宾语自动化引导教程

前言

相信你已经玩到了一些城夺地图，并且想要自己制作一个城夺地图。但是，当你了解了城夺地图在制作宾语方面的繁琐之时，可能心里已经在打退堂鼓了。一个能玩的城夺地图至少需要建筑刷新和兵力刷新两项内容。建筑刷新需要 2 个宾语，分别是 unitAdd 和 unitDetect。兵力刷新需要 1 个宾语。如此，玩家能获得兵力进行对战，并拥有建筑来同化兵力，实现最基础的城市争夺玩法。你需要大量铺设这些宾语，才能构建起一个能玩的城夺地图。可以发现，相对于许多原版地图，城夺地图在宾语上的工作量要显著增大。

在这里，我们事先约定，城夺地图是无建造者的刷兵地图，或者有建造因素，但受到限制的一类刷兵地图。 原版地图是指除城夺地图之外的铁锈对战地图（不包括mod地图，也不包括一些非对战的特殊玩法，比如冒险地图，自走棋等）

然而，如果想要一张城夺地图可以正常游玩，最基础的是胜负判断。对于原版地图来说（有建造者的地图），胜负判断都是自动的。然而对于一张城夺地图来说，地图上没有建造者，那么胜负就无从判断。这时，通常的胜负判断就是在场外放置玩家的建造单位（但是不能用），然后检测场内建筑的归属，来删除玩家的建造单位，胜负就此决定。

城夺地图中默认城市是补给站，需要在上面添加城市名字。国家、地区、政权等需要额外添加名称，这称为动态国名。动态国名往往也需要检测场内建筑的归属。不同势力占领同一地区的名字有时候并不一样，比如“乌克兰专员辖区”和“苏维埃乌克兰社会主义共和国”。兵力的刷新有时也需要检测建筑归属。敌国占领本国领土，为了符合实际，往往无法给敌国刷兵。己方的兵力生产有时也需要因为敌方的占领而撤退到后方。字幕（事件）的发送也需要检测建筑归属，例如某某占领了某地的字幕。

然而，检测建筑归属由于铁锈原版宾语系统的固有缺陷，一次只能检测一个 team（队伍号）。因此，想要完整检测一个建筑的归属，需要写玩家数目的检测宾语，这是非常麻烦的事情。还有动态国名的调整（颜色，大小，文本），字幕的调整（出现时间，检测，内容），兵力刷新的调整，建筑分布和归属的调整，胜负检测的调整等等等。这些都需要大量宾语，需要大量时间调整，才能真正做好。地图作者的大部分时间都用在调整宾语的细枝末节上，因为想要实现一个小功能都需要对宾语进行许多复杂的调整。此外，经过城夺作者的长期实践，发现了一些铁锈宾语的“特性”以及获取了许多制作经验，这些对于刚入门的城夺地图作者来说，需要花费更多时间翻阅相关地图来获取这些经验，这是不友好的。

正是因为城夺制作有那么多的问题，尤其是在宾语方面。我制作了这个自动化程序，来帮助简化以上提到的所有步骤。一些城夺地图还有独特的地块风格，但是地块并不是城夺制作的核心问题，当然合适的地块能带来更好的游玩体验。那么接下来进入正文，我将逐渐为您介绍城夺地图制作的基本方法，并介绍宾语自动化程序是怎样使用的。

前置知识

首先，我们需要理解铁锈地图是什么。铁锈地图是一个 xml 格式的文件（后缀为.tmx），本质上是一颗文本树。铁锈地图有三大内容，地块集 (tileset)，底图 (layer) 和宾语 (objectgroup)。地块集为底图提供图片信息和属性信息，本文不多做介绍。底图对于几乎所有铁锈地图来说都是核心。因此挑选一张合适的底图做图至关重要，当然也可以自行绘制底图。宾语是本文的重心。在这里，宾语自动化仅仅为宾语 (objectgroup) 中的触发器 (Triggers) 服务，而不处理单位宾语 (unitObject)，请务必注意。

在进行城夺地图制作之前，必须阅读过铁锈地图制作的基础教程，对铁锈地图制作有最基本的了解。获取有关教程和帮助，请加入 qq 群铁锈地图制作劳动力压榨俱乐部 (**933066487**)，或者在本群 (**699981990**) 查看有关教程。前置教程包括 notTiled 做地图基础教程，触发层宾语教程。你可以在群文件找到这些教程。请确保自己至少粗略读过一遍。如果您是熟练的地图制作者，可以忽视这些要求。

如果您看完了以上内容，并且愿意参与进城夺地图制作，那么可以继续阅读了。接下来的教程仅关于城夺图的宾语如何写以及宾语自动化如何使用，会默认读者对宾语有基础认知。

开始自动化（建筑）

城夺地图的核心是建筑刷新。让我们以建筑刷新为例，开始了解自动化程序是如何运行的。在引导教程中出现的所有有效普通宾语、info 宾语和标记宾语均会在宾语自动化引导教程地图示例中出现。并且也会在对应位置出现坐标。关于地图示例格式，请参考地图示例格式，实际使用的地图示例和教程示例均遵循这样的格式。

如果想用常规宾语完成一个建筑，至少需要两个宾语，比如：

```
unitAdd //刷新建筑
    spawnUnits : "supplyDepot*1" //生成一个供应站(塔防同理)
    team : "-1" //队伍中立，允许被玩家同化
    activatedBy : "id123" //被哪个 id 激活
    warmup : "20s" //宾语激活后延迟20s后才生成
    resetActvationAfter : "20s" //重置宾语状态，时间较短即可

unitDetect //建筑刷新后，抑制建筑刷新
    id : "id123" //检测成功后使该 id 激活
    unitType : "supplyDepot" //检测的单位类型
    maxUnits : "0" //数量为 0 时检测成功
    resetActivationAfter : "20s" //每过 20 秒检测一次
```

对于普通宾语来说，我们约定，开头为宾语的类型，后面紧跟该普通宾语的在示例中的坐标。换行后是普通宾语的属性。不要把双引号也加上。"//"是注释符号，后面的文本请无视。

但是，有时候，建筑上要有名字，这时需要添加：

```
mapText //建筑名字或者城市名
    text : "维希" //地图内显示“维希”名字
    textColor : "#8EA69C" //颜色十六进制码，默认白色
    textSize : "6" //文字大小，参考区间 5~9
```

同样的，有时候，建筑需要初始单位来进行同化。

```
unitAdd //刷新建筑
    spawnUnits : "supplyDepot*1" //生成一个供应站（塔防同理）
    team : "0" //初始单位队伍
```

对于一张地图来说，同时可能出现很多建筑，上面的宾语的大部分参数都是相同的，而只有少部分参数不同，比如城市的名字，比如建筑的初始队伍等。

如果想要对建筑进行自动化，我们应当如何做呢？宾语自动化的核心原理是使用外部程序处理地图文件，从而实现自动化。既然我们想要放置的建筑有如此多的相同参数，那么我们自然而然能想到，将建筑的共同信息提取出来，专门储存起来，然后在每一个建筑处再获取一部分具体的信息，最后产生具体建筑宾语。

由此，产生了宾语自动化的两大概念，info 宾语和标记宾语。info 宾语是提取了自动化共同特征的宾语。标记宾语是产生具体结果的宾语。标记宾语从 info 宾语中提取了共同特征，也从自身参数获取了具体信息，最后生成了一些具体宾语，从而实现具体功能。所以说，想要通过自动化产生建筑，需要先写一个建筑 info 宾语，再写许多标记宾语。每一个标记宾语都将产生具体宾语来实现建筑这个功能。

然而，除了与自动化相关的宾语之外，还有一些正常使用的宾语，我们不希望对它们产生变动。所以，info 宾语、标记宾语和正常宾语需要区分开来。最后，我们采取的策略是，与自动化相关的宾语的类型统统不填，从而与正常宾语区分。info 宾语和标记宾语的识别则是依靠其名字的前缀，后面会具体说明。info 宾语需要名字进行识别，还有属性来规定该 info 的信息（填写一个 info，需要更加详细）。标记宾语只需要填写名字，不需要填写属性（填写许多标记宾语，需要更加方便）。

宾语自动化参数说明包含当前版本中宾语自动化的所有参数和介绍，最为全面完整。

此外，还有一份宾语自动化地图示例，这是一份地图文件，可供实际作图是复制粘贴和参考。还有一份该地图示例用法的具体说明，宾语自动化示例说明。

接下来，我们将具体讲解。

我们可以先查看 building_info 的内容。请查看参数说明 inadd_info的 inadd_info 到 building_info 的内容。

```
## inadd_info

初始单位添加宾语，是附属宾语，且无参数。

### inadd_info必选参数

prefix：其他info宾语引用标记。（[info宾语默认参数](#info宾语中基本参数介绍)）
```

```
8
9  inaddteam：当isinadd为是时必选，表示单位添加的队伍。
10
11 ### inadd_info可选参数
12
13 isinadd：可选，默认为是，在building_info中默认为否，是否添加城市的初始
       刷新。
14
15 inaddwarmup：当isinadd为是时可选，默认为{addWarmup}，表示单位添加的
       warmup。为0s或addWarmup不存在时，不产生warmup属性。
16
17 inaddunit：当isinadd为是时可选，默认{aunit}，表示单位添加的类型。
18
19 inaddspawnnum：当isinadd为是时可选，默认为1，表示单位添加的数量。
20
21 inaddisshowOnMap：当isinadd为是时可选，默认为否，启用时，初始城市生成时
       小地图显示。
22
23 inaddname：当isinadd为是时可选，默认为""，建筑初始添加宾语名字。
24
25 inaddoffset：当isinadd为是时可选，数字数组，默认为"0 0"，建筑初始添加宾
       语偏移。
26
27 inaddoffsetsize：当isinadd为是时可选，数字数组，默认为"0 0"，建筑初始添
       加宾语大小改变。
28
29 inaddisinitialunit：可选，默认为否。启用时，建筑刷新将使用"unit"，而不
       是"spawnUnits"。并且，除去"unit"和"team"以外的选项将不会出现。
30
31 ## mtext_info
32
33 建筑文本显示，是附属宾语，且无参数。
34
35 ### mtext_info必选参数
36
37 mtext：当ismtext为是时必选，表示城市文本的内容。
38
39 mcolor：当ismtext为是时必选，表示城市文本的颜色。
40
41 mtextsize：当ismtext为是时必选，表示城市文本的字号。
42
43 ### mtext_info可选参数
44
45 ismtext：可选，默认为是，在building_info/idcheck_info中默认为否，是否启
       用该宾语。
46
47 mname：当ismtext为是时可选，默认为""，文本宾语名字。
48
49 moffset：当ismtext为是时可选，数字数组，默认为"0 0"，文本宾语偏移。
50
51 moffsetsize：当ismtext为是时可选，数字数组，默认为"0 0"，文本宾语大小改
```

```
    变。
52
53  ## building_info
54
55  这是一个建筑添加info宾语。可以生成自动刷新建筑的宾语。
56  可以添加初始城市（inadd），确定城市初始阵营，附属宾语为inadd_info。城市可以提供文本生成（mtext），显示城市名称，附属宾语为mtext_info。因此，inadd_info和mtext_info的所有必填参数和选填参数都在building_info中存在。可进行时间修正。
57
58  ### building_info必填参数
59
60  prefix：标记宾语引用前缀。（[info宾语默认参数](#info宾语中基本参数介绍)）
61
62  idprefix: 表示所使用的id前缀，城市id在idprefix后将自动按照1,2...顺序延伸，请确保其他id没有此前缀。
63
64  detectReset: 建筑检测的resetActivationAfter。
65
66  aunit：建筑单位类型。
67
68  ### building_info可选参数
69
70  isprefixseg: 可选，默认为否。（[info宾语默认参数](#info宾语中基本参数介绍)）
71
72  args：可选，默认没有，必填参数。（[info宾语默认参数](#info宾语中基本参数介绍)）
73
74  opargs：可选，默认没有，可选参数。（[info宾语默认参数](#info宾语中基本参数介绍)）
75
76  cite_name：可选，默认没有，标志宾语标记。（[info宾语默认参数](#info宾语中基本参数介绍)）
77
78  brace：可选，默认没有，外部引用翻译列表。（[info宾语默认参数](#info宾语中基本参数介绍)）
79
80  addWarmup：可选，建筑添加的warmup。
81
82  addReset：可选，建筑添加的resetActivationAfter。
83
84  spawnnum：可选，默认为1，建筑添加的数量。
85
86  minUnits：可选，默认没有，建筑检测的minUnits。
87
88  maxUnits：可选，默认为0，建筑检测的maxUnits。
89
90  team：可选，默认为-1，建筑添加的队伍。为-1时，建筑检测没有team。不为-1时，建筑检测有team。
```

```
91
92  isonlybuilding：可选，默认为否，是否启用onlybuilding来检测建筑，如果启
        用，则不使用unitType检测。
93
94  isshowOnMap：可选，默认为否，启用时，建筑生成时小地图显示。
95
96  acti：可选，字符串数组，默认没有。建筑添加宾语添加的额外activatedBy。
97
98  deacti：可选，字符串数组，默认没有。建筑添加宾语添加的额外deactivatedBy
        。
99
100 addname：可选，默认为""，建筑添加宾语名字。
101
102 addoffset：可选，数字数组，默认为"0 0"，建筑添加宾语偏移。
103
104 addoffsetsize：可选，数字数组，默认为"0 0"，建筑添加宾语大小改变。
105
106 detectname：可选，默认为""，建筑检测宾语名字。
107
108 detectoffset：可选，数字数组，默认为"-10 0"，建筑检测宾语偏移。
109
110 detectoffsetsize：可选，数字数组，默认为"20 0"，建筑检测宾语大小改变。
111
112 inadd_prefix：可选，默认不存在。将会导入对应inadd_info的数据，建立初始
        建筑。
113
114 mtext_prefix：可选，默认不存在。将会导入对应mtext_info的数据，生成建筑
        名字。
115
116 time_prefix：可选，默认不存在。将会导入对应time_info的数据，进行时间修
        正。
117
118 ### building_info可被引用的其他参数
119
120 idprefix0是建筑检测的id。
```

其中，building_info 的含义是，如果想要被识别为一个 building_info 宾语，那么该宾语名字的前缀必须是 building_info（后面可以任意添加）。

有的属性你可能认识，大概了解是做什么的。有的属性可能一头雾水。一些参数在许多 info（其他自动化模板）中也会出现，一些参数则是 building_info 独有的参数。它们有着各自不同的功能。

我们知道，标记宾语必须从 info 那里获得共同信息，才能具体产生宾语。二者之间是如何产生联系的呢？是通过 **prefix** 参数。

```
1   prefix：填写标记宾语的前缀（或者其他info宾语的引用），在标记宾语名称开
        头加入prefix即证明该标记宾语使用该info宾语的格式。
```

以建筑为例，我们设定两个宾语。这样前者被识别为 building_info 宾语，后者被识别为该 building_info 宾语的标记宾语。由于 **prefix** 是标记宾语识别的关键参数，因此所有 info 都有 **prefix** 这个参数。

为了保证识别的唯一性，不同的info的prefix不可以互为前缀（想一想为什么）。否则会报错，错误码22。

```
info : "building_info_ci"
    prefix : "ci" // 前缀

标记 : "ci"
```

对于自动化宾语来说，我们约定，info打头为info宾语，标记打头为标记宾语，打头后面是该宾语在示例中的坐标。后面再跟它们的名字，类型不填。如果是info宾语，后面还会跟上它们的属性。不要把双引号也加上。"//"是注释符号，后面的文本请无视。

对于建筑来说，想要实现对建筑刷新的控制需要一个 id 作为信息传递的中介。这个 id 具体是什么不重要，关键是不能与其他 id 重复。在这里，我们设定其只需要填写 id 的前缀 (**idprefix**)，之后每添加一个新建筑，新产生建筑的 id 都会是该 id 前缀后面加一个数字，并且保证不会与其他 id 发生重复。

请确保，普通宾语中不会出现该前缀+数字的id，否则仍然可能发生重复。 idprefix在不同的info中可以重复的，不同于prefix。

```
info : "building_info_ci"
    prefix : "ci" // 前缀
    idprefix : "ci" //  建筑id前缀

标记 : "ci"
```

接下来，我们将试图向 info 和标记宾语中添加参数。曾经，在 1.5 版本，该程序是将标记宾语中所需的参数固定下来的，这种模式比较僵化。然而，如果想要实现更灵活的自动化，需要灵活设置参数。

```
args: 声明标记宾语的必填参数。比如"aunit,str;num,str"，这一串表明标记宾语必须填入2个参数，第一个是代入aunit，第二个代入num。参数名后面目前支持str和bool格式。bool格式下，可以输入"true"或"false"。标记宾语在前缀之后添加supplyDepot.1意思是aunit为supplyDepot，num为1。如果info宾语的prefix为e，那么标记宾语将填esupplyDepot.1。如果isprefixseg标记了，那么标记宾语将填e.supplyDepot.1。

opargs: 声明标记宾语的选填参数。比如"u,aunit,str,supplyDepot;n,num,str,2"，这一串表明aunit和num是可以选填的（第一个参数必须只有一个字母）。想填写aunit为supplyDepot时需要输入",usupplyDepot"。想填写num为1时需要输入",n1"。注意，即使没有填写num，num默认为2。
如果格式为bool。那么，只要填写了该选项，不需要额外输入参数，认为是"true"。
*,d是默认存在的选项，加入即证明该标记宾语被打上了删除标记。*
*,D是默认存在的选项，加入即证明该标记宾语将会被彻底删除。*
```

在这里，大部分 info 都有 **args** 和 **opargs** 这两个参数，这两个参数具体决定了标记宾语书写参数的格式。**args** 决定了，哪一些参数是必须在标记宾语填写的。**opargs** 决定了，哪一些参数是可填可不填的，在标记宾语中是不确定的。

在标记宾语中，“.” 分割了所有 **args** 参数。“,” 分割了所有 **opargs** 参数。因此，标记宾语的格式是这样的” 前缀 {参

数 0}.{参数 1}...,{可选参数 0},{可选参数 1}... ”。必填参数的数量是固定有序的。可选参数是可填无序的。为了确定一个可选参数是哪一个，需要一个字母作为前缀确定，后面才是真正的参数。特别的，如果 **isprefixseg** :” true”，那么前缀和参数 0 之间也需要” .” 分隔。

如果"."遗漏或必填参数漏填，会导致错误，错误码6。

如此，我们确定 **inaddteam** 和 **mtext** 作为必填参数，**team** 作为选填参数，t 为选填参数的标记，**team** 默认为-1。这样，我们能灵活地确定哪一些信息是需要在 info 中就确定的共同信息，哪一些信息是需要在标记宾语中具体确定的信息。这些参数的具体含义将在后续讲解。

```
info : "building_info"
    prefix : "ci" // 前缀
    idprefix : "ci" // id前缀
    args : "inaddteam,str;mtext,str"
    opargs : "t,team,str,-1"

标记 : "ci0.莫斯科,t0" 或 "ci0.莫斯科"
```

或者

```
info : "building_info"
    prefix : "ci" // 前缀
    idprefix : "ci" // id前缀
    args : "inaddteam,str;mtext,str"
    opargs : "t,team,str,-1"
    isprefixseg : "true"

标记 : "ci.0.莫斯科,t0" 或 "ci.0.莫斯科"
```

info 宾语的默认参数讲解告一段落。读者可能发现还有两个默认参数 **cite_name** 和 **brace**。这对 building_info 暂时没有用处，将会在进阶教程介绍。现在回到 building_info 的具体参数讲解中来。

在 building_info 的必填参数还有两个，分别是 **detectReset** 和 **aunit**。前者规定了 unitDetect 的 resetActivationAfter，后者规定了建筑单位的类型。这些应当很好理解。

```
info : "building_info"
    prefix : "ci" // 前缀
    idprefix : "ci" // id前缀
    args : "inaddteam,str;mtext,str"
    opargs : "t,team,str,-1"
    detectReset : "20s" //检测reset
    aunit : "supplyDepot" //单位类型

标记 : "ci0.莫斯科,t0" 或 "ci0.莫斯科"
```

然而，一个建筑通常还需要一个可选参数 **addReset**。这是 unitAdd 的 resetActivationAfter。保证建筑可以持续刷新。

```
info : "building_info"
```

```
    prefix : "ci" // 前缀
    idprefix : "ci" // id前缀
    args : "inaddteam,str;mtext,str"
    opargs : "t,team,str,-1"
    detectReset : "20s" //检测reset
    aunit : "supplyDepot" //单位类型
    addReset : "20s" //添加reset

标记 : "ci0.莫斯科,t0" 或 "ci0.莫斯科"
```

我们现在具体讲解一下 **args** 和 **opargs** 里面的参数。首先我们需要明白，building_info 有两个“附属” info，它们是 inadd_info 和 mtext_info。所谓附属的意思是，这些 info 将不能自行产生标志宾语，它们的 **prefix** 不是给标记宾语用的，而是给 building_info 用的，分别在 **inadd_prefix** 和 **mtext_prefix**。如此，building_info 可以导入它的附属 info 来实现其功能。

出现附属宾语的原因正是作为建筑的两个附属功能。一个是单位队伍初始化，一个是建筑上可能有名字。两个功能可能存在也可能不存在，因此我们如此设置。

```
inadd_prefix：可选，默认不存在。将会导入对应inadd_info的数据，建立初始建筑。

mtext_prefix：可选，默认不存在。将会导入对应mtext_info的数据，生成建筑名字。
```

同时，building_info 同样可以自由修改 inadd_info 和 mtext_info 的参数，将会覆盖引用的附属 info 信息（附属 info 信息也可以没有）。在这里，我们需要启用 inadd。使得 **isinadd**：“true”。如此，我们就能理解 **inaddteam** 了，是建筑的初始队伍。同样，我们需要启用 mtext。使得 **ismtext**：“true”。如此，我们也能理解 **mtext**，这是建筑文本的内容。

注意，为与铁锈地图的team（从0开始）保持一致，宾语自动化的team也是从0开始的。

```
info : "building_info"
    prefix : "ci" // 前缀
    idprefix : "ci" // id前缀
    args : "inaddteam,str;mtext,str" // 必填参数列表，第一个为建筑的初始队伍，第二个为建筑名字
    opargs : "t,team,str,-1"
    detectReset : "20s" //检测reset
    aunit : "supplyDepot" //单位类型
    addReset : "20s" //添加reset
    isinadd : "true" // 启用建筑初始刷新
    ismtext : "true" // 启用建筑文本

标记 : "ci0.莫斯科,t0" 或 "ci0.莫斯科" // 莫斯科初始被1号玩家占领
```

还有一个可选参数 **team**。这是建筑的刷新队伍，是刷新为中立 (-1，可被同化)，还是刷新给特定玩家。这是可选参数是因为，大部分建筑都是需要同化的，可能少部分建筑需要给特定玩家刷新，这时需要添加可选参数。

```
info : "building_info"
    prefix : "ci" // 前缀
    idprefix : "ci" // id前缀
    args : "inaddteam,str;mtext,str" // 必填参数列表，第一个为建筑的初始队伍，第二个为建筑名字
    opargs : "t,team,str,-1" // 建筑刷新队伍
    detectReset : "20s" //检测reset
    aunit : "supplyDepot" //单位类型
    addReset : "20s" //添加reset
    isinadd : "true" // 启用建筑初始刷新
    ismtext : "true" // 启用建筑文本

标记 : "ci0.莫斯科" // 莫斯科初始被1号玩家占领
标记 : "ci0.莫斯科,t0" // 莫斯科初始被1号玩家占领，之后的刷新也一直刷给1号玩家。
```

还有一个参数值得注意，**isonlybuilding**。通常来说，补给站不用这个选项，而炮塔等可升级的建筑需要添加**isonlybuilding**：“true” 这一选项，这将拒绝使用 unitType 检测，而是使用 onlyBuildings 检测。如此，炮塔升级后也能被检测到，同时其他单位在炮塔的位置站住并不会抑制炮塔生成。坏处是，炮塔将不能与其他建筑重叠放置。

恭喜，我们共同完成了一个 info 并且书写了标志宾语，完成了宾语自动化的基础学习。

现在，请将以上的 info 宾语和标记宾语添加到一张地图上进行实验。（什么，不知道怎么做？请重新看前置知识，直到能做出来。）

然后，请查看您地图的文件路径。对于电脑来说，想必并不困难。对于手机来说，建议使用 mt 浏览器，找到您的地图，长按地图文件，点开属性，长按名称，可以看到地图路径已被复制。

怎么将 info 和标记宾语变成实际的宾语呢？如果你已经完成了软件安装步骤，那么请打开你的终端，输入

```
objectgroupauto "输入地图路径" -o "输出文件路径"
```

注意要对地图路径加双引号，这样路径中的空格不需要转义符处理。输出地图路径注意，尽量与输入地图路径不完全相同，这样自动化后的地图不会覆盖原地图。

如此，你可以在输出文件路径获得一张新地图，这张地图就是宾语自动化的结果。如果您不想在自动化后的地图看到 info 宾语和标记宾语，请加入 -D 选项，例如

```
objectgroupauto "输入地图路径" -o "输出文件路径" -D
```

-D 选项会彻底删除 info 宾语和标记宾语，因此是不可逆的，强烈建议使用 -o 选项输出到新文件路径中。

还有更多的宾语选项可供选择。可以这样查看帮助。还可以在命令行参数查看所有选项

```
objectgroupauto -h
```

现在，您已经可以开始使用宾语自动化了。在 inadd_info、mtext_info 和 building_info 中还有许多参数可以探索，试着查找并添加试试效果？在建筑自动化和刷兵宾语手写的情况下，您已经可以制作比如 1939 城夺这种类型的地图了，这是早期城夺地图的水平，可喜可贺。

进阶教程（队伍检测，字典）

相信您已经对宾语自动化有了最基础的了解，学会建筑自动化了。这已经能为你制作城夺地图提供最基础的帮助。

然而，正如前文所说，城夺地图制作还有许多其他内容，它们都可以实现自动化，那么让我们进入下一个重要部分，队伍检测。

所谓队伍检测，是一张地图除去建筑、刷兵之外，所需要的一项重要的基础设施。队伍检测可以用来判定胜负，控制城市名称，控制动态国名，控制剧情放送，控制刷兵等等，十分重要。然而队伍检测也非常繁琐，下面是例子

```
有几个玩家，出现几个unitDetect宾语

unitDetect:
    unitType : "supplyDepot" //检测单位
    minUnits : "1" // 最小数量，检测到即激活
    team : "4" // 5号同化了该城市时激活，每一个unitDetect宾语的team均不同
    id : "teamAcontrolcity123" //检测成功就激活该 id，不同unitDetect宾语同team相同 id
    alsoActivate : "teamAcontrolcity123" //多个检测，需要额外添加
    resetActivationAfter : 3s //每隔三秒检测一次
```

想要制作对一个建筑的队伍检测，必须制作与玩家数目相同的检测宾语。这需要花费很多精力。想象一张城夺地图，共有 100 个城市 10 个玩家，就需要 1000 个队伍检测宾语（如果全覆盖）。如果有 200 个城市 20 个玩家，就需要 4000 个队伍检测。这是十分沉重而机械的工作。

有关于队伍检测宾语自动化的 info 是 teamDetect_info。下面是该 info 的详细参数，您也可以在teamDetect_info参数中的 teamDetect 部分看到。

```
## teamDetect_info

这是一个队伍检测info宾语。可以生成许多检测不同队伍单位宾语。

### teamDetect必选参数

prefix：标记宾语引用前缀。（[info宾语默认参数](#info宾语中基本参数介绍)）

reset：必选，表示检测宾语的刷新周期。

setTeam：数字二维数组，表示队伍分组。同阵营内部用空格隔开，阵营之间用逗号隔开。填写例子"0 2 4 6 8 10 12 14 16 18,1 3 5 7 9 11 13 15 17"。

setidTeam：字符串数组，表示队伍id前缀。不同阵营之间用逗号隔开，一个阵营仅有一个id。填写例子"A_city,B_city"。

### teamDetect可选参数

isprefixseg：可选，默认为否。（[info宾语默认参数](#info宾语中基本参数介绍)）
```

```
18
19 args：可选，默认没有，必填参数。（[info宾语默认参数](#info宾语中基本参数介绍)）
20
21 opargs：可选，默认没有，可选参数。（[info宾语默认参数](#info宾语中基本参数介绍)）
22
23 cite_name：可选，默认没有，标志宾语标记。（[info宾语默认参数](#info宾语中基本参数介绍)）
24
25 brace：可选，默认没有，外部引用翻译列表。（[info宾语默认参数](#info宾语中基本参数介绍)）
26
27 aunit：可选，检测单位类型，用于unitType。
28
29 minUnits：可选，默认为1，建筑检测的minUnits。
30
31 maxUnits：可选，默认没有该参数，建筑检测的maxUnits。
32
33 name：可选，字符串数组，默认没有，表示不同阵营宾语的名称。每个阵营仅会显示一个，不同阵营的名称之间用逗号隔开。填写例子"setidTeam0_0,setidTeam1_0"。
34
35 offset：可选，数字二维数组，默认"0 0"，表示不同阵营宾语偏移。当只有一个坐标时，该偏移对所有阵营均适用。不同阵营之间用逗号隔开，一个阵营有x和y两个坐标，中间空格隔开。填写例子"0 -20,-20 0"。
36
37 offsetsize：可选，数字二维数组，默认"0 0"，表示不同阵营宾语大小改变。当只有一个坐标时，该偏移对所有阵营均适用。不同阵营之间用逗号隔开，一个阵营有x和y两个坐标，中间空格隔开。填写例子"0 40,40 0"。
38
39 此外，还有大量only，将会原样添加到检测宾语中（如果有的话）。它们是：
40 onlyIdle,onlyBuildings,onlyMainBuildings,onlyEmptyQueue,onlyBuilders,onlyOnResourcePool,onlyAttack,onlyAttackAir,onlyTechLevel,includeIncomplete,onlyWithTag
41
42 ### teamDetect_info可被引用的其他参数
43
44 setidTeam0_0,setidTeam1_0...是生成的id。
45
46 setidTeam_id 是生成id的列表。
47
48 setidTeam_id_dep 是生成对应id补集的列表。
49
50 teamtoi是队伍到id索引的字典。
51
52 teamtoid是队伍到id的字典。
53
54 teamtoid_dep是队伍到对应id的补集的字典。
```

我们可以看到，teamDetect_info 有 4 个参数。**prefix** 和 **reset** 就不做介绍了。我们重点介绍 **setTeam** 和

setidTeam。**setidTeam** 是不同阵营的队伍检测的 id 的前缀。如果做的是 2 队团战，那么只需要写两个前缀。如果做混团或混战，那么需要写相当于阵营数的前缀。**setTeam** 则是不同阵营各自拥有的玩家队伍。通过书写两个参数，便可确定本局游戏的玩家阵营情况。

对于经典的 19p 两队团战来说，teamDetect 将会如此书写

```
info : "teamDetect_info_td"
    prefix : "td" // 前缀
    reset : "3s" // 检测刷新时间
    setidTeam : "Ac,Bc" // 阵营检测id
    setTeam : "0 2 4 6 8 10 12 14 16 18,1 3 5 7 9 11 13 15 17" // 阵营
        内的队伍

标记 : "td"
```

对于一张 20p 的混战地图来说，teamDetect 将会如此书写

```
info : "teamDetect_info_td"
    prefix : "td" // 前缀
    reset : "3s" // 检测刷新时间
    setidTeam : "t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,t14,t15,
        t16,t17,t18,t19" // 阵营检测id
    setTeam : "0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19" // 阵
        营内的队伍

标记 : "td"
```

对于一张 8p 4 队的混团战地图来说，teamDetect 将会如此书写

```
info : "teamDetect_info_td"
    prefix : "td" // 前缀
    reset : "3s" // 检测刷新时间
    setidTeam : "Ac,Bc,Cc,Dc" // 阵营检测id
    setTeam : "0 4,1 5,2 6,3 7" // 阵营内的队伍

标记 : "td"
```

在这里，我们可以用标记宾语制作很多队伍检测了，能大大提高效率。然而，在地图制作界面，我们看不到每一个标记宾语的 id 的具体序号是怎样的，必须去翻看标记宾语产生的宾语，才能知道 id 的具体序号，这是很繁琐的。

为了解决这个问题，在 1.5 版本，我们使用了 name 参数，来看到具体的 id 数字。

```
info : "teamDetect_info_td"
    prefix : "td" // 前缀
    reset : "3s" // 检测刷新时间
    setidTeam : "Ac,Bc" // 阵营检测id
    setTeam : "0 2 4 6 8 10 12 14 16 18,1 3 5 7 9 11 13 15 17" // 阵营
        内的队伍
    name : "{'setidTeam0_0'[len(setidTeam[0]):]},{'setidTeam1_0'[len(
        setidTeam[1]):]}" // 查看id数字，不必细究怎么回事
```

```
    offset : "-20 0" // 将检测宾语x轴（竖直）向上提一格
    offsetsize : "20 0" // 将检测宾语x轴（竖直）方向扩大一格

标记 : "td"
```

在 **name** 里，我们让队伍检测宾语的名字显示出 id 的数字。在 **offset** 和 **offsetsize** 里，我们让检测宾语稍微扩大一点，以便让作者可以看清楚队伍检测宾语的 id 数字。这样，玩家在完成自动化之后，就可以看 id 数字自行推算对应的 id 并将 id 用于其他地方了。

然而，这有以下三大缺点。第一，需要作者运行自动化之后才能得到具体 id 用于处理。第二，需要作者亲自去找 id。第三，如果需要重置 id(-r 选项)，那么很可能所有 id 都会乱掉。比如前面的一个队伍检测被删除，那么后面的队伍检测的 id 数字会统统-1。

基于以上理由，我们提供了 **cite_name** 这一参数。该参数强烈建议放置在 **args** 中，在标记宾语中具体填写。以下是 **cite_name** 的具体介绍

```
cite_name：标记宾语的引用，如果没有，无法被引用。比如在某info宾语中设置
    args为"cite_name,str"，在使用该info宾语格式的标记宾语中名称为"e.td"
    ，其中e是前缀。在cite_name后面加"."再添加属性名称，即可实现引用。比
    如td.aunit表明将会引用该标记宾语的其中aunit的内容。
*标志宾语引用不得发生重复。*
*不得出现循环引用，文件中靠前的宾语不得引用靠后的宾语。*
*引用过程不能直接向已有参数（类型不为str）引用。*
```

那么，队伍检测可以如此书写

```
info : "teamDetect_info_td"
    prefix : "td" // 前缀
    reset : "3s" // 检测刷新时间
    setidTeam : "Ac,Bc" // 阵营检测id
    setTeam : "0 2 4 6 8 10 12 14 16 18,1 3 5 7 9 11 13 15 17" // 阵营
        内的队伍
    args : "cite_name,str" // 添加cite_name
    isprefixseg : "true" // 允许前缀与参数之间出现"."

标记 : "td.td莫斯科"
```

那么，假设还有那么一个宾语。

```
mapText :
    globalMessage : "A队占领城市"
    activatedBy : "td莫斯科.setidTeam0_0"
```

接下来如果在宾语自动化中添加 -c 选项。那么该普通宾语的”td 莫斯科.setidTeam0_0”将会被翻译为该队伍检测阵营 1 的 id，可能是”Ac1”或者”Ac10”之类的。使用如下命令

```
objectgroupauto "输入地图路径" -o "输出文件路径" -D -c
```

如此，我们不需要自动化后才能找到 id，而是使用引用自动转换即可。然而，我们发现外部引用的”td 莫斯

科.setidTeam0_0”似乎还是更繁琐了。这时 **brace** 参数就可以派上用场了。**brace** 的具体介绍

```
brace：字符串数组。在最后会将数组中的所有键进行字符串翻译，以为外部引用提供良好的代入环境。比如，设置brace为"a"，a为"{setidTeam0_0}"，cite_name为"m1"。这样外部出现m1.a时，相当于得到了队伍检测的第一个id。
```

我们可以这样设计队伍检测

```
info : "teamDetect_info_td"
    prefix : "td" // 前缀
    reset : "3s" // 检测刷新时间
    setidTeam : "Ac,Bc" // 阵营检测id
    setTeam : "0 2 4 6 8 10 12 14 16 18,1 3 5 7 9 11 13 15 17" // 阵营内的队伍
    args : "cite_name,str" // 添加cite_name
    isprefixseg : "true" // 允许前缀与参数之间出现"."
    a : "{setidTeam0_0}" // {}表示将进行翻译
    b : "{setidTeam1_0}" // {}表示将进行翻译
    brace : "a,b" // 哪些参数将会被翻译从而用于引用。

标记 : "td.td莫斯科"
```

那么前文的普通宾语可以设计为

```
mapText :
    globalMessage : "A队占领城市"
    activatedBy : "td莫斯科.a"
```

这样能实现更加方便的引用。

其他队伍检测可能添加的参数包括 aunit,onlyBuildings 等，请自行使用。

除了许多功能性的 info，还有专门的 info 用于引用。它就是 dictionary_info。下面是该 info 的详细参数，您也可以在dictionary_info 参数中的 dictionary 部分看到。

```
## dictionary_info

这是一个可以书写键值对以供引用的info宾语，任何键值对均可写入。info宾语写完后，再写一个标志宾语就可以进行引用了。

### dictionary_info必填参数

prefix：标记宾语引用前缀。（[info宾语默认参数](#info宾语中基本参数介绍)）

cite_name：第一个必填参数，标志宾语标记。（[info宾语默认参数](#info宾语中基本参数介绍)）

brace：可选，默认没有，外部引用翻译列表。（[info宾语默认参数](#info宾语中基本参数介绍)）
```

```
12
13  ### dictionary_info可选参数
14
15  isprefixseg: 可选，默认为是。（[info宾语默认参数](#info宾语中基本参数介
        绍)）
16
17  还有任意键值对。
```

你可以简单部署 dictionary_info 来进行引用。

```
1  info : "dictionary_info_d"
2      prefix : "d" // 前缀
3      isprefixseg : "true" // 允许前缀与参数之间出现"."
4      mh10 : "mechGun*10,heavyTank*10"
5
6  标记 : "d.d"
```

普通宾语引用：

```
1  unitAdd :
2      spawnUnits : "d.mh10" // 刷新10小机甲，10重坦
3      team: "0" // 队伍0
```

现在，我们学会了队伍检测的自动化和引用如何使用。现在你可以更方便的检测你的建筑的归属，然后将队伍检测 id 用于控制其他各种宾语，确定地图的胜负判定方式，并且提高地图的可玩性。学会这些，你已经完全可以利用以上知识来大幅减轻您的工作量了。阅读至此，你已经可以直接去查看地图示例以及地图示例说明去直接使用那些已经做好的宾语了。目前的城夺地图，您已经都可以去做了。然而，我们还有更灵活的宾语自动化处理方式。

任意自动化（tree_info 和 object_info）

看完以上两则教程，读者可能会疑惑，为什么建筑和队伍检测是分割开来的？

通常来说，很多建筑上就要摆放队伍检测来检测队伍归属。那将二者合一不是更好吗？这在 1.5 版本确实是这么做的。但是后来发现，这样做比较僵化。毕竟，除了对建筑进行检测，还可能会对其他单位进行队伍检测。而且，大面积的队伍检测建筑也不是不可能。由此，二者被分割开来，分为两个 info 分别处理。

然而，仍然是有二者一起使用的需求的。想必作者不愿意放置一个城市的时候要同时放两个标记宾语。我们希望让自动化在具体放置上更简便。这样的话，tree_info 就派上用场了。以下是 tree_info 的参数，您也可以在tree_info 参数中的 tree 部分看到。

```
1  ## tree_info
2
3  这是一个可以产生其他标志宾语的info宾语。产生的新标志宾语将会继续产生新
       宾语。会形成类似一棵树的结构。
4
5  ### tree_info必填参数
6
```

```
prefix：标记宾语引用前缀。（[info宾语默认参数](#info宾语中基本参数介绍)）

name：字符串二维数组。用;分割开来新产生标志宾语的name，建议大量使用引用。

### tree_info可选参数

isprefixseg: 可选，默认为是。（[info宾语默认参数](#info宾语中基本参数介绍)）

args：可选，默认没有，必填参数。（[info宾语默认参数](#info宾语中基本参数介绍)）

opargs：可选，默认没有，可选参数。（[info宾语默认参数](#info宾语中基本参数介绍)）

cite_name：可选，默认没有，标志宾语标记。（[info宾语默认参数](#info宾语中基本参数介绍)）

brace：可选，默认没有，外部引用翻译列表。（[info宾语默认参数](#info宾语中基本参数介绍)）

idprefix：可选，字符串二维数组，默认没有。每一列第一个表示要获取的id前缀，第二个表示要获取的id前缀的数量。这些id应当被用于cite_name。

exist：可选，字符串数组，默认全部为True。当只有一项时，该计算结果对所有分支均适用。每一项都将进行计算，结果为True时，对应位置的标志宾语将正常部署。结果为False或者为其他结果时，对应位置宾语将不能正常部署。

offset：可选，数字二维数组，默认"0 0"，表示不同分支标记宾语偏移。当只有一个坐标时，该偏移对所有分支均适用。不同分支之间用逗号隔开，一个分支有x和y两个坐标，中间空格隔开。填写例子"0 -20,-20 0"。

offsetsize：可选，数字二维数组，默认"0 0"，表示不同分支标记宾语大小改变。当只有一个坐标时，该偏移对所有分支均适用。不同分支之间用逗号隔开，一个阵营有x和y两个坐标，中间空格隔开。填写例子"0 40,40 0"。

### tree_info可被引用的其他参数

idprefix{i}_{j}是检测的id。i是第几个id前缀产生的id，j是该id前缀获得的第几个id。
```

我们现在想将建筑生成和队伍检测结合在一起，应当如何做呢？首先我们列出我们的建筑生成 info 宾语和队伍检测 info 宾语。

```
info : "building_info"
    prefix : "ci" // 前缀
    idprefix : "ci" // id前缀
    args : "inaddteam,str;mtext,str" // 必填参数列表，第一个为建筑的初始队伍，第二个为建筑名字
```

```
    opargs : "t,team,str,-1" // 建筑刷新队伍
    detectReset : "20s" //检测reset
    aunit : "supplyDepot" //单位类型
    addReset : "20s" //添加reset
    isinadd : "true" // 启用建筑初始刷新
    ismtext : "true" // 启用建筑文本

标记 : "ci0.莫斯科" // 莫斯科初始被1号玩家占领

info : "teamDetect_info_td"
    prefix : "td" // 前缀
    reset : "3s" // 检测刷新时间
    setidTeam : "Ac,Bc" // 阵营检测id
    setTeam : "0 2 4 6 8 10 12 14 16 18,1 3 5 7 9 11 13 15 17" // 阵营内的队伍
    args : "cite_name,str" // 添加cite_name
    isprefixseg : "true" // 允许前缀与参数之间出现"."
    a : "{setidTeam0_0}" // {}表示将进行翻译
    b : "{setidTeam1_0}" // {}表示将进行翻译
    brace : "a,b" // 哪些参数将会被翻译从而用于引用。

标记 : "td.td莫斯科"
```

我们发现，只要同时产生两个标记宾语，一个名字叫”ci0.莫斯科”，另一个名字叫”td.td莫斯科”，并且放在一个位置，并且继续产生下去，自然就能实现建筑生成和队伍检测两个功能的结合。tree_info 正是这个思路。下面，我们将详细介绍 tree_info 怎么使用。

tree_info 的 **prefix** 无需赘述。我们需要考虑 **name** 是什么。理论上来说，我们可以这样写 tree_info。这将产生两个标记宾语以供后续继续生成。它们被称为分支宾语。

```
info : "tree_info_ctd"
    prefix : "ctd" // 前缀
    name : "ci0.莫斯科;td.td莫斯科"

标记 : "ctd"
```

这样就能产生一个同时有建筑生成和队伍检测的莫斯科了。然而，由于队伍检测的引用不得相同，那么标记宾语只能生成这一个城市。每产生一个新建筑 + 队伍检测，就需要重新写一次 tree_info。这无疑太麻烦了。这时我们把目光望向了 **args** 和 **opargs**。

等等...**args** 和 **opargs** 是直接将标志宾语中的参数代入的。tree_info 又能代入进什么参数呢？在进阶教程里面，{} 代表着翻译。那么 tree_info 其实可以自己创建参数代入，然后再通过 {} 在 **name** 中翻译，从而实现变相的参数代入。嗯，有办法了

与id前缀有关的选项、附属宾语导入prefix均不能有{}，不会进行翻译。

```
info : "tree_info_ctd"
    prefix : "ctd" // 前缀
```

```
    name : "ci{inaddteam}.{cityname},t{team};td.{td_cite}" // 需要代入的分支宾语参数
    args : "inaddteam,str;td_cite,str;cityname,str" // 必填参数列表，分别是建筑初始队伍，队伍检测引用和城市名字
    opargs : "t,team,str,-1" // 选填参数，代入建筑的,t选填参数中

标记 : "ctd0.td莫斯科.莫斯科" 或 "ctd0.td莫斯科.莫斯科,t0"
```

tree_info 标记宾语可以自由代入这些参数了，并产生分支宾语以同时满足建筑生成和队伍检测功能。想要引用队伍检测时，引用”td 莫斯科”即可。

我们像是搭积木一样，把这样一个建筑 + 队伍检测的宾语自动化搭出来了。然而这些积木块的功能还是比较完整的。我们能不能从头搭一个自动化模式呢？这就需要 object_info 了。object_info 的参数如下，您也可以在object_info参数中的 object 部分看到。

```
## object_info

这是一个任意生成一个宾语的info宾语。可进行时间修正。

### object_info必填参数

prefix：标记宾语引用前缀。（[info宾语默认参数](#info宾语中基本参数介绍)）

objectType：必选，object的类型。

### object_info可选参数

isprefixseg: 可选，默认为否。（[info宾语默认参数](#info宾语中基本参数介绍)）

args：可选，默认没有，必填参数。（[info宾语默认参数](#info宾语中基本参数介绍)）

opargs：可选，默认没有，可选参数。（[info宾语默认参数](#info宾语中基本参数介绍)）

cite_name：可选，默认没有，标志宾语标记。（[info宾语默认参数](#info宾语中基本参数介绍)）

brace：可选，默认没有，外部引用翻译列表。（[info宾语默认参数](#info宾语中基本参数介绍)）

name：可选，默认为""，宾语名字。

offset：可选，数字数组，默认为"0 0"，宾语偏移。

offsetsize：可选，数字数组，默认为"0 0"，宾语大小改变。

time_prefix：可选，默认不存在。将会导入对应time_info的数据，进行时间修
```

```
        正。
30
31  所有可能的宾语参数都能添加，标志宾语将会按程序生成一个对应的宾语。
```

里面没有什么特别值得注意的参数。那让我们试一试用 object_info 和 tree_info 从头搭一个 id 检测器？当这个 id 激活时，原地产生一个敌对 T2 高射炮；当这个 id 没有激活时，原地没有敌对 T2 高射炮。这个自动化模式是不是很有用呢？

首先是高射炮添加了（借鉴了上面的创造参数和 {} 的用法）

```
1  info : "object_info_uad"
2      prefix : "uad" // 前缀
3      args : "itsid,str;otherid,str" // 必填参数，当地没有高射炮时激活id
           ，要检测的id
4      isprefixseg : "true" // 允许前缀与参数之间出现"."
5
6      objectType : "unitAdd" // 生成宾语类型
7      // 以下是参数（一些参数里有引用）
8      activatedBy : "{itsid},{otherid}"
9      allToActivate : "true"
10     resetActivationAfter : "1s"
11     spawnUnits : "antiAirTurretFlak"
12     team : "-2"
13     warmup : "20s"
```

然后是高射炮检测

```
1  info : "object_info_udd"
2      prefix : "udd" // 前缀
3      args : "itsid,str" // 必填参数，当地没有高射炮时激活id
4      isprefixseg : "true" // 允许前缀与参数之间出现"."
5
6      objectType : "unitDetect" // 生成宾语类型
7      // 以下是参数（一些参数里有引用）
8      id : "{itsid}"
9      maxUnits : "0"
10     resetActivationAfter : "1s"
```

如此可以实现，高射炮在 **otherid** 激活时才能产生，并且已经产生后会被高射炮检测抑制。

然后是高射炮移除。

```
1  info : "object_info_urd"
2      prefix : "urd" // 前缀
3      args : "otherid,str" // 必填参数，要检测的id
4      isprefixseg : "true" // 允许前缀与参数之间出现"."
5
6      objectType : "unitRemove" // 生成宾语类型
7      // 以下是参数（一些参数里有引用）
8      deactivatedBy "{otherid}"
9      resetActivationAfter : "1s"
```

这样，当 **otherid** 非激活时，高射炮会被移除。

为了看到这个位置上的 **otherid** 是什么，需要添加

```
info : "object_info_mi"
    prefix : "mi" // 前缀
    args : "otherid,str" // 必填参数，要检测的id
    isprefixseg : "true" // 允许前缀与参数之间出现"."

    objectType : "mapText" // 生成宾语类型
    // 以下是参数（一些参数里有引用）
    text : "{otherid}"
    textSize : "10"
```

这四个 object_info 共同组成了一个完整的检测 **otherid** 是否激活的系统。然后我们用 tree_info 将其黏在一起。

```
info : "tree_info_pt"
    prefix : "pt" // 前缀
    args : "itsid,str;otherid,str" // 必填参数，当地没有高射炮时激活id
        ，要检测的id
    isprefixseg : "true" // 允许前缀与参数之间出现"."
    name : "udd.{itsid};uad.{itsid}.{otherid};urd.{otherid};mi.{otherid
        }" // 四个新标记宾语的产生

标记 : "pt.its1.Ac1" // 探测Ac1是否激活
```

这时我们发现，**otherid** 是我们必须要输入的，而 **itsid** 实际上只是 id，不与其他 id 重复就行，不用我们自己输入。在这里 tree_info 提供了 **idprefix** 来申请一个或多个前缀的一个或多个引用。如此，我们可以修改 tree_info 为

```
info : "tree_info_pt"
    prefix : "pt" // 前缀
    args : "otherid,str" // 必填参数，当地没有高射炮时激活id，要检测的
        id
    isprefixseg : "true" // 允许前缀与参数之间出现"."
    idprefix : "pi,1" // id前缀申请
    name : "udd.{idprefix0_0};uad.{idprefix0_0}.{otherid};urd.{otherid
        };mi.{otherid}" // 四个新标记宾语的产生

标记 : "pt.Ac1" // 探测Ac1是否激活
```

这样就不必自己输入 **itsid** 了。非常有趣的一点是，tree 标记宾语产生的标记宾语也可以是 tree 标记宾语。这也是 tree 标记宾语为什么被称为”tree”的原因。上面的设置可以检测一个 id 是否被激活。这对于胜负检测是至关重要的。因此，我们可以结合城市做一个进一步的粘合。

```
info : "tree_info_pn"
    prefix : "pn" // 前缀
    args : "ctd_cite,str" // 必填参数，当地没有高射炮时激活id，要检测的
        id
```

```
    isprefixseg : "true" // 允许前缀与参数之间出现"."
    ida : "{ctd_cite}.a" // 方便代入，因为引用的"."和参数的"."会混
    idb : "{ctd_cite}.b" // 方便代入，因为引用的"."和参数的"."会混
    name : "pt.{ida};pt.{idb}" // 两个分支宾语的产生，分别是队伍检测的a
        ,b两队
    offset : "0 0,0 80" // 第二个竖直向下移动4格

标记 : "pn.ctd莫斯科" // 探测Ac1是否激活
```

它将产生两个 id 探测器，上下分别探测城市的 A 队占领 id 和 B 队占领 id。当我们把这一部件横着排成一排时，将会自动产生两排检测。我们对这些城市被 A/B 队占领的数量进行检测，并用检测结果控制删除宾语，删除玩家的建造单位来控制胜负。这是非常有效的。如果我们需要检测的是城市的队伍检测 id，那么也许上面的 object_info_mi, tree_info_pt, tree_info_pn 可以有所变化，让它显示城市的名字而非检测 id。留作思考。

运用同样的方法，我们可以对 tree_info 进行一些改动，使得引用的是 tree_info 的标记宾语，而不是 teamDetect_info 的标记宾语。虽然可能使用起来是一样的。

```
info : "tree_info_ctd"
    prefix : "ctd" // 前缀
    name : "ci{inaddteam}.{cityname},t{team};td.{idprefix0_0}" // 需要
        代入的分支宾语参数
    args : "inaddteam,str;cite_name,str;cityname,str" // 必填参数列表，
        分别是建筑初始队伍，队伍检测引用和城市名字
    opargs : "t,team,str,-1" // 选填参数，代入建筑的,t选填参数中
    idprefix : "mtd,1" // id申请

标记 : "ctd0.ctd莫斯科.莫斯科" 或 "ctd0.ctd莫斯科.莫斯科,t0"
```

并且从 teamDetect_info 中提取重要参数（不然无法通过.a.b 方式获取 id）。可以进一步修改

```
info : "tree_info_ctd"
    prefix : "ctd" // 前缀
    name : "ci{inaddteam}.{cityname},t{team};td.{idprefix0_0}" // 需要
        代入的分支宾语参数
    args : "inaddteam,str;cite_name,str;cityname,str" // 必填参数列表，
        分别是建筑初始队伍，队伍检测引用和城市名字
    opargs : "t,team,str,-1" // 选填参数，代入建筑的,t选填参数中
    idprefix : "mtd,1" // id申请
    a : "{idprefix0_0}.a" //A队id提取
    b : "{idprefix0_0}.b" //B队id提取
    brace : "a,b" // 哪些参数将会被翻译从而用于引用。

标记 : "ctd0.ctd莫斯科.莫斯科" 或 "ctd0.ctd莫斯科.莫斯科,t0"
```

任意自动化到此告一段落。相信你已经基本掌握了如何使用 object_info 和 tree_info。现在，你能否使用 object_info 和 tree_info 来搓出一个建筑呢？留作作业吧（笑）。理论上来说，object_info 和 tree_info 可以拼出任意形式的自动化。作者可以试着做出更多有意义的部件。

多个文本/添加/删除（multi info 系列）

在前言中，我们提到过队伍检测可以操控很多其他宾语。例如判定胜负，控制城市名称，控制动态国名，控制剧情放送，控制刷兵等等。队伍检测不是目的，目的是使用这些检测来对这些宾语进行控制。

总的来说，总共有三种宾语需要被控制。mapText,unitAdd 和 unitRemove。并且，我们希望同时控制多个相同类型的宾语。这就是三个 info 的出现的原因，它们是 multiText_info,multiAdd_info,multiRemove_info。这三个 info 非常相似，仅在类型和一些具体参数上略有不同。

首先是 multiText_info 的应用。一般来说，城市名称都是不随队伍检测改变的。但是有时，我们需要随着队伍检测改变城市名称/颜色/文本大小。不同阵营攻占同一城市时，名称可能需要不同，比如古比雪夫和萨马拉，斯大林格勒和保卢斯堡等。颜色也会发生变化。颜色和城市名称的变化在混战图中也许更加有用，这样混战玩家能更好地观察其他势力的状况。

就如前文所述，出现了一个莫斯科城。

```
标记 : "ctd0.ctd莫斯科.莫斯科"
```

现在，我们需要根据莫斯科城的归属决定城市的名称和颜色，例如。

```
mapText:
activatedBy : "ctd莫斯科.a" //A队占领激活
resetActivationAfter : "3s" //每隔三秒检测一次
text : "莫斯科" //城市名称
textSize : "11" //文本大小
textColor : "#FF4500" //RGB格式颜色

mapText:
activatedBy : "ctd莫斯科.b" //B队占领激活
resetActivationAfter : "3s" //每隔三秒检测一次
text : "莫斯科维恩" //城市名称
textSize : "11" //文本大小
textColor : "#191970" //RGB格式颜色
```

我们发现，这个莫斯科城本身也有城名，可能也需要进行一些修改。让我们先来看 multiText 是如何实现的吧。multiText_info 的参数如下。您也可以在multiText_info 参数中的 multiText 部分看到。

```
## multiText_info

这是一个产生多个文本的info宾语。会根据规则自动产生一系列文本宾语。

### multiText必选参数

prefix：标记宾语引用前缀。（[info宾语默认参数](#info宾语中基本参数介绍)）

text：必选，字符串数组。文本宾语文本内容。如果仅有一个，那么该text将会应用于所有产生文本宾语中。text将尽力全部加入文本宾语组中。

```

```
### multiText可选参数

isprefixseg: 可选，默认为否。（[info宾语默认参数](#info宾语中基本参数介绍)）

args：可选，默认没有，必填参数。（[info宾语默认参数](#info宾语中基本参数介绍)）

opargs：可选，默认没有，可选参数。（[info宾语默认参数](#info宾语中基本参数介绍)）

brace：可选，默认没有，外部引用翻译列表。（[info宾语默认参数](#info宾语中基本参数介绍)）

reset：可选，默认为1s，表示文本宾语的刷新周期。

acti：可选，字符串二维数组，默认没有。文本宾语的激活来源被";"分隔，将会添加进入。文本宾语数量保证将acti均加入。

deacti：可选，字符串二维数组，默认没有。文本宾语的抑制来源被";"分隔，将会添加进入。文本宾语数量保证将deacti均加入。

teamDetect_cite：可选，默认没有，teamDetect标记宾语引用。将会将teamDetect的id逐个加入激活中。如果isdefaultText选择，将继续所有teamDetect的id加入下一个的抑制中。文本宾语数量保证teamDetect_cite充分发挥作用。

isdefaultText：可选，默认为否，teamDetect_cite是否包含默认文本。具体见上文teamDetect_cite。

numDetect_cite：可选，默认没有，numDetect标记宾语引用。将会将numDetect的id逐个加入激活中。文本宾语数量保证numDetect_cite充分发挥作用。

textsize：可选，字符串数组，默认没有。文本宾语文本的大小。如果仅有一个，那么该textsize将会应用于所有产生文本宾语中。textsize将尽力全部加入文本宾语组中。

color：可选，字符串数组，默认没有。文本宾语文本的颜色。如果仅有一个，那么该color将会应用于所有产生文本宾语中。color将尽力全部加入文本宾语组中。

name：可选，字符串数组，默认为""，表示不同阵营宾语的名称。每个阵营仅会显示一个，不同阵营的名称之间用逗号隔开。

offset：可选，数字二维数组，默认"0 0"，表示不同组宾语偏移。当只有一个坐标时，该偏移对所有组均适用。不同组之间用逗号隔开，一个组有x和y两个坐标，中间空格隔开。填写例子"0 -20,-20 0"。

offsetsize：可选，数字二维数组，默认"0 0"，表示不同组宾语大小改变。当只有一个坐标时，该偏移对所有组均适用。不同组之间用逗号隔开，一个组有x和y两个坐标，中间空格隔开。填写例子"0 40,40 0"。
```

info 默认参数不必解释。值得注意的是默认参数中没有 **cite_name**，这意味这 multi 系列的 info 均没有引用，它们都是“效应器”，不需要引用以供其他标记宾语使用。

我们可以发现，并没有一个参数指明了一共会产生几个 mapText 宾语。这一过程是自动的。对于这些参数来说，有两类参数。一类是引用参数，比如 teamDetect_cite,numDetect_cite 等，这些参数可以填入对应的引用，它们会提供 id 激活的相关信息，mapText 会“保证”它们充分发挥作用。对于 teamDetect_cite 来说，就是会至少产生 setidTeam 的 id 前缀数量的 mapText，并且一个个的被队伍检测的 id 激活。特别的，如果 isdefaultText 为真，那么还会产生第 n+1 个 mapText，并且该 mapText 会被所有队伍检测 id 抑制。isdefaultText 的含义是，有可能要求部署一个在所有阵营都不占领城市的情况下产生的城市文本。

而普通的参数则当只有一个时，所有 mapText 都将添加这一个参数。当普通参数不止这一个时，会有两种情况。一种是“保证”充分发挥作用，即如果 mapText 数量不足以塞进那么多参数时，将增加 mapText 数量。另一种是“尽力”加入，即如果 mapText 数量不足以塞进那么多参数时，将会忽视后面的那些参数。

由此，我们可以部署我们的城市文本了。

```
标记 : "ctd0.ctd莫斯科.莫斯科"

info : "multiText_info_mt"
    prefix : "mt" // 前缀
    isprefixseg : "true" // 允许前缀与参数之间出现"."
    args : "teamDetect_cite,str;atext,str;btext,str" // 必填参数列表，分别是队伍检测引用，A队占领文本，B队占领文本
    color : "#FF4500,#191970,white" // 颜色，分别是A占，B占，中立时的颜色
    text : "{atext},{btext},{atext}" // 文本内容，分别是A占，B占，中立时的文本
    textsize : "11" // 文本大小
    isdefaultText : "true" // 允许中立文本出现

标记 :  "mt.ctd莫斯科.莫斯科.莫斯科维恩"
```

这个将会产生 3 个城市文本，满足我们的要求。然而，有时候我们可能并不需要那么多必填参数。在大多数时候，城市名称是一样的，但是颜色不一样。同时我们做一些可以泛化的调整，因此我们可以这样

```
标记 : "ctd0.ctd莫斯科.莫斯科"

info : "multiText_info_mt"
    prefix : "mt" // 前缀
    isprefixseg : "true" // 允许前缀与参数之间出现"."
    args : "teamDetect_cite,str;mtext,str" // 必填参数列表，分别是队伍检测文本，默认城市名称
    opargs : "s,textsize,str,7;a,atext,str,{mtext};b,btext,str,{mtext}" // 可填参数列表，分别是文本大小，默认为7，A队占领，默认为“默认名称”，B队占领，默认为“默认名称”
    color : "#FF4500,#191970,white" // 颜色，分别是A占，B占，中立时的颜色
    text : "{atext},{btext},{mtext}" // 文本内容，分别是A占，B占，中立时的文本
```

```
    isdefaultText : "true" // 允许中立文本出现

标记 : "mt.ctd莫斯科.莫斯科,b莫斯科维恩,s11"
```

这时，我们还是可以把 multiText_info 粘合入 tree_info。形成建筑、队伍检测、城市文本的综合体。并且将队伍检测的 a,b 导入 tree_info 中

```
info : "building_info"
    prefix : "nc" // 前缀
    idprefix : "nc" // id前缀
    args : "inaddteam,str" // 必填参数列表，第一个为建筑的初始队伍
    opargs : "t,team,str,-1" // 建筑刷新队伍
    detectReset : "20s" //检测reset
    aunit : "supplyDepot" //单位类型
    addReset : "20s" //添加reset
    isinadd : "true" // 启用建筑初始刷新

info : "teamDetect_info_td"
    prefix : "td" // 前缀
    reset : "3s" // 检测刷新时间
    setidTeam : "Ac,Bc" // 阵营检测id
    setTeam : "0 2 4 6 8 10 12 14 16 18,1 3 5 7 9 11 13 15 17" // 阵营内的队伍
    args : "cite_name,str" // 添加cite_name
    isprefixseg : "true" // 允许前缀与参数之间出现"."
    a : "{setidTeam0_0}" // {}表示将进行翻译
    b : "{setidTeam1_0}" // {}表示将进行翻译
    brace : "a,b" // 哪些参数将会被翻译从而用于引用。

info : "multiText_info_mt"
    prefix : "mt" // 前缀
    isprefixseg : "true" // 允许前缀与参数之间出现"."
    args : "teamDetect_cite,str;mtext,str" // 必填参数列表，分别是队伍检测文本，默认城市名称
    opargs : "s,textsize,str,7;a,atext,str,{mtext};b,btext,str,{mtext}" // 可填参数列表，分别是文本大小，默认为7，A队占领，默认为“默认名称”，B队占领，默认为“默认名称”
    color : "#FF4500,#191970,white" // 颜色，分别是A占，B占，中立时的颜色
    text : "{atext},{btext},{mtext}" // 文本内容，分别是A占，B占，中立时的文本
    isdefaultText : "true" // 允许中立文本出现

info : "tree_info_cco"
    prefix : "cco" // 前缀
    name : "nc{inaddteam},t{team};td.{idprefix0_0};mt.{idprefix0_0}.{cityname},s{textsize},a{atext},b{btext}" // 需要代入的分支宾语参数
    args : "inaddteam,str;cite_name,str;cityname,str" // 必填参数列表，分别是建筑初始队伍，队伍检测引用和城市名字
    opargs : "t,team,str,-1;s,textsize,str,7;x,atext,str,{cityname};r,
```

```
            btext,str,{cityname}" // 选填参数，,t代入建筑的,t选填参数中，,s
            代入建筑文本的,s参数，,x代入建筑文本的,a参数，,r代入建筑文本的,b
            参数。（规避参数中的a,b）
36      idprefix : "mtd,1" // id申请
37      a : "{idprefix0_0}.a" //A队id提取
38      b : "{idprefix0_0}.b" //B队id提取
39      brace : "a,b" // 哪些参数将会被翻译从而用于引用。
```

这样，我们可以使用 cco 来部署不同阵营占领时颜色不同的城市了。

振荡器（flash_info 和 building_f_info）