task_contract_location
    value <= root.location
}
```

此法可读取条件的具体数值，而无需使用大量elseif判断。
（CK3）可以使用`|`分割入参，如`has_trait_xp(lifestyle_hunter|falconer)`输出`lifestyle_hunter`特质的`falconer`路线的特质经验。
（Vic3）部分常用条件也可以作用域固有属性的形式访问，以`:`标注条件参考对象，
如`"scope:target_country.relations:root"`

注意

脚本值在执行中可能出现后发先至的连锁错误，尤其是在涉及作用域的销毁时。
避免在执行中访问会因本连锁执行发生改变的脚本值。

变量 variable

在不同游戏中变量的功能差异相当大，在Jomini以前，变量是唯一能处理复杂运算的手段。 变量必须属于一个作用域，或者是**全局变量 global_variable** (Jomini)。
不是所有作用域都可以有变量，如Vic3的人群pop就不能拥有变量。
变量的值可以是数值型、作用域/对象（仅Jomini，EU4部分支持）、布尔型（Jomini）、flag（Jomini，实际上也是一种对象类型），注意Clausewitz引擎的布尔型使用`yes/no`而非`true/false`。
Clausewitz引擎的数值型变量使用有符号定点数位移。在CK3中其值域为正负1e14-1，精度为小数点后两位，过多的小数位数会被截断，过多的整数位数会导致指令无效。
切记使用flag型变量时必须带flag:前缀，不能直接使用字符串

全局变量的使用

（Jomini）涉及全局变量的所有语句都只需在局部变量版本的`var`前加`global_`
（~HOI4）不存在全局变量，因此必须使用一个固定的作用域来代替，如：

- CK2: 任意总是存在的实地头衔~~P社用的教宗，结果因为这个屎出而拒绝出销毁教宗的功能~~
- EU4: 任意省份，也总是存在

变量的设置

以Jomini引擎为界，变量的代码有明显区别

```
# Jomini
set_variable = var_name # Jomini缩写格式，等同于
set_variable = {
    name = var_name
    value = yes
}
# `flag:`部分实现了传字符串的功能，但只能传入而不能传出，一般用于多选项的判断处理。
set_variable = {
    name = var_name_2
    value = flag:any_string
}
# 可以用变量记录一个完整作用域（的指针）
set_variable = {
    name = var_name_3
    value = scope:actor
}
# 变量可以指定有效时间，达到时间之后该变量便自动消失。
# 变量的取值可以调用脚本值。
set_variable = {
    name = var_name_4
    value = any_script_value
    months = 6
}
```

Jomini前的变量则费解得多：

```
# Jomini前
# 定值
```

```
set_variable = {
    which = var_name
    value = 1
}
# 复制当前作用域的其他变量值
set_variable = {
    which = var_name
    which = other_var_name
}
# 复制其他作用域的同名变量值
set_variable = {
    which = var_name
    which = PREV # 或任何其他作用域
}
```

变量的判断

```
# Jomini
# 判断存在有两种方法
exists = var:XXX
has_variable = XXX
# 判断取值，需先判断是否存在，如前文所述支持各种简单数值关系运算。
var:XXX >= YYY
```

```
# 非Jomini
# 只有数值型，未定义变量默认为0，且为0的变量不会被记录到存档中
# 注意此处的`value = `实际上是大于等于，而要实现小于的判定则需要套一层NOT
check_variable = {
    which = <var_name>
    value = 123
}
```

作用域型变量使用

（Jomini）`var:var_name`，两用。
（~EU4？）功能可能不完整，`variable:var_name`，调用其他作用域的变量则如`variable:FROM:var_name`，似乎只能用于右值，不能用于跳转作用域。

数值型变量的进一步运算赋值

Jomini变量的修改非常简单，如同**一次**脚本值运算，注意应该尽可能使用脚本值来储存系数以获得更好的性能和可靠性。

```
# Jomini
change_variable = {
    name = <var_name>
```

```
    add = 1
}
change_variable = {
    name = <var_name>
    add = {
        value = 1
        multiply = var:var_name_2
    }
}
# 限制最大最小值
clamp_variable = {
    name = <var_name>
    max = <script_value>
    min = <script_value>
}
```

而非Jomini的变量则麻烦得多，以下4种语句分别对应了加减乘除。

```
change_variable = { which = var_name value = 1 }
subtract_variable = { which = var_name value = 1 }
multiply_variable = { which = var_name value = 1 }
divide_variable = { which = var_name value = 1 }
```

变量的移除

```
remove_variable = var_name
remove_global_variable = var_name
```

导出到变量（EU4/非Jomini？）

变量可用于导出一个作用域的某项数值，如税收、金库等。
`export_to_variable = { which = <var_name> value = <property> who = <scope> }`
但更为强大的功能是，这一语句可以调用对象的任何修正值、累加数据等等，从在游戏内积攒的专制度到从堆修正获取的外交声誉均可输出：
`export_to_variable = { which = <var_name> value = trigger_value:<modifier_value> who = <scope> }`
甚至可以通过这一语句读取一些通常无法获得的类型为作用域值的变量
~~强烈谴责P社至今没给CK3提供类似功能~~

旗标 flag （~CK3）

Vic3已完全删除此功能，事实上旗标的功能在Jomini引擎游戏中已完全被变量所覆盖。

旗标的设置

`set_<scope type>_flag = flag_name`，如`set_character_flag`或`set_country_flag`

旗标的读取

同上，`has_<scope type>_flag = flag_name`

旗标的取对象技巧

在没有存储作用域功能的CK2等早期游戏中，旗标有独特的作用：可以通过`set_flag = flag_name@scope`来记录一个作用域，如`set_character_flag = source@FROM`，如此则`flag_name@A`和`flag_name@B`是不同的flag。
注意`set_flag`**不是正确的代码**，因为不同作用域类型需要使用不同的语句。

全局旗标

在Jomini以前，flag有全局版本，即`<set/has>_global_flag = xxx`，此功能在Jomini后被全局变量取代。

on_action

`common/on_action`或`common/on_actions`，不同游戏不同 可译作动作，但一般直接称action避免歧义。 ~~想不出来好翻译，摆子~~

在较早的游戏中只能由内部代码触发，但最迟从CK3起可以由action套娃调用其他action，从而**极大提高了兼容性**。
注意：不同游戏此路径单复数形式不一致！！！务必确认原版文件！！！

详细的支持代码块参见`_on_actions.info`或类似的info文件，此处仅列出最关键的部分。

```
on_action_name = {
    trigger = {
        # 触发条件
    }
    weight_multiplier = {
        base = <script_value>
        <modifier = {}>
    }
    events = { # 依次触发以下所有事件
        event_id1
        delay = { <时间语句> } # (Jomini)
        event_id2
        event_id3
    }
    first_valid = { # 触发第一个`trigger`代码块返回真的事件后结束此代码块
        event_id_1
        event_id_2
        fallback_event_without_trigger
    }
    random_events = { # 加权随机触发一个`trigger`代码块返回真的事件
        <weight> = event_id1
        <weight> = event_id2
        <weight> = event_id3
    }
    on_actions = {
        action1
        action2
    }
```

```
    random_on_actions = {
        100 = on_action_1
        200 = on_action_2
        100 = 0
    }
    first_valid_on_action = {
        on_action_1
        on_action_2
    }
    effect = {
        # 执行效果
    }
    fallback = another_on_action
}
```

其中，仅有trigger、effect和weight_multiplier代码块采用正常的**覆盖规则**。 而on_actions可以触发其他的action，且是叠加的，意味着不同文件在同一个action后追加action**不会互相覆盖**。
random_系的每个选项都按照<数值型字面量> = <事件ID或action名>描述，此数值即为加权随机中该选项的基础权重，并可被该选项（要触发的事件或action）的weight_multiplier进一步修饰。 如果没有任何事件或action在effect之外的代码块被直接触发，作为替代，引擎将调用fallback指向的action。

（Jomini）此外，还能以触发事件的形式触发action：

```
trigger_event = {
    on_action = on_action_name
    <可选：延时时间语句>
}
```

较旧的游戏如EU4则采用如下语法：

```
on_action_name = {
    # 执行效果
}
```

这种模式下不同文件的同名action会互相**覆盖**，且action都由游戏内核触发，除本身相当于effect代码块之外，只拥有random_events、events代码块。

执行顺序（Jomini）

测试版本 CK3 1.13 effect > event > actions 应为并行逻辑

杂项

特别说明，在EU4等游戏中的游戏开始action包括读档，需要检查游戏是否开始来判断是初始化还是读档。

在Jomini中则只有游戏初始化才执行，如果需要读档立即执行的指令，请参考自定义GUI配合脚本GUI执行。

随机性

Clausewitz引擎支持的随机功能主要包括：

- `random = {`
- `random_list = {`
- 前文提到的`random_`访问不定关联作用域
- 前文提到的`action`的`random_`随机触发事件或`action`
- `= { min max }`，使左侧指令的取值在`[min,max]`间，只有一部分指令语句支持，典型是用于延时触发的时间语句
- 自定义本地化`customizable_localization`及类似随机本地化`random_valid`的随机性
  注意，目前只有指令支持随机性，并且脚本值本身没有原生随机性支持。
  如果在自定义GUI的本地化中直接使用随机功能，会导致显示内容以CPU时钟级的频率不断刷新。

本节主要讲前两者。

单随机 / random = {}

采用百分比描述和计算发生概率

```
random = {
    chance = <script_value> # 百分比概率
    <effect>
}
```

如：

```
random = {
    chance = 50
    add_gold = 1
}
random = {
    chance = {
        value = 0
        add = 50
    }
    add_gold = 1
}
```

以上两者是等价的，均为50%概率得到1金币。

随机列表 / random_list = {}

采用加权随机计算概率

```
random_list = {
    <num1> = {
        效果1
    }
```

```
    <num2> = {
        trigger = {
            条件2
        }
        <modifier = {}> # 权重修正，(Jomini)
        效果2
    }
    ...
}
```

一个random_list代码块必须含有不少于2个子条目（选项），子条目以一个数值型字面量（不能是脚本*值*）开始，这个数值就是其基础权重，可选代码块包括条件、权重修正（Jomini）和效果。
如果条件满足，基础概率经过权重修正的全部计算得到最终权重，每一个选项被选中触发的概率就是该选项基于最终权重进行加权随机计算得到的概率。

随机权重的修饰

注意不要与一般的修正 modifier 相混淆，修正是面向作用域的，而权重修饰面向一个涉及加权随机的代码块
涉及AI选择事件选项、各类随机列表等的权重修饰，但具体写法有所不同，主要分为两类，即脚本值逻辑和modifier逻辑，而具体的关键词又进一步有所差异，并可能在代码块内关键词和逻辑完全一致的情况下，有许多因文件目录而异的代码块本身名称。
典型的几种权重修饰代码如下：

```
# 事件选项，或其他需要AI在数个选项中选择的
ai_chance = {
    # jomini前用factor，jomini用base
    base = <script_value>
    factor = <script_value>
    # 支持任意数量个modifier代码块，按从上到下检查条件并连续计算
    modifier = {
        # add加，factor字面意为因子或系数，在此处即乘。
        # 不支持减法和除法，必须使用加法或乘法进行逆运算
        <add/factor> = <script_value>
        # 不满足条件的modifier被忽略
        <trigger>
    }
}
#  (Jomini) 用于随机列表触发事件、自定义本地化和action等
weight_multiplier = {
    # 内容同 ai_chance
}
#  (Jomini) 用于`random_`类不定作用域等
weight = <script_value>
# 部分作为游戏内核调用的weight代码块必须写成下面这种形式
weight = {
    value = <script_value>
}
```

截止1.12版本，CK3的random_调用不定关联作用域的weight中使用与ai_chance一致的格式（base+modifier），而非像上文一样直接调用脚本值。

（CK3）除了modifier之外，在存在对应作用域的场合还可以使用compare_modifier和opinion_modifier。此功能也可以通过将条件值转化为脚本值的方法opinion(scope:target)实现。
例： 下面的代码表示当条件满足时，取target作用域（可选，如当value为数值型作用域时即不填）的value（不确定该值是否必须是内核对象属性，或可以使用脚本值），加上offset~~P社自然语言数学~~，乘以multiplier，step作用未验证，应为将数值取到step的倍数，最小不小于min，最大不大于max，本条目在细分显示时用loc_key注明。

```
compare_modifier = {
    trigger = {
    }
    target = <scope>
    value = <property value?>
    offset = <script_value>
    multiplier = <script_value>
    step = <script_value>
    min = <script_value>
    max = <script_value>
    desc = "<loc_key>"
}

opinion_modifier = {
    who = <scope>
    opinion_target = <scope>
    ...
}
```

opinion_modifier则用who = <who>（可选，不填为当前**角色**作用域）和opinion_target = <opinion_target>作为关系判断的主语和宾语代替了target和value，其余代码块不变，即从who对opinion_target的看法作为原始值，后续运算一致。

警告：
如果所有子条目都不满足条件，引擎将随机选择，可能导致意想不到的后果。
如果需要复杂条件处理，应该设置一个0权重的空选项，因为条件判定通过的0权重选项优先权高于条件判定未通过的选项。

列表 / list （Jomini）

虽然叫列表，但是相当于一般语言的**集合**（没有重复元素），与变量一样有本地和全局之分，并且本地列表还在有效范围/生命周期上分为临时列表list和永久列表variable_list~~鬼知道P社为什么要叫他可变列表，明明都可变也都可以存作用域对象~~，前者逻辑如同非永久保存作用域，后者逻辑如同变量。

列表语法

添加元素

将元素添加到不存在的列表时便会自动创建列表。
如果要加入的元素已经在该列表中，**不会执行任何效果**。

- `add_to_list = list_name` :将当前对象加入临时列表`list_name`中
- `add_to_variable_list = { name = list_name target = <scope>}`: 将`target`加入当前作用域的永久列表`list_name`中
- `add_to_global_variable_list = { name = list_name target = <scope>}`: 将`target`加入全局永久列表`list_name`中
  以上语句有的拥有一些完全没用的别名，或者说较短版本的才是它们的别名，如`add_to_local_variable_list`和`add_to_temporary_list`，作为对概念的理解，读者可自行推断其对应关系。
  后两者支持固定时间语句，经过指定时间后**该元素**从列表中移除。

移除元素

- `remove_from_list = list_name`
- `remove_list_variable = { name = list_name target = <scope>}`
- `remove_list_global_variable = { name = list_name target = <scope>}` 依次对应添加元素的三种语句

清空列表

- `clear_variable_list = list_name`
- `clear_global_variable_list = list_name` 依次对应添加元素的后两种语句，临时列表没有清空语句

列表规模判断

- `list_size = { name = list_name value >= <script_value> }`
- `variable_list_size = { name = list_name target >= <script_value> }`
- `global_variable_list_size = { name = list_name target >= <script_value> }` 依次对应添加元素的三种语句，判断列表中的元素是否至少有`script_value`个，当然其他关系运算也是支持的。
  注意： 临时列表版本用`value`而非`target`
  当列表不存在时也可以使用这些语句判断是否存在列表

列表元素的访问

同不定关联作用域，关键字为`in_list`，但使用列表时必须指明是何列表：

```
<prefix_>in_list = {
    # 临时列表用list，永久列表用variable
    list = list_name
    variable = list_name
}
<prefix_>in_global_list = {
    variable = list_name
}
```

列表元素判断

注意： 当列表不存在时会报错，必须确保列表存在（有（过？）元素）或先判断列表大小非0来检查

- `is_in_list = list_name`: 判断当前对象是否在`list_name`中
- `is_target_in_variable_list = { name = list_name target = <scope> }`: 判断`target`是否位于当前作用域的永久列表`list_name`中
- `is_target_in_global_variable_list = { name = list_name target = <scope> }`: 判断`target`是否位于全局永久列表`list_name`中

修正 / modifier

所有修正都必须属于一个作用域，如角色修正或省份修正

修正的构成

位于`common/modifiers`，较早的游戏用`common/event_modifiers`和`common/static_modifiers`分别处理自定义修正和内核修正。~~Vic组改文件夹名刷KPI~~
（Vic3）所有修正位于`common/static_modifiers`。
此外，EU4及更早的一部分游戏有`common/triggered_modifiers`等，即在普通的修正基础上再增加显示条件和触发条件。
一部分修正是**内核**的，必须有这些名字的修正被游戏引擎直接调用。

```
modifier_key = {
    icon = icon_name #  (jomini+CK2? ) 需要在texticon里声明
    scale = {  #  (CK3) 应用修正倍数
        value = scripted_value # 修正生效的倍数，接受修正的作用域的脚本值
        desc = base_value_description # 每一倍修正值的“来源”，显示类似于`每
base_value_description提供一份修正`，再加上一倍修正的数值。
    }
    # 任意条原子修正，亦可0条原子修正而作为占位修正
    <modifier_definition> = <script_value>/<boolean>
    stacking = yes #  (CK3) 修正是否可堆叠。在Vic3中总是可堆叠，更早的其他游戏总是不可
堆叠。
    # 一些早期游戏如EU4还有其他对这条修正生效的额外代码遗留，如
    religion = yes  # 国教变更时自动移除
}
```

修正的操作

（Vic3）引擎处理修正相关的代码时不再区分作用域类型，统一采用形如`add_modifier`的格式操作

增加修正

```
add_<scope_type>_modifier = <modifier_key>
add_<scope_type>_modifier = {
    modifier = <modifier_key>
    # 限时修正用，到期自动移除
```

```
    <时间语句>
    multiplier = <script_value> #  (Vic3) 修正倍率
    is_decaying = yes #  (Vic3) 令修正数值乘上一个（剩余时间/总时间）的系数
}
# 如
add_character_modifier = <modifier_key>
```

移除修正

```
remove_<scope_type>_modifier = <modifier_key>
```

检查是否有修正

```
has_<scope_type>_modifier = <modifier_key>
```

原子修正 / modifier_definition

原子修正分为两类，其取值分别为数值型和布尔型，布尔型原子修正也称为规则修正。
原子修正必然属于一类或几类作用域，但一部分原子修正在作用域结构上并不严格。
较高层级的修正中包含较低等级的原子修正会自动将此原子修正应用到每一个下级作用域。 此处的上下级指具有拥有关系的作用域，如（Vic3）地区是国家的下级作用域，建筑是国家和地区的下级作用域。
（CK3）没有上下级作用域，头衔也不是一个能接受修正的作用域。
（EU4和其他一些游戏）省份修正中包含国家的原子修正会对持有该省份的国家生效。

- Vic3和HOI4可自定义非内核原子修正，分别位于`common/modifier_type_definitions`和`common/modifier_definitions`
- CK3只能为内核原子修正设置显示格式，位于`common/modifier_definition_formats`
  内核原子修正是基于其他对象（如Vic3每种产品的产出修正）自动生成或游戏引擎直接指定的修正，但内核修正必须经过格式定义才真正存在且可用。

（Vic3）可以用`modifier:modifier_type_key`的方式读取某个原子修正在当前作用域的总和，从而实现真正的自定义修正。

不同游戏的原子修正显示格式有很大差异，但都包括了颜色（好坏，决定正值为绿色还是红色，或中性的黄/白色）、小数点位数等。
（Vic3）可以定义AI对原子修正的价值评估。

本地化

本地化文件的冲突逻辑是先抢占后。

（Jomini）本地化和GUI使用相同引擎。
（EU4）在Clausewitz代码中引用本地化键可以加双引号"也可以不加。

本地化的嵌套引用（Jomini）（EU4部分参考）

在本地化的值中使用`$loc_key$`可以引用该本地化键的值，如：

```
loc_key_1: "123"
loc_key_2: "$loc_key_1$456"
```

- 则loc_key_2将显示为123456
- 如果loc_key_1不存在，则loc_key_2将显示为loc_key_1456
  （CK3，其他游戏应该也有）如果嵌套形成循环，文本将不能正常显示，转而显示MAX_RECURSIVE_DEPTH（最大递归深度）。

本地化显示格式

通过使用#<mode> <content>#!的格式，可以设置文本的显示样式，具体可用格式参考原版文件。
#G 这里是绿字#!这里是默认颜色 （非Jomini）使用§而非#
通过使用$VALUE|0P%$这样的格式，可以对数值的显示格式作出编辑，多种（功能上不互斥的，你不能要求一个数字既红又绿）描述符可以共存。
其中数字0代表显示0位小数，P代表按增益显示颜色（正值为绿负值为红），%代表显示为百分比。

本地化的模块化 （Jomini）

本地化是可嵌套的，即一个desc既可以指向一个本地化键，也可以成为一个代码块：

```
desc = {
    desc = desc0
    first_valid = {
        triggered_desc = {
            trigger = {
                trigger1
            }
            desc = desc1
        }
        desc = desc2
    }
}
```

上文的引用将根据trigger1的情况显示为desc0+desc1或desc0+desc2 此类代码的关键字包括：

- desc：最基本的本地化语句，如前所述，其右值可以是一个本地化键，也可以是一个更加复杂的代码块
  注意事件和其他一些文件中最外层的desc和模块化本地化中的desc不完全是相同概念，许多本地化不以desc标志。
- triggered_desc：包含一个trigger代码块和一个desc代码块或语句，当满足条件时显示desc中的内容，否则不显示。
- first_valid：选择其子代第一个应当显示的triggered_desc，其他几种代码块也可以写入，但都会被视为是无条件显示的，因此只用于最后一项的缺省。
- random_valid：类似first_valid，但在所有满足条件的子代中随机选择。
  逻辑上，desc外的三种代码块都可以等价于一个desc = {}代码块。
  以上几种代码块可以任意合理方式嵌套得到复杂的模块化显示逻辑。

自定义本地化 （EU4~？）

不确定此功能是否支持模块化本地化。

- （Jomini）位于`common/customizable_localzation`，冲突逻辑为后覆盖前
- （EU4）位于`customizable_localzation`，冲突逻辑为先抢占后
- （HOI4）位于`common/scripted_localisation`，冲突逻辑未测试
  群星应该也有

```
# （Jomini）
ClocKey = {
    type = <scope_type> # 如 character
    random_valid = yes  # 随机选择，默认为否，即选择第一项符合条件的

    text = {
        setup_scope = { #  （CK3，不确定Vic3是否支持）可选，在判断条件前执行，用于保存作用域并在条件和本地化中使用
            <effect>
        }
        trigger = {}
        localization_key = loc_key_2
        fallback = yes # 可选，当所有项目的条件都不满足时选择此项，应该只能有一个。
    }
    text = {
        localization_key = loc_key_1
    }
}
# （EU4）注意是localisation_key而非localization_key
defined_text = {
    name = ClocKey
    text = {
        localisation_key = loc_key_1
        trigger = {}
    }
    text = {
        localisation_key = loc_key_2
        trigger = {}
    }
}
```

这些自定义本地化可以在本地化值中通过Jomini代码调用，关于Jomini代码的知识见后文Jomini代码一节：

- EU4: `[scope.ClocKey]`
- CK3: `[scope.Custom('ClocKey')]`
- Vic3: `[scope.GetCustom('ClocKey')]`

（CK3）多作用域自定义本地化：
用于显示两个对象之间的关系或互动等。
在自定义本地化中以`scope:second`访问第二个作用域，相应的本地化键也使用这个保存作用域。
调用时使用`[scope.Custom2('ClocKey', scope2)]`。

分支本地化 - 本地化的选择

见于EU4的事件描述、CK3的事件选项等处。
与本地化的模块化一定相似处，相当于处于一个隐含的`first_valid`下。
使用分支本地化时，使用原本语句的左值，但改为代码块，如：

```
# CK3事件选项
# 不使用分支本地化
option = {
    name = name_loc_1
}
# 使用分支本地化
option = {
    name = {
        trigger = {}
        text = name_loc_1
    }
    name = {
        trigger = {}
        text = name_loc_2
    }
}
```

需要注意，使用此逻辑的不同游戏内容会使用不同的关键字，如EU4的事件描述在子项中仍然使用`desc`而不是`text`

条件与指令的本地化 （Jomini）

本*节*可能需要基*础*英*语语法*或*语*言学知*识*

（原子或经过`custom_description`自定义的）条件与指令的显示经过以下环节：

- 引擎根据条件或指令的内容抓取对应作用域等参数
- 游戏引擎根据指令的时态、阅读提示信息的（玩家的）作用域与相关作用域是否相同选择不同的时态和人称
- 根据选择的时态和人称选择本地化键
- 按照一般的本地化逻辑进行参数传递和本地化显示

Jomini以前的游戏使用内核生成的本地化键，因此只能修改最终的本地化一层，不能修改本地化键的选择逻辑。
条件与指令的本地化键选取逻辑分别位于`common/trigger_localization`和`common/effect_localization`。
其形式为

```
trigger/effect_loc_key = {
    global = LOC_KEY_GLOBAL
    first = LOC_KEY_FIRST
    third = LOC_KEY_THIRD
}
```

此处一共有3种人称：

- `global`，全局，在执行全局效果或者没有来源（主语）时使用
- `first`，第一人称，即阅读本地化的玩家的作用域作为主语时使用
- `third`，第三人称，即阅读本地化的玩家的作用域不作为主语时使用 此外，还有以下时态变体，加在人称后形成选择关键词如`third_not`：
- 无时态，直接使用人称项
- `_not`，取反，在条件中使用，条件被取反（NOT等）时使用，如果在不定关联作用域使用了比例或计数，即使取反也按正常时态显示
- `_neg`，负数，在指令中使用，当有数值型参数并且参数为负时使用
- `_past`，过去式，在指令中使用，用于事件的`immediate`等在玩家看到之前已经执行的指令
- `_past_neg`，负数过去式，即上两项的组合

当游戏调用时缺少所需人称/时态的组合会导致无法正常显示，在调试模式中可以以默认时态显示并以紫色字体警告。

条件与指令的本地化中，使用`$VALUE$`访问数值（若有），使用`<SCOPE>`如`CHARACTER`访问主语，使用`TARGET_<SCOPE>`访问宾语。

图形用户交互界面 / GUI （CK2~？）

进阶向，不推荐新手接触。
GUI文件的冲突逻辑是先抢占后。

GUI也可分为两个部分，即描述界面的部分，下称GUI代码，和执行功能的部分，下称Jomini代码。

GUI代码面向对象的程度远高于Clausewitz代码。

注意在这样的代码中不能直接使用数字，因为游戏将输入全部作为字符串解析，必须使用`(int32)1`这样的方式来传递整型值~~，有时要用`(float)1`转为浮点型~~。

GUI文件的扩展名是`.gui`，（Jomini）但自定义控件的目录文件仍为`.txt`

- （Jomini）GUI文件位于`gui`，本地化和GUI使用相同引擎。
- （非Jomini）GUI文件位于`interface`，需要在`.gfx`文件中注册`spriteType`来访问图片资源，且自定义空间相对较少。一部分语法与Jomini相似。
  非Jomini GUI实际上是将大部分功能内核封装起来，而Jomini GUI将这些功能也暴露出来允许修改，**本节基本只直接讲Jomini GUI**，一部分内容可套用在非Jomini GUI上。

内核窗口 / hard window

由内核调取的窗口，实际上是根据窗口的`name`属性调用，无需在意文件名。
（Jomini）一般以`window_`开头，全小写下划线分词
（非Jomini）一般以`view`结尾，全小写不分词~~P社你是真该死啊~~
建议模组新增文件避开原版命名习惯以示区分。

控件基本概念 / widget

关于UI的绘制，建议参考CK3官方Wiki（英文），本文暂不作全面翻译。
GUI的逻辑是层级的，即控件拥有子控件，如此形成整个界面。

控件的排列逻辑有横向和纵向两种，典型代表如`hbox`和`vbox`，即水平horizontal和垂直vertical的box，这两个单词在GUI中很常用。

控件的布局属性

每个控件都是一个面向对象逻辑意义上的对象，拥有许多属性，为避免混淆，下文一般不以对象称呼控件。

名称 / name

- 许多控件的名称是无必要的，但从Clausewitz脚本（`.txt`文件）调用GUI控件则必须通过名称访问。
- 在同一个视图中调用/定义同名的控件会发生冲突，只有前一个会显示，并且会报错。

可见性 / visible

- 最重要的属性莫过于可见性`visible`。
- 通常和Jomini代码联合使用，见下文Jomini代码一节。
- `visible`返回真才会显示，如果父级控件具有`ignoreinvisible = yes`属性，则不可见的子控件不会占据布局空间。
- 一个控件**只能拥有一个**`visible`属性。

文本 / text

`text = "loc_key"` 也可调用Jomini代码动态选择本地化。
如果整条文本内容都不需要直接调用文字和语言符号，如只由空格等不属于特定语言的字符和Jomini代码构成，用`raw_text`代替`text`，则不需要经过本地化键值匹配。

大小 / size

```
size = { x y }
```

- 其中x、y是x轴和y轴的整型或百分比坐标，-1代表不指定，自动扩展到父级控件该轴大小，百分比代表占父级控件大小的比例
- 也可以使用Jomini代码返回大小对象

位置 / position

- `parentanchor = hcenter|vcenter` 基点位于父级控件的何处
  - 分横轴和纵轴，用`|`隔开，可只写其一，另一项则取默认值。
  - 选项包括:
    - `center`: 在两个轴都取中间位置
    - `hcenter`, `vcenter`: 依次代表水平轴和垂直轴的中间位置
    - `left`, `right`, `top`, `bottom`: 依次代表左右上下
- `position = { x y }` 控件相对`parentanchor`基点的位置偏移
  - 其中x、y是x轴和y轴的整型或百分比坐标，负数代表向左/上偏移，百分比代表偏移量占父级控件大小的比例
  - 也可以使用Jomini代码`[GetGuiPositionFromPercentCoordinates( PdxGuiWidget.Self, x, y)]`返回位置对象，其中x、y需要转化为浮点型

模板 / templates

模板用于套用控件属性。

```
# 定义
template template_name {
    <property>
}
# 使用
hbox = {
    using = template_name
}
```

如此，则将template_name中的属性套用到了这个hbox中。
是多个模组修改同一个界面时的兼容性选择，但需要注意，由于一个控件只能拥有一个visible属性，而套多层元素又会导致ignoreinvisible属性无法正常生效，如果多个模组都需要修改visible，最好使用脚本GUI和宏的组合来实现。

控件的继承、多态

可以直接调用已经写好的控件，类似于前面提到的封装条件和指令。
同样，控件也可以嵌套引用，一个控件可以在已有的自定义控件的基础上再作修改。
通过block和blockoverride可以方便地覆盖一部分属性。

```
# 声明
types random_types_name {
    type type_name = widget {
        block "visible" {
            visible = yes
        }
    }
}
# 调用
# 这两种写法都可覆盖visible属性，但后者可以在调用的控件的子控件中覆盖属性
# 按前者的写法会再增加一个新的子控件而非覆盖原有子控件的属性
# 如果原控件中有多个同名block，那么blockoverride会同时覆盖所有同名block
type_name = {
    visible = no
    blockoverride "visible" {
        visible = no
    }
}
```

非Jomini游戏的脚本GUI / scripted_gui before Jomini （HOI4~群星）

HOI4和群星的脚本GUI与GUI耦合度较大，是真正意义上的脚本GUI。

Jomini代码

进阶向，不推荐新手接触基础本地化以外的内容。
自Jomini引擎起，GUI的非布局部分和本地化脚本使用了相同的语法，且功能更加多样化，本文姑且称作Jomini代码，但部分逻辑在Jomini前的游戏已在使用。
Jomini代码同样具有条件与指令之分，并且调用条件、指令或对象等（的最外层）均需要以`"[ ]"`的形式包裹。
建议自行搜索学习面向对象基础知识中的类、对象、方法等概念后再阅读本节。

Jomini逻辑运算

只支持与`And(,)`、或`Or(,)`、非`Not()`，本质上是函数。
注意与Clausewitz引擎不同，Jomini的与、或均能且只能传入2个参数，多个条件必须嵌套。

Jomini对象

本地化、GUI和Jomini代码中的“对象”一词特指这套系统中的对象，与作用域相区分~~其实这个对象字面上也是作用域，为了避免混淆刻意区分译名~~。
在调用本地化的Clausewitz脚本中，本地化键同样属于一个作用域（或特殊作用域全局Global），而在真正运行本地化时，作用域需要经过一定的转换才能成为本地化引擎能识别的对象。

（EU4）Jomini对象的语法非常简单，不需要进行对象格式转化，但本地化中的对象不一定是Clausewitz脚本支持的作用域/对象，如统治者`Monarch`在本地化中是一个对象，但在Clausewitz脚本中完全依附于国家存在。

Jomini方法

方法是通过对象调用的函数，方法总是属于一个对象类别，从对象调用。

使用`[]`标识Jomini方法，不同语言的细节有所不同，但都遵循以下逻辑：

- 所有方法使用大驼峰命名
- 以`.`访问对象的方法
- 可以接受保存作用域
- 若有参数，用`()`包括其参数，用`,`分隔各个参数（Jomini）

自定义本地化也是一种Jomini方法。

注意： （CK3）Jomini方法中的`XXXXOrMe`在当前作用域是当前玩家时显示的是第二人称的“你”，而非真正第一人称的“我”。

引用保存作用域

本地化遵从作用域逻辑，但不同游戏的具体语法有很大差异。
如在CK3中，`[saved_scope.GetFirstName]`代表显示先前保存作用域`saved_scope`的名字（不含姓氏）
（非Jomini）使用和CK3一样的调用方式。
（Vic3）使用`SCOPE.s<scope_type>('saved_scope')`而非`saved_scope`，如`[SCOPE.sCountry('saved_scope').GetName]`
Vic3的语法更为严格，但相关对象在CK3的文档中也有相同出现，部分写法是通用的，即CK3内置了这些代码糖。

引用动态作用域

由游戏引擎选取调用的本地化中有时会以动态作用域的形式传入作用域，比如前文提到的条件与指令的本地化。

- （EU4）引用动态作用域使用大驼峰
- （Jomini）引用动态作用域全词大写

引用特殊作用域 （Jomini）

实际上都是无参函数。

- `EmptyScope`返回一个空作用域。
- `GetPlayer`返回阅读当前内容的玩家。

Jomini对象的作用域化 / MakeScope （Jomini）

Jomini对象的内部逻辑比较复杂，建议参考输出文档和原版文件阅读。
所有调用的对象都需要进行作用域化，但动态作用域由引擎自动处理此步骤。
非动态作用域提取变量/脚本值相关内容时需要在对象后调用前增加`MakeScope`，如
`[attacker.MakeScope.ScriptValue('number_of_glory_hound_vassals')|V0]`

Jomini对象的类别转换 （Jomini）

- CK3
  - 由作用域型变量和动态作用域引出的Jomini对象必须进行类别转换，保存作用域则不需要
  - 对象转换的方式是在作用域后接`Char`等对应方法，可在`logs\data_types\data_types_script.txt`中查找（参见控制台一节）。
- Vic3
  - 所有由Clausewitz脚本引出的作用域都必须经过类别转换才能被Jomini引擎识别
  - 对象转换的方式是在作用域后接`GetCharacter`等方法，相比CK3更具一致性，可在`logs\data_types\data_types_script.txt`中查找（参见控制台一节）。

引用变量

- EU4: `scope.var_name`，如`[Root.var.GetValue]`
- Jomini: `scope.Var('var_name')`

引用脚本值 （Jomini）

同变量，但用`ScriptValue`代替`Var`。

控件的非布局属性

部分概念可以直接用于本地化

点击 / onclick

GUI执行脚本代码的手段，即点击该控件后执行的功能，一个控件可以有多个`onclick`属性，依次执行。

数据上下文 / datacontext

~~好烂的译名~~以下称为上下文对象。
类似于保存作用域，但是匿名的。
每个控件自动继承父级控件在上文声明的上下文对象列表，理论上控件可以在每个对象类型都存储一个上下文对象，调用时引擎根据所需对象的类型自动选取。
由控件引出的本地化中可以像在控件中一样访问上下文对象。

数据模型 / datamodel

~~更烂的译名~~
文档见`logs/data_types/data_types_common.txt`
严格来说是一种对象数组，可以通过下标取子集。
但是，数据模型**无法单取一个元素**，只能取出子集或遍历。
数据模型的遍历如下，注意`item`中只能有一个子控件，如果需要多个，必须使用容器包裹起来。

```
# 遍历
dynamicgridbox = {
    datamodel = "[datamodel]"
    # 依次访问datamodel中每一个元素
    item = {
    }
}
# 截取
DataModelLast(datamodel, '(int32)1')
```

在模组中最常用的数据模型是`GetList`，即读取前文介绍的列表。

Jomini中的脚本GUI / Jomini scripted_gui

脚本GUI不是GUI，是Clausewitz脚本
Jomini的脚本GUI将GUI部分完全剥离到了GUI中，脚本GUI仅用于描述自定义GUI的显示、执行等。
脚本GUI位于`common/scripted_gui`
（Jomini）脚本GUI主要以下部分组成，详细可参考Vic3下此目录的说明文档，但不确定是否所有代码块都在CK3生效：

```
sgui_name = {
    scope = <character>/<none>/...
    saved_scopes = { scope } # 列出所要求的传入保存作用域，但实测不写此块也可正常在下方三个代码块中正常访问
    is_shown = {}
    is_valid = {}
    effect = {}
    # 省略了关于文本提示和AI行为的代码块
}
```

其主要作用是在GUI中自定义可见性`IsShown`、可用性`IsValid`和执行效果`Execute`，调用格式相仿，样例如下。

```
# Jomini代码
datacontext = "[GetScriptedGui('sgui_name')]"
visible = "[ScriptedGui.IsShown( GuiScope.SetRoot( GetPlayer.MakeScope ).End)]"
# 等价于
visible = "[GetScriptedGui('sgui_name').IsShown( GuiScope.SetRoot(
GetPlayer.MakeScope ).End)]"
# 如需传入保存作用域，写法如下，可有任意数量的`AddScope`：
onclick = "[ScriptedGui.Execute( GuiScope.SetRoot( <对象1>.MakeScope ).AddScope(
'scope', <对象2>.MakeScope ).End ) ]"
```

需要注意

- 如果SetRoot指向的对象的类型与脚本GUI声明接受的作用域类型不匹配（如前文的EmptyScope空对象不属于单一对象类型），则会在**每次**访问时报错，但以AddScope传入的保存作用域则不在此限，因此用于检测对象或涉及肖像等情况时，建议不SetRoot，直接AddScope。
- is_valid条件会限制effect的执行，即即使GUI从onclick等途径触发effect，若传入作用域不满足is_valid也不会执行。

状态 / state

可以用于执行循环指令或条件指令，通过与脚本GUI（见下文）的联动实现tick级的检测。

```
state = {
    name = _show
    on_start = "[ScriptedGui.Execute( GuiScope.SetRoot( GetPlayer.MakeScope ).End
)]"
}
```

此代码将在控件由不可见转为可见时执行"[ScriptedGui.Execute(GuiScope.SetRoot(GetPlayer.MakeScope).End)]"这一指令，注意这里用到了数据上下文。
可以想到，另一个内核的状态名是_hide，在由可见转为不可见时执行。

```
state = {
    name = "b"
    next = "a"
    trigger_on_create = yes
    duration = 7
}
```

这是一个自定义状态，在该状态保持duration（支持小数）时间（单位不明）后自动转入next所指的状态，可以实现循环。
trigger_on_create属性使该状态在控件展示时激活，不同于_show状态，始终显示的控件也会触发这样的状态。
也可以通过高频循环来执行Clausewitz代码无法执行而Jomini代码可以执行的一些操作。
此外，trigger_when也可以作为条件判断来唤起状态。

```
state = {
    name = "name"
    trigger_when = "[ScriptedGui.IsValid( GuiScope.SetRoot( GetPlayer.MakeScope
).End )]"
}
```

通过状态指令配合脚本GUI，可以实现Clausewitz代码无法即时执行的检查行为，如读档自动将旧版本模组的存档转化为新版本模组的格式。
注意P社没有做线程锁，Jomini线程比Clausewitz线程快，通常会导致执行两次。一个简单的解决方法是指向一个事件。

宏 / data_binding (marco)

本质还是字符串替换，可以参考百科的解释。
由于Jominin代码必须写在一行，使用宏可以有效改善Jomini代码的可读性。
位于`data_binding`，后缀名为`.txt`。

```
# 参数解析
macro = {
    description = "描述，游戏内的宏列表可以被script_docs输出，这里的文本会作为介绍一同
输出"
    definition = "调用名(可选，用逗号分隔的参数，无参数无需括号)"
    replace_with = "替换Jomini代码，所有参数都要在此使用至少一次"
}
# 样例
macro = {
    description = "Calculates the named script value with target character as root
scope"
    definition = "CalculateScriptValue(Target,ScriptValueName)"
    replace_with = "Target.MakeScope().ScriptValue( ScriptValueName )"
}
# 调用
raw_text = "[CalculateScriptValue( AIWatchWindow.GetCharacter, 'script_name'
)|V2]"
# 等价于
raw_text = "[AIWatchWindow.GetCharacter.MakeScope().ScriptValue( 'script_name'
)|V2]"
```

其他

报错

在`文档/Paradox Interactive/游戏名/logs`下，游戏会自动记录error.log，即报错记录。
通常，此记录对排查游戏闪退没有帮助，因为闪退多是内存越界引起，程序没有机会写下记录便已被操作系统杀死。
但对modder而言，报错记录有助于排查问题，建议常读。

模组设计

模组间交互

模组识别

一般来说，需要制作适配和联动内容时，不同模组之间需要互相识别。
识别载入有两种策略：

- 初始化游戏时设置全局变量
- 设置一个scripted_trigger

后者需要占位符以免被频繁触发空条件报错。

接口化

为减少兼容补丁的使用，可以在各模组都修改的文件或涉及联动的内容中使用其他模组的封装内容，通常是封装条件和指令，有时也可以使用action和脚本GUI。
这几种功能的空访问兼容性如下：

- 不存在的封装条件和指令在游戏初始化时和每一次被调用时产生一次报错，条件视为恒为真（不存在）。
- 不存在的action在游戏初始化时产生一次报错，调用未确认。
- 不存在的脚本GUI不产生任何报错，与其相关的**整行**语句被忽略，这意味着如果一条不存在的脚本GUI检查与一条恒为假的检查条件被`And(,)`包括，这条检查语句在逻辑上本应恒为假，但却会被游戏引擎按恒为真处理。
- 不存在的GUI控件在游戏初始化时和每一次被调用时产生一次报错。
- 不存在的本地化在游戏初始化时产生一次报错，调用未确认。
- 不存在的模板不会产生任何报错，而只在`gui_warning.log`中记录。

与之相对的，每条/次重复的内容只会在游戏初始化时产生一次报错。
综上，建议不能确保上述其他模组中定义的内容存在时，在对应目录创建占位符文件，如`000_<mod_prefix>_placeholder_<filetype>.txt`（GUI和本地化采取先抢占，因此以`zzz_`开头，扩展名分别为`.gui`和`.yml`，**脚本GUI是脚本，不是GUI**），以确保其存在。
封装条件的占位符应当清楚地写出在没有对应模组时的默认逻辑是恒为真（可直接留空）还是恒为假`always = no` 关于带有参数的封装条件和指令在参数问题上的逻辑，请查阅前文该节。

兼容性排查

（CK3）访问的修正`modifier`若不存在，会导致游戏启动**概率性**发生崩溃
在Jomini中（GUI等）访问不存在的内容（脚本GUI、宏）会使整行条件（跨越逻辑运算符）恒为真。
在Clausewitz中访问不存在的内容会使该条件恒为真，因此可以以`NAND = { always = yes trigger_in_other_mod = yes }`的形式安全判断——如果那个模组没有载入，则这条条件恒为假。

性能优化

和其他语言一样，Clausewitz引擎会在条件确认成功或失败后（如`AND`中有一条为假，或`OR`中有一条为真）跳过后续代码检查，这意味着你应该总是将最简单而容易筛掉最多情况的条件摆在前面。
（Jomini）由于Jomini与Clausewitz的异步问题，GUI层会以**系统时钟级的频率**不断访问UI上出现的对象，如果

条件涉及遍历就会导致严重的卡顿掉帧。因此所有涉及直接展示的提示（包括但不限于`is_shown`、`is_valid`、`effect`三大类代码块）都应该尽可能避免反复检查，尤其是`is_shown`系，因为不点开其界面也需要判定，一旦打开决议列表或类似情景便会开始不断计算。这种情况下最好使用脚本值封装其条件，利用脚本值的懒更新特性避免浪费资源制造卡顿。