

# GMIS 使用指南

作者：张泓冰<hongbing75@gmail.com>

Copyright @2014 保留所有权利。

## 简介

通用机器智能系统（GMIS）旨在构建一个通用的机器人头脑，让所有具有现代操作系统的机器，包括个人电脑，手机，智能电视，智能汽车，以及其他工业机器等，都变成普通用户可以用母语驱动的机器人。

不同于当前各种智能语音助理，GMIS 提供一门类自然语言，通过与用户母语兼容的方式，实现编程控制机器人的实时行为，这给机器人的应用带来非常大的实用性和灵活性。

我们已经能用类自然语言控制一个 GMIS 原型自动玩一款在线德克萨斯牌游戏。它将帮助我们验证和展示如下能力：

- 能接受类自然语言的编程驱动

目前为止，所有的编程语言都是为程序员设计的，它严重限制了普通用户与机器的交互能力，我们急需能用自然语言编程驱动机器人的能力，特别是那些只会用母语，急需机器人辅助照顾的老年人。当前，我们提供一个称为 Final C 的语言代替用户的母语和 GMIS 原型交互，它实际上是自然语言的极度简化版，没有用户需要特别学习的语法，它可以帮助 GMIS 在近期为带来实用价值，未来我们将逐步提高 GMIS 对自然语言的理解程度。

- 通过本能实现物种多样性

GMIS 为所有机器人提供一个统一的，可配置的智能架构，不同的机器人可以根据不同应用目的拥有特定的本能，实现机器人物种的多样化。

- 能像人一样学习和记忆

就像初生的婴儿一样，机器人的语言和能力都是后天学习得到的，即使拥有相同本能的机器人，不同成长经历的也会具有不同的能力。这意味着用户必须自己教授自己的机器人，当然，这会给用户体验带来一些障碍，我们将让这一切变得友好。

- 能像神经树那样执行逻辑

任何任务逻辑在 GMIS 内部会编译成一颗树来执行。任何树重组后得到的依然是一棵树，这是 GMIS 未来自主编程自主智能的关键，也是世界首次实现此类设计。

- 具备逻辑重入能力

人可以让一件事做到一半后暂时终止，不占用大脑资源，然后在数十年后接着完成而不是重新开始，GMIS 也具备这样的能力，任何一个逻辑任务都可以暂停执行，释放内存，

即使关闭电脑，重新打开后依然可以接着执行，这意味着机器人也可拥有像人类那样的睡眠修复机制。

- 可以实现动态逻辑  
就像人做事可以随时随地地根据条件变化而改变预定行为，GMIS 也可以动态边建设边执行逻辑，或运行时改变一个正在执行逻辑。
- 可以使用外部物体  
通过使用外部物体的形式，GMIS 可以完美承继现有的各种应用代码成果，这点也有重要的现实意义。
- 自然的并行执行能力  
并行计算向来是一个难点和热点，特别是并行和串行执行交织的情况下，目前并没有一个通用编程模式，可这对人类大脑来说这是基本能力，我们常常说：“先同时做 A 和 B，然后做 C”，即可实现并行和串行协同计算，现在 GMIS 也可以如此轻松做到。
- 可与其他机器人分工合作  
就像人类之间的分工合作一样，任何基于 GMIS 的机器人都可以互联互通，动态组成物联网，分工合作执行同一个逻辑任务，释放巨大的生产力。

实际上，你可以把 GMIS 看作是下一代操作系统，它最终将覆盖现有操作系统，使其变为底层。

本指南的目的是帮助用户了解以上提到的各种技术特性和突破，并提供了一些实际应用案例来证明 GMIS 原型已经具备一定程度的实用价值。

用户可以按本指南在个人电脑(目前仅在 windows 系统上实现)上操作配套的 GMIS 原型测试上述能力。毫无疑问，GMIS 原型未经严格测试，存在各种可能的编程 BUG。阅读本指南最好具有一定的 C++编程经验。

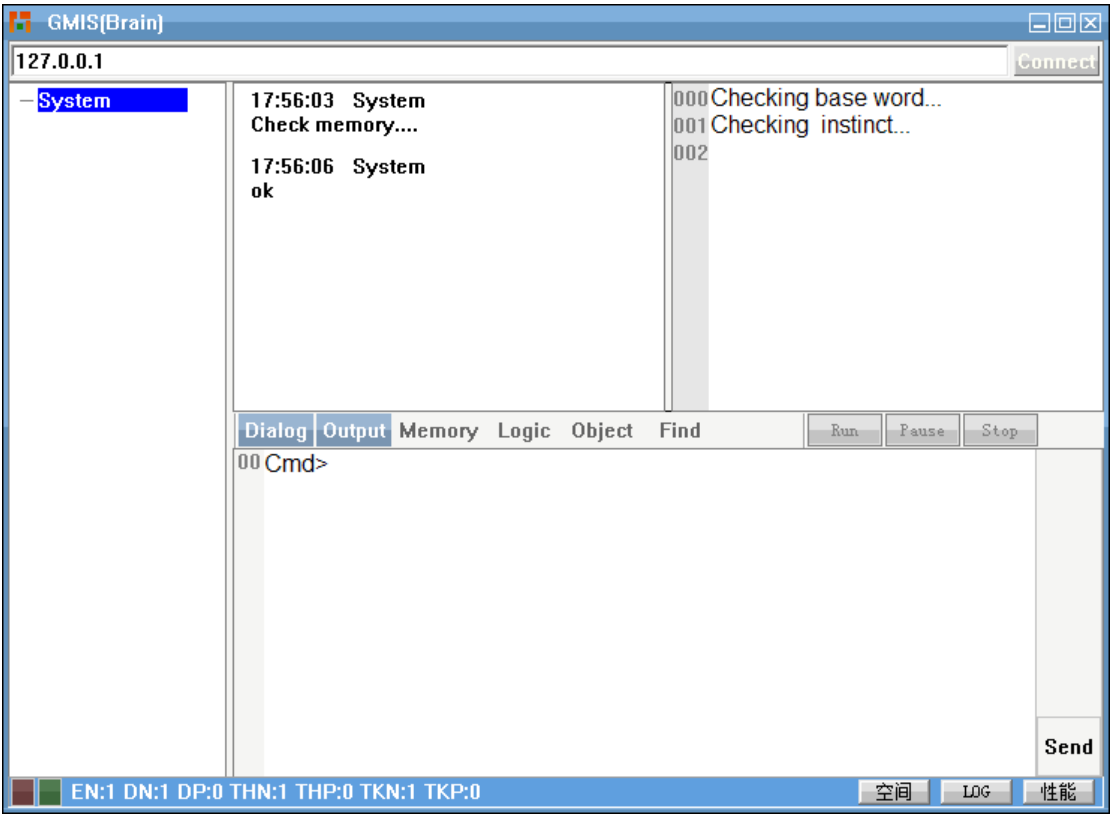
## 用户界面

先让我们来认识一下 GMIS，一个完整的 GMIS 实现主要分为三部分，或者说三个应用程序，分别代表：大脑，空间入口和执行器。

当前没有器官给大脑提供信息和反馈，所以我们提供一个数字空间，使得大脑可以直接使用数字空间里的物体。执行器则负责模拟数字空间里的物理反应，实际上就是执行数字空间里的物体。

# 大脑

理论上，大脑应该由信息器官负责提供输入输出，这里我们用一个用户界面来代替，这个界面可以远程连接大脑，当然也可以在本地和大脑集成在一起，用户在安装目录下启动：GMIS.exe 既可以看到如下界面：



如图，一个用户界面分为三部分：左边是一个对话列表，右下是输入框，右上则是一个信息输出窗口。

## 对话列表

界面的左边是一个 **tree view**，用来显示当前大脑内存在的对话，每一个对话代表着大脑内部的一个思绪或任务。

系统会提供一个缺省的主对话，主对话类似神经中枢，外界的一切信息会先送给它，如果有需要，它会生成一个子对话，再把信息交给子对话执行。

子对话也可以有自己的子对话，每一个子对话都是为父对话服务的，它的任务执行结果会反馈给自己的父对话。这种安排有点像你在做一件事，期间有一个问题需要问同事，这样就产生了一个子对话，等同事给你回话以后，你则继续做之前的事。换句话说，GMIS 具有交互式逻辑对话的能力。

## 输入窗口

界面的右下是一个输入窗口，但用户在对话列表点击一个对话后，此对话就成为当前焦点对话，用户可以输入窗口输入信息，输入完后通过点击“Send”按钮发送给此焦点对话执行。

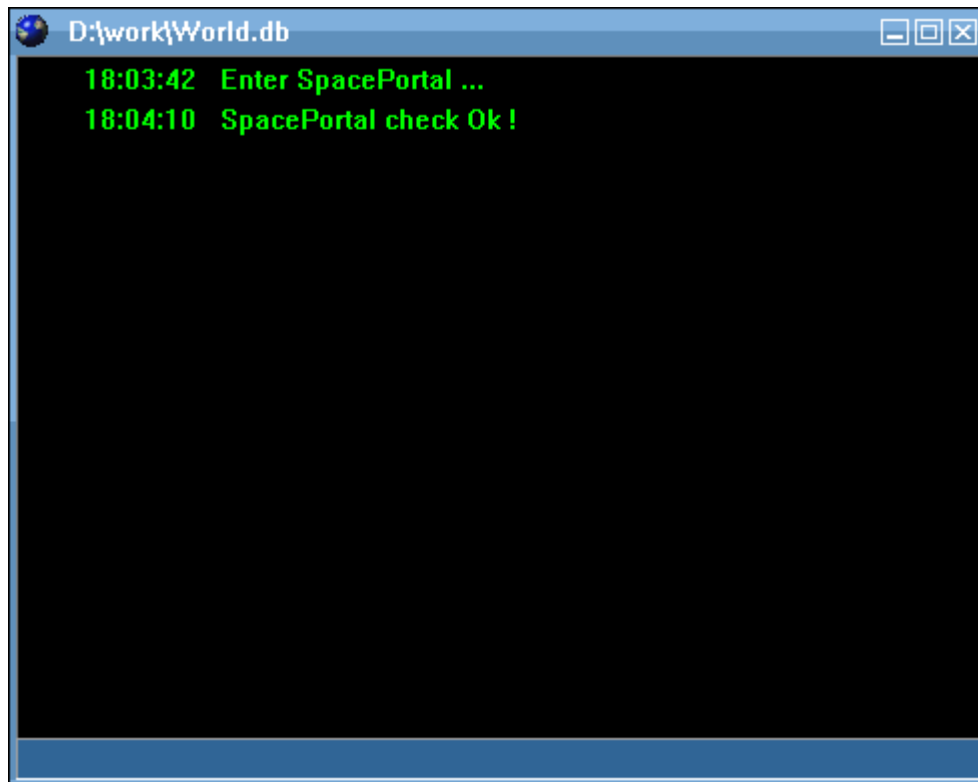
## 输出窗口

输出窗口包含多个子窗口，负责显示不同的信息，用户可以通过中间 toolbar 上的按钮选择打开或者关闭。主要有：

- 1) **Dialog view**: 负责显示用户输入的历史对话以及的反馈结果。
- 2) **Output view**: 负责显示一个逻辑在执行时输出的运行时信息
- 3) **Memory view**: 显示临时记忆，主要是当前对话执行逻辑后生成的一些临时数据，这些数据仅对当前对话有效
- 4) **Object view**: 事实上也是一个临时记忆窗口，专门负责显示当前使用的外部物体，关于什么是外部物体，随后的章节会有进一步的解释。
- 5) **Logic view**: 同 **Object view** 类似，专门显示用户当前生成的临时逻辑。
- 6) **Find view**: 用户可以搜索大脑里的长期记忆，然后在这里显示搜索结果。
- 7) **Debug view**, 此 **view** 只有用户暂停一个逻辑的执行后才可看到，负责显示当前逻辑的调试信息。

## 空间入口

空间入口基本上是一个服务程序，它没用用户需要操作的界面，但是为了显示一些 log 信息，我们还是给它配上了一个窗口。



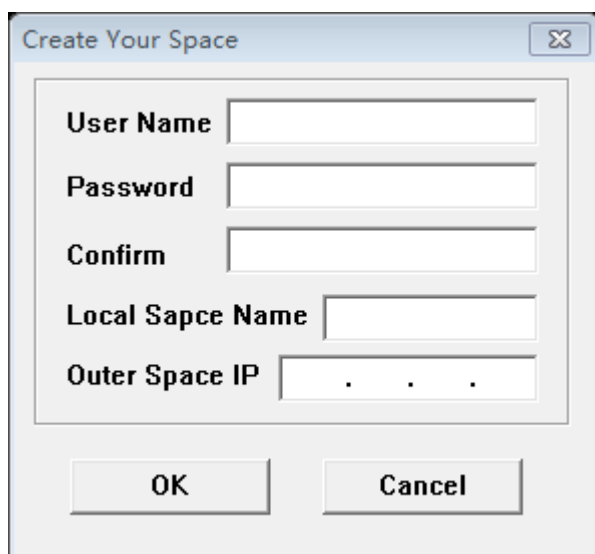
## 执行器

执行器主要负责执行具体外部物体，由空间入口程序在有需要的时候启动，执行空间交付的任务，并把执行结果反馈给空间并自动关闭。

执行器不需要自己的界面，但为了调试方便，我们也会给它配上一个 log 窗口，不过默认不显示，但你可以任务管理器里看到它的存在。

## 启动 GMSI 实例

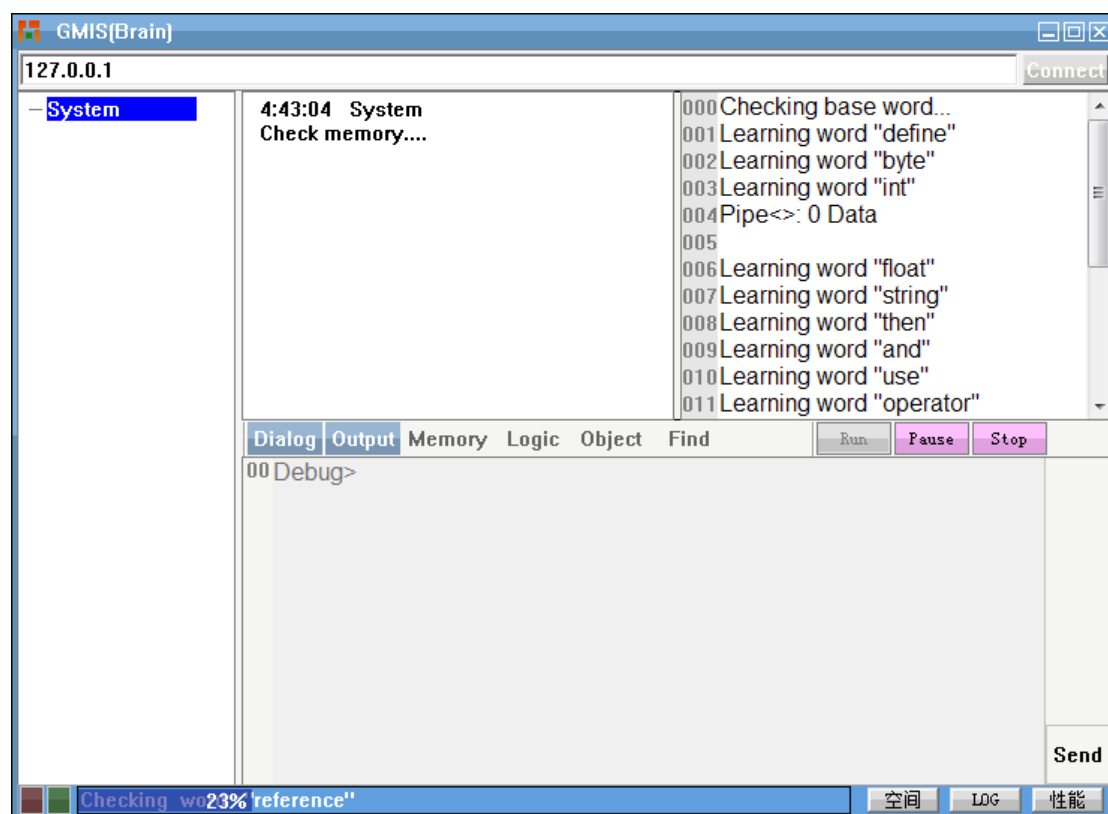
正常情况下，你可以在本指南同一个目录下发现 **GMIS.EXE** 程序，打开它，首次打开会要求你注册用户名：



A dialog box titled "Create Your Space" with a close button (X) in the top right corner. It contains five input fields: "User Name", "Password", "Confirm", "Local Sapce Name", and "Outer Space IP". The "Outer Space IP" field has pre-filled dots indicating a dotted decimal format. At the bottom are "OK" and "Cancel" buttons.

其中,Local Space Name 你可以填写"Home",或者任何你喜欢的名字, Outer Space IP 可以不填。

然后你会发现 GMIS 开始检查大脑记忆, 我们之前说过, 除了本能, GMIS 的所有能力都是后天学习的, 由于这是一个新生儿, 所以会预先学习一些单词和命令, 耐心等待它完成。



下次启动时, 只是简单的检查, 会快很多。完成之后, 你就可以输入命令了。但是 GMIS 如果不连接到数字空间, 那么将无法使用外部物体。

连接空间，可以点击左上角的“Connect”按钮，注意，我们预置的 IP 为 127.0.0.1，这意味着空间入口程序应该预先在本地被启动。如果 GMIS 检测到空间入口程序没有启动，会自动先打开，你也可以手动在与 GMIS.exe 相同的目录下打开 SpaceSport.exe 程序，首次打开同样需要你注册一个用户，作为空间的主人。

注意：你必须填写和你之前注册 GMIS 一样的用户名和密码，因为 GMIS 会用这个用户名去登录空间。

成功之后，你会看见地址栏变成你所登录空间的名字，比如是“Home”。你也可以点击左下角的“空间”按钮，这时会看到一个三维房间：



为什么是三维的呢？这和另一个目标有关，我们希望每一个用户的空间都可以相互连接，变成一个三维的数字世界。

目前而言，只是做了初步的探讨，它是否成功不影响 GMIS 的使用，可以直接忽略。

让我们点击“空间”按钮，回到对话模式，你可能会觉得无从下手吧？这里有一个建议：在输入窗口输入：“define int 23;”，然后点击“Send”按钮，观察变化，正如命令的字面意义，你已经命令 GMIS 做了第一件事。以后所有 GMIS 的所有命令都是类似的短语组合，是不是和自然语言非常类似呢？

# Final C 语言

目前的 GMIS 原型还不能直接理解人类语言，我们提供一个称为 Final C 的语言代替用户的母语和这个原型交互，使其内部实现逻辑编程能力。

首先要指出的，用户不用担心 Final C 需要特别的学习，它只是简化了你现在使用的母语。Final C 不再是为程序员设计的另一门新语言，它不再有内存，指针，函数这类概念，也没有需要用户记忆特别的语法。

不过，作为起点，仍无法避免需要一些编程知识背景，这里假设用户具有至少一门高级语言的编程经验。

当我们学习 C 语言时，知道了面向过程编程，当我们学习 C++或者 JAVA 时，知道了面向对象编程，当我们试图了解 Final C 时，这里提出一个面向逻辑编程的概念。

我们定义，所谓逻辑就是行为的有意义组合。所以面相逻辑编程，实际上就是以行为为单位来编程。

行为很难说是一个纯粹的技术概念，它的背后总是在特定语境下对人有某种意义某个动作，这个动作对计算机来说，可能只是一个函数，也可能是一个函数的集合，但不是每一个函数或者一个函数的集合都可以称为行为。

比如当一个程序执行一个打开文件函数，我们可以说它是一个行为，因为我们可以直接跟另一个人说：“打开某个文件”，对方能完全理解，但如果在打开文件这个函数内部又调用内存存取函数，那么这个内存存取函数对我们来说又不是我们所关注的行为(至少在多数语境下)，更不用说这个内存存取函数实际上又会在内部调用更多的系统 API 函数。

函数与行为的差别其实可以归结为一个颗粒度问题，目前的编程语言的颗粒度是函数，而 Final C 编程的颗粒度是对人有交流意义的函数和其组合。

## 定义本能

就好比任何一个 C 程序最终可以分解为 C 库函数的组合，Final C 编程也有自己的基本行为单位，这个单位称之为本能。

我们预定义了几十种本能，至于这些本能在计算机内部是由一个或多个函数实现的，这不是重点，我们只需知道，这些本能是一个机器所能做出的，在我们看来有意义的一些最基本行为，它是 GMIS 实例执行逻辑或维持自身运转所必须的最小行为集合。

比如“`define int 23`”就是我们预定义的本能之一，它生成一个整数：23，如果缺少这个行为，那么就没有整数可以使用。



其他本能还有（节选）：

```
//通用命令
////////////////////////////////////
#define    INSTINCT_DEFINE_STRING          127282417796630008
...
#define    INSTINCT_USE_OBJECT              127282417797000004

//内部命令——仅仅影响大脑内部的行为
////////////////////////////////////
// 1) 内部独立命令——只能单一执行，不能与其他命令组合为逻辑
#define    INSTINCT_BRAIN_INIT             127282417798000000
#define    INSTINCT_THINK_LOGIC            127282417798000001
...
#define    INSTINCT_CLOSE_DIALOG           127282417798000008

// 2 内部非独立命令（21）
#define    INSTINCT_LEARN_TOKEN            127282417798010001
#define    INSTINCT_LEARN_ACTION           127282417798010015
#define    INSTINCT_FIND                   127282417798010019
...
```

基本上，这些的本能除了一些数学上的抽象行为，剩下就是一些针对 **GMIS** 实例自身的内部操作。

现在大家可能有一个疑问，所有的 **Final C** 逻辑都是本能的组合，那么这样的组合能让机器人实现我们所需要的行为表现吗？

对于专用机器人，我们可以定制特别的本能，这样既有助于执行效率，又体现出机器人物种的不同，就像自然界不同的动物都有各自特别的本能一样。

而对于像“人”这样的高等机器人来说，它有一个特别的本能：可以使用外部物体，所谓的外部物体就是你用高级编程语言编写的各种程序模块，这就像人类通过使用工具而实现了自己各种目标一样，**GMIS** 通过此方式可以承继现有 IT 世界的技术成果。

当人类的原始人开始学会使用工具后，衍生出学习逻辑能力，以及相互合作的能力，最终走向智能进化的道路，**GMIS** 机器人也将走向类似的道路。

从技术上来看，当本能数量和类型被固定化后，让机器自动编程变为可能，你不用再考虑无数个函数带来的差别，剩下的只需要考虑如何组合使用它们。

## 行为短语

每一个本能在大脑内部用一个 64 位整数 ID 来表示，显然直接用一个 ID 来调用本能是一件很难受的事，这就好比直接用机器码编程，现代编程语言解决这个问题的办法是使用函数名。

使用函数名代替 ID 提高了程序员的编程效率，但却无助于人与机器人的交互，很多人可能还没意识到，现代计算机语言的源程序只能在程序员可视情况下阅读理解，即使我们读给其它程序员听也没用，而我们现在需要的却是：在听说环境下也能编程驱动机器人。

既然本能是对人有意义的某种行为，那么我们为什么不用像人类自然语言那样易于理解语句去标识它呢？

当前，Final C 规定：所有行为（本能是最简单的行为）都用一个行为短语去标记。

行为短语必须符合以下语法规则：

**动词+[副词]+[形容词]/[名词]+[名词]**

也就是说，所有行为短语，第一个单词必须是动词，动词之后可以有其它单词，也可以没有，如果有，动词后可以接一个副词，其后可以含有名词，名词则允许被其它形容词或名词修饰。

比如，一个定义整数的本能，我们可以用“define int”去表达，这里 define 是动词，int 是名词，GMIS 先分别学习这两个单词，然后学习这两个单词的组合，然后把这个组合的记忆 ID 去标记本能 ID，那么我们通过输入“define int”，就可以回取得到一个本能 ID，然后系统要求这个本能 ID 必须要跟着输入一个数字，于是，用户完整输入一个“define int 2”便能得到一个具体的定义整数 2 的行为。

你可能发现这里没有提到主语，一方面缺省的主语是机器人自己，另一方面，GMIS 对语言理解还在做进一步研究，目前强行规定语法只是一个权宜之计，是为了能兼容之后的改变。

当我们用按上述方式标记行为后，不仅利于自己理解，也利于和其他人交流。为了方便用户使用，GMIS 实例在第一次初始化时给所有本能标记了一个英语版本的命令，但用户随时可以用其他语言后天学习同一个本能。

除了本能，GMIS 同样可以把本能组合得到的逻辑视为行为，让用户用一个行为短语去标记这个逻辑，然后像使用本能一样使用它，这样一个 GMIS 的语言能力和行为能力将完全是用户后天教授的，这和人类成长过程类似。

## 逻辑关系

这里的逻辑关系是指行为的组合方式，不同与其他编程语言，为了方便程序员，把编程时所需的逻辑关系封装成了循环，条件，顺序，这大大减轻了程序员负担，但代价是程序的表达与人类日常使用的语言习惯严重背离，导致一段简单的程序也很难读给其他人听。

而 Final C 只有两种：串联或者并联，换句话说就是你要么先做什么后做什么，要么同时做什么。

以下列本能的组合为例：

1) Define int 2, then define int 3;

这句包含两个子句，每一个表达一个定义整数的本能，后一个本能用了 then 开头，表示这两个行为的逻辑关系是，先执行 define int 2, 然后执行 define int 3, 通常为了方便，可以忽略输入 then，那么系统默认这两个行为是串联关系

2) Define int 2, and define int 3;

同样是这两个子句，但后一个子句是以 and 开头，表示这两个行为是同时执行。

只用串联或并联来表达行为之间的逻辑关系很符合人类语言表达的直觉，在日常生活中，人们用这两个关系简单的组合行为就可以满足绝大多数需求，但这并不是没有代价的，使用这两个关系要表达复杂逻辑关系就不像其他高级编程语言那么简单。

可以这么说，Final C 让简单的更简单，因为人们日常生活中多数语言仅仅是几个行为短语的组合，这样老太太都可以下意识的用母语去编程驱动机器人做 90% 的事，它也让复杂的更复杂，要让机器人学习复杂算法，需要更专业的人去编程，幸运的是，这个复杂逻辑编程完成后可以视为单一行为被机器人学习，从而被老太太简单的使用。

## 面向逻辑编程

完全由行为组合而成的逻辑称为面向逻辑编程。问题是，当一段逻辑只剩下串联与并联关系后，它能表达任意算法吗？本节就是要证明它的可行性。

首先，让我们用一句话来介绍 GMIS 的逻辑执行形式，它和你中学物理里学习的电路很类似：一根数据管道就像电线，流经行为的组合，每一个行为就像电子元件，它们要么并联组合，要么串联组合，每一个元件从中取出自己所需的数据，执行完后又释放数据（如果有）到数据管道中。

根据结构化编程理论，我们只要实现了条件循环，就可以实现任何算法。因此，我们将编程让 GMIS 实例去做这样一件事：定义一个字符串 s="hello"，要让 GMIS 的大脑连续说（在 Output 窗口显示）10 次。完成这个任务中会用到条件、循环和顺序三种结构。

### 常用本能

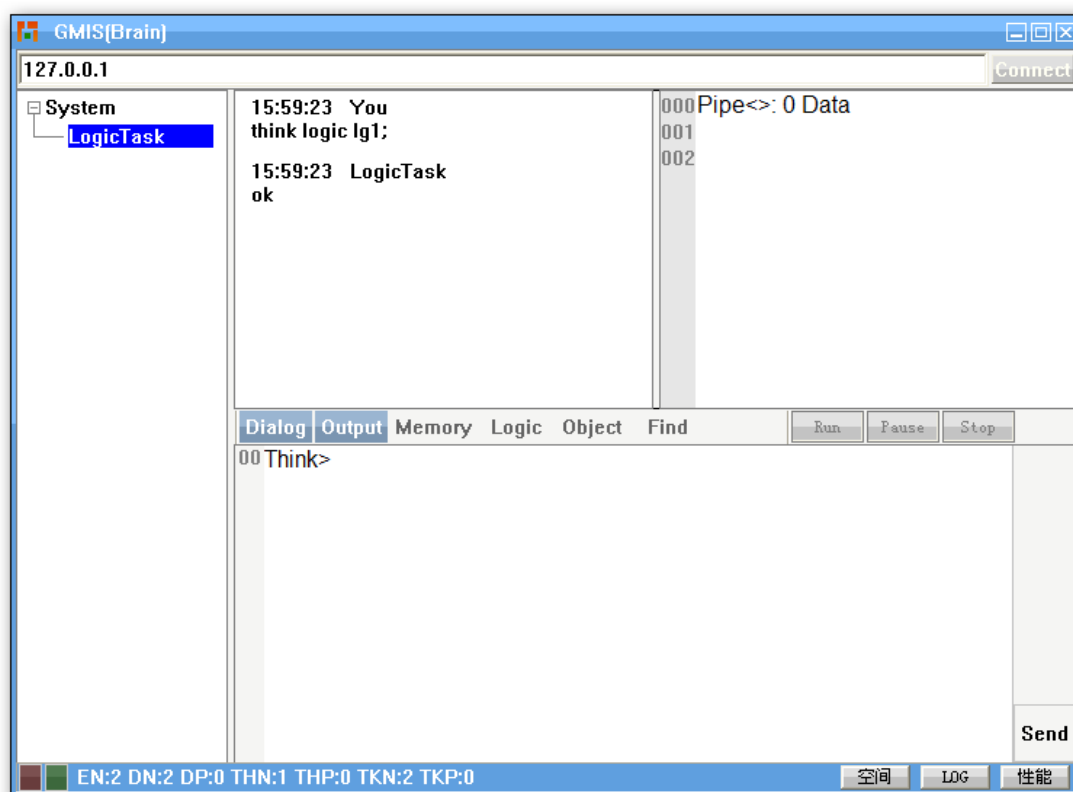
在我们具体编程之前，必须介绍一下将要用到的一些本能。

1) think logic "xxx"

这个命令将让大脑进入思考状态，准备构建一个临时逻辑，其中 xxx 是你所需要构建的逻辑名字，有了逻辑名字才能方便在其他逻辑中引用你已经构建好的临时逻辑，所以这个名字应该在当前对话中保证唯一，至于名字格式，没有特别要求，如果名字比较简单，可以无需用引号。

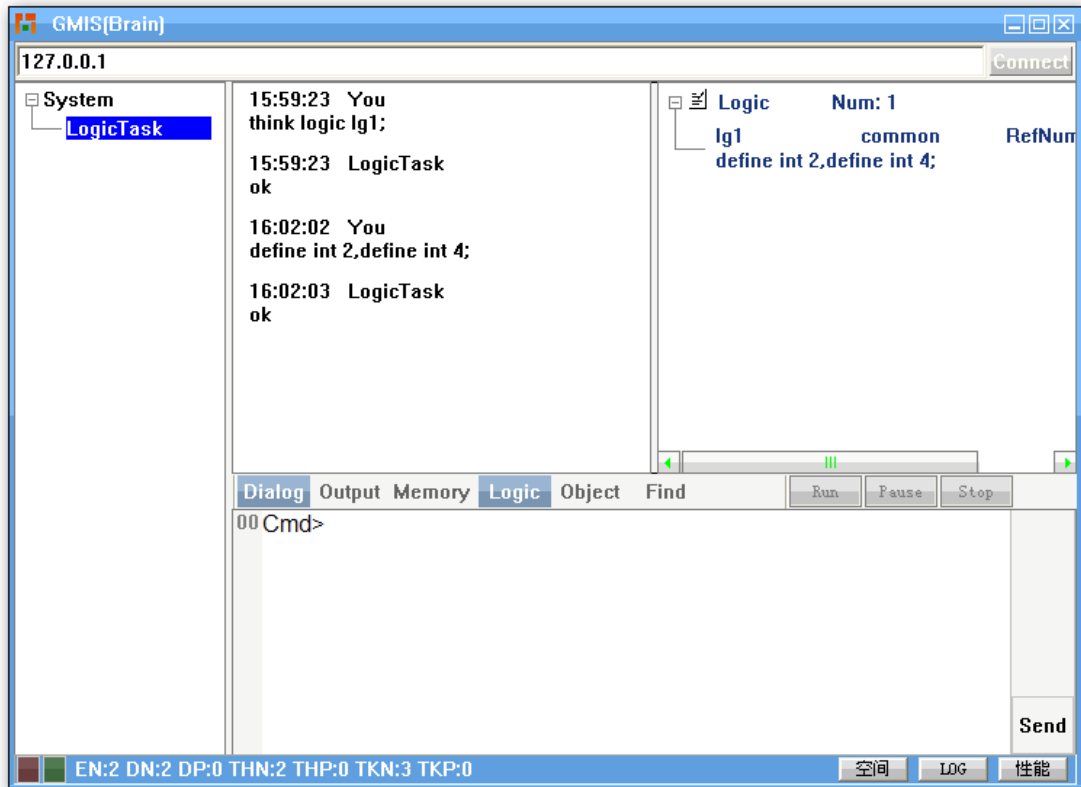
比如，我们输入：think logic lg1;

那么输入窗口会变成这样：



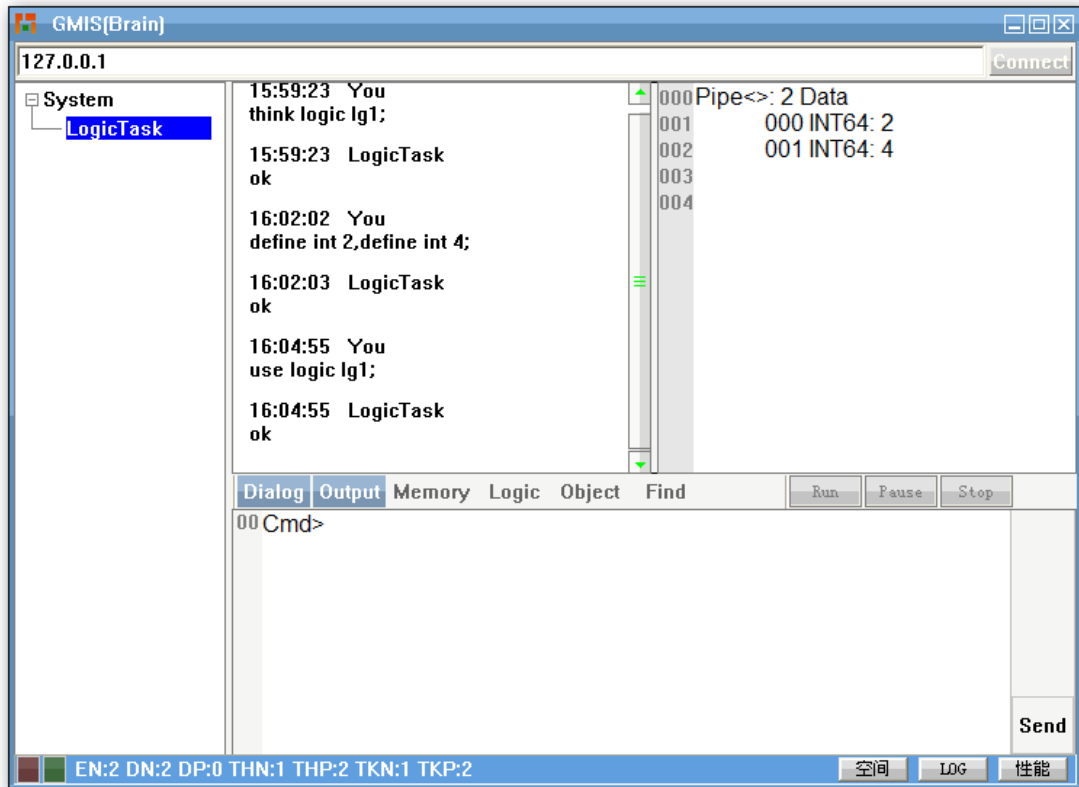
注意：在输入窗口，提示符由 “Cmd>” 变成了 “Think>”。

假设你输入其他任何命令，比如：define int 2,define int 4; 点击 “Send” 按钮之后，你看到的不是立即执行这句命令，相反，你会在临时逻辑窗口里看到你已经定义好的临时逻辑：



2) use logic “xxx”

这个命令的作用显而易见,以前例生成的临时逻辑 lg1 为例,我们输入:“use logic lg1;”,执行后, 你会在输出窗口看到执行结果,即执行了: define int 2,define int 4;



- 3) define string “xxx”  
这个命令和定义整数类似，其中 xxx 就是你所定义的字符串。比如：  
Define string “hello”；
- 4) set label “xxx”；  
这个命令是定义一个转向标签，其中 xxx 是你决定的标签名字
- 5) goto label “xxx”；  
之前使用过 set label 命令后，goto label 则是让系统转到执行标签处开始执行。
- 6) use capacitor xxx  
这个命令是定义一个名字为电容器，我们都知道电容器的特点：当电容没电时充电，当电容有电时放电，这里我们用数据管道来代替电流，也就是说，这里的电容实际上是一个数据暂存装置。
- 7) reference capacitor xxx  
这个命令是引用之前命名为 xxx 的电容，也就是说再次使用名字为 xxx 电容，如果这个电容里已经有数据，那么数据将被取出。
- 8) use inductor xxx；  
这个命令是定义一个电感器，电感器的特点是：如果流经它的电流不为零则给电感器充电，但电流继续保持数据，如果流经它的电流为零，则是电感器放电。这里我们用数据管道来代替电流，那么电感实际上也是一个数据暂存装置。

9) reference inductor xxx;

这个命令是引用之前命名为 xxx 的电感，也就是说再次使用名字为 xxx 电感。

10) use resistor xxx;

这个命令是生成一个电阻，我们都知道电阻作用就是消耗电流，这里电阻的作用就是消耗数据，其中 xxx 就是要消耗的数据个数，它可以大于实际存在的数据个数，如果 XXX=0 则什么也不做。

11) use diode xxx;

这个命令是使用一个二极管，我们知道真正的二极管是选择性打开某个分支，而这里的作用是从数据管道中取出第一个数据，然后与 xxx 比较，如果相等，则接通此数据管道，否则关闭此数据管道。

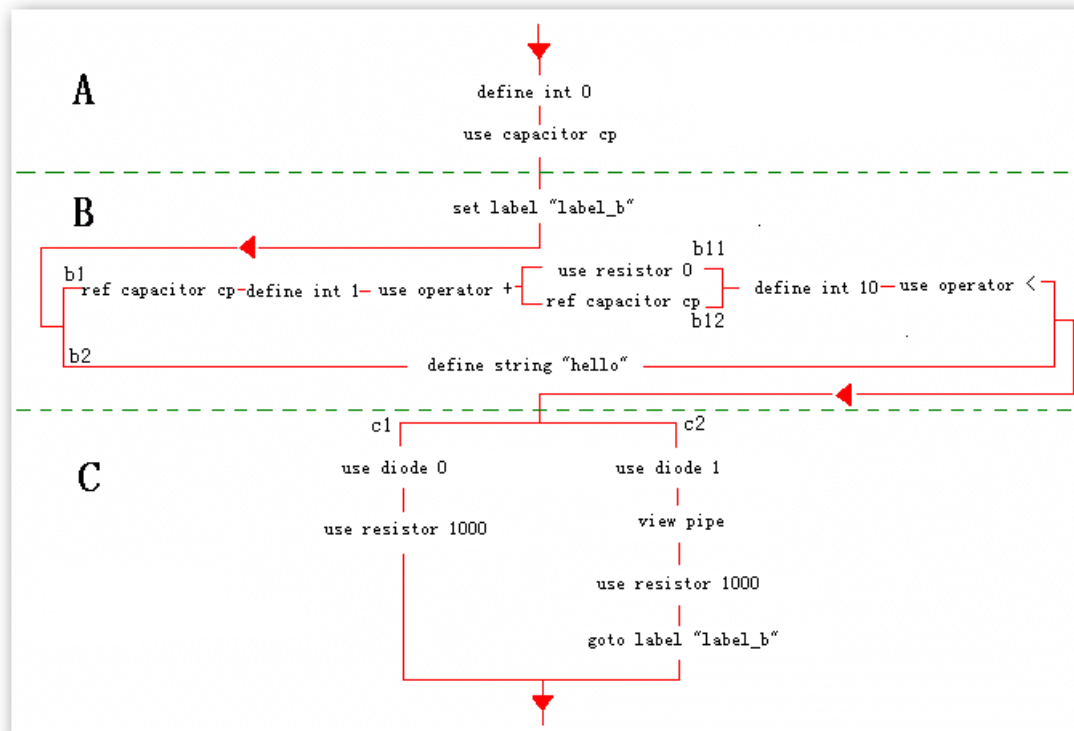
12) use operator xxx;

这里 xxx 是+、-、\*、/等运算符号，使用此命令将执行对应的运算操作。

## 循环实例

现在让我们正式编程实现这样一个任务：定义一个字符串 s="hello"，要让 GMIS 的大脑连续说(在 Output 窗口显示)10 次。通过这个例子，我们将在理论上证明用 Final C 编程的可行性。

让我们直接给出它的执行流程图：



我们把整个任务逻辑体分成 A、B、C 三部分来解释。

A 部分：定义循环的起始记数，即往管道里写入让整数 0，再让它经过电容 CP，此时电容会暂存数据管道里的数据，也就是保存整数 0，而数据管道则为空。

```
think logic a;  
define int 0, use capacitor cp;
```

B 部分：数据管道经过 A 后，先流过一个不影响管道数据的标签 label\_b，然后在流向一个并联体 b (b1, b2)，注意：并联意味着，数据管道里的数据会复制成完全相同两份的分别流向 b1 和 b2，等 b1 和 b2 分别执行完后，再按 b1 和 b2 顺序合并到一根数据管道流出。

```
think logic b;  
use logic b1, and define string "hello";
```

对于 b1, 数据管道首先流过电容 CP 的引用（实际就是再次流过 CP），电容向数据管道里放电，使其含有先前定义的循环起始记数 0，随后的 define int 1 将定义一个步长，即向管道里写入整数 1，然后使用加法让两个数相加，此时管道消耗了仅有的两个整数，并放入一个新的整数，为了以后还能使用这个整数，我们引用电容 CP 重新充电保存这个记数，此过程通过另一个嵌套的并联体 (b11, b12) 来实现。(b11, b12) 由一个零电阻与电容 CP 的引用组成，数据管道经过这个并联体后，零电阻的分支数据原封不动的流过（依然含有刚才加法得到的整数），而流过 CP 的引用时会给电容 CP 重新充电，此分支数据管道将没有数据流出，这样最后数据管道里的数据就只包含分支 b11 流出的数据。接着数据管道在 b1 分支流过一个“define int 10”行为，它向数据管道里写入循环最终的次数，此时数据管道里含有两个整数，一个是当前循环次数，一个是最终循环次数，最后它们流过一个“use operator <”行为，比较结果要么是 0，要么是 1，将作为数据管道里含有的唯一整数。

```
think logic b1;  
reference capacitor cp, define int 1, use operator +, use logic b11, define int 10, use operator <;
```

```
think logic b11;  
use resistor 0, and reference capacitor cp;
```

并联分支 b2 仅仅定义一个字符串“Hello”；



数据管道从 B 流出后，两个并联分支的数据合并，得到最终包含两个数据：整数 (0 or 1)，字符串 (Hello)，然后流向 C；

C 部分：依然是一个并联体 C (c1, c2)。两个并联分支都以一个单向导通的二极管打头，其中 c1 从管道取出第一个整数，如果等于 0 才继续执行，否则此分支执行管道失效并停止执行，c2 则反之，只有等于 1 才能继续执行。

现在程序选择那个分支，取决与 B 部分的执行结果，管道里的第一个数据为整数 0，则 c1 得到执行，此时，表示循环已经完成，我们只是简单的设置一个大电阻来清除管道里的所有数据（电阻的数字表示要清除管道里的数据个数，1000 显然已经足够大）。否则选择 c2 分支，此时管道经过二极管后只剩下一个数据即字符串“Hello”，输出字符串“hello”，然后设置一个大电阻，最后返回预先设置的标签“label\_b”，继续循环。

如果你对中学的电路知识还没有完全遗忘的话，整个逻辑循环的实现虽然烦琐，但也不难理解。

完整的逻辑清单如下：

*think logic a;*

*define int 0, use capacitor cp;*

*think logic b11;*

*use resistor 0, and reference capacitor cp;*

*think logic b1;*

*reference capacitor cp, define int 1, use operator +, use logic b11, define int 10, use operator <;*

*think logic b;*

*use logic b1, and define string "hello";*

*think logic c1;*

*use diode 1, view pipe, use resistor 1000, goto label "label\_b";*

*think logic c2;*

*use diode 0, use resistor 1000;*

*think logic c;*

*use logic c1, and use logic c2;*

*think logic e;*

*use logic a, set label "label\_b", use logic b, use logic c;*

你可以直接把上述逻辑整个拷贝，然后粘帖到输入窗口。点击按钮“send”，正常情况下，能一次性执行，如果不行（拷贝不完整），可以在输入窗口的鼠标右键弹出菜单里点击“Think”，检查到底那个子句出错，也可用点击“Analyse”检查是否正确划分了你输入的句子。

如果逻辑理解无误，输入：*Use logic e;*

点击 Send 后，你将在对话的 Output 窗口看到 10 次“hello”（由 view pipe 命令输出），你可以用任何其它命令（逻辑）来替换这个命令，那么 GMIS 就可以重复做其它任何事了。

## 高级编程

我们已经证明 Final C 理论上可以构建任何复杂逻辑，但要实现实用化，我们还需要实现以下目标：

- 能构建任意数据结构  
我们可以通过本能直接生成基本数据释放到数据管道中，但编程有时需要结构化的数据结构。我们设计的数据管道 (ePipeline) 本身是有能力表达任何数据结构的，只需要我们设计一些本能来完成此功能。
- 能逻辑化的使用数据  
目前，我们只能通过命名的电容或电感来有限的操作特定数据，相当于使用命名变量，而逻辑化的使用数据意味着我们需要类似于数组一样的东西，可以选择性或者循环使用特定数据。
- 能动态改变逻辑  
与其他高级编程语言智能按既定程序执行任务不同，智能还意味着一个大脑有能力随时随地根据前期执行结果或环境的变化而改变既定逻辑。
- 能使用外部物体  
一个大脑的智能体现在它如何管理和使用外部物体。大脑自身只提供一些构建抽象逻辑以及维持自身运转所需的基本本能。
- 能与其他机器人分工合作  
一个机器人无论再怎么智能，能力终究是有限的，如果不同机器人之间能像人类那样分工合作，那么这个世界将被彻底改变。

## 焦点

在进行复杂逻辑编程之前，我们先引入焦点概念。

随着大脑内部操作对象的复杂化，单一的命令语句已经无法完整表达操作对象，此时我们可以先把某个对象设定为焦点，作为行为的操作对象。

有些命令执行后会生成缺省焦点，我们也可以显式的用 **focus** 语句设定焦点。

当前，焦点分为以下几类：

- 1) 临时逻辑
- 2) 外部物体
- 3) 临时数据
- 4) 人

这几种焦点可以同时存在，构成一个场景，而场景对于智能和语言理解有重大意义。场景可以给行为提供主语或宾语，比如 **sb todo sth** 相当于 **focus sb, focus sth,to do**; 大脑可以从场景得到行为的抽象模式而记忆之，之后又可以通过记忆的抽象模式反推焦点对象，既能避免动词的歧义，也能帮助实现智能的主动性。

当前一切简化，我们设置焦点的目的只为顺利完成特定命令。目前显示的焦点命令有：

#### 1) 物体焦点

```
focus object xxxx ;  
focus object;
```

第一个命令是静态版本，其中 **xxx** 为需要设置为当前焦点的物体实例名字。第二个为动态版本，即需要设置为焦点的物体实例名字从数据管道中获得。

#### 1) 逻辑焦点

```
focus logic xxx;  
focus logic;
```

与物体焦点类似，第一个命令为静态版本，直接给出逻辑的实例名字，第二个则为动态版本，从数据管道中获得逻辑实例名字。

#### 2) 数据焦点

```
focus memory xxx;  
focus memory;
```

与前两个一样，分为两个版本。

注意：同一个类型当前焦点会保持到下一个新焦点设立为止。对于并联分支的执行，当一个分支暂停后，其他分支会继续执行，因此存在焦点在其它分支执行时被潜在改变的问题。

## 调试执行

**GMIS** 可以以调试状态执行逻辑，但这种调试执行与其他高级语言的调试执行有明显不同。

通常，其他高级语言的调试执行仅仅是在专用调试工具的帮助下用于程序员排错,应用程序自身无法进入调试状态。

但 **GMSI** 如同人一样，如果觉得对某个任务没有把握，可以一步一步试着来，如果觉得没问题后又会转为正常执行，如果正常执行过程中出了某种意外，那么又可以转为调试状态，一步一步的一边执行一边检查。可以说 **GMIS** 的调试执行是与正常执行同等重要的另一种执行

形态。

以上一章的循环逻辑为基础，我们稍微改一下，把循环改为无限循环，以便让我们有机会暂停，即把逻辑：

```
think logic b1;  
reference capacitor cp , define int 1, use operator +,use logic b11,define int 10, use operator <;
```

改为：

```
think logic b1;  
reference capacitor cp , define int 0, use operator +,use logic b11,define int 10, use operator <;
```

也就是把循环步长改为 0，这样永远也无法大于 10；

改过后的循环所有逻辑为：

```
think logic a;  
define int 0, use capacitor cp;
```

```
think logic b11;  
use resistor 0, and reference capacitor cp;
```

```
think logic b1;  
reference capacitor cp , define int 0, use operator +,use logic b11,define int 10, use operator <;
```

```
think logic b;  
use logic b1, and define string "hello";
```

```
think logic c1;  
use diode 1, view pipe, use resistor 1000, goto label "label_b" ;
```

```
think logic c2;  
use diode 0, use resistor 1000;
```

```
think logic c;  
use logic c1, and use logic c2;
```

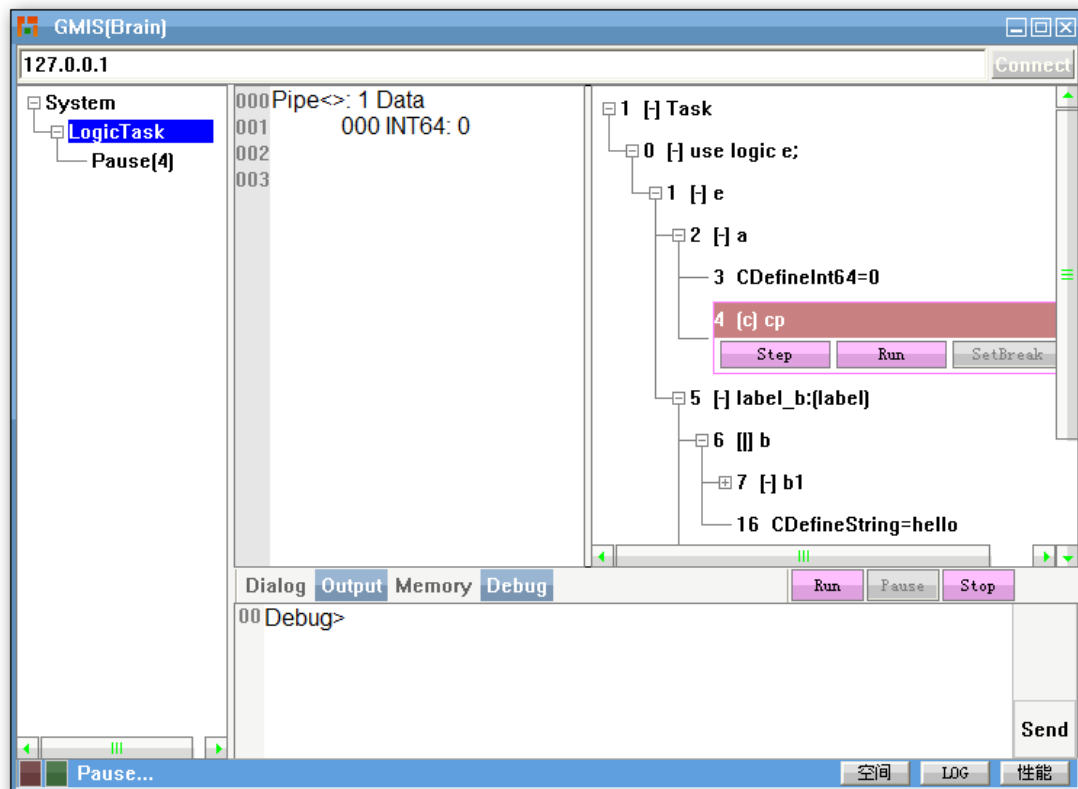
```
think logic e;  
use logic a, set label "label_b", use logic b, use logic c;
```

注意：如果你当前的对话已经拥有了上述逻辑，你可以有三个选择：

- 1) 在临时逻辑窗口点击任何一个条目，然后在工具条点击按钮清空所有临时逻辑。
- 2) 在临时逻辑窗口点击 **b11** 条目，删除此逻辑，然后只重新生成 **b11** 临时逻辑。
- 3) 拷贝以上所有逻辑，然后在主对话输入，主对话会转而生成一个新的子对话执行。

无论哪一种，做完之后，我们接着输入：“debug;” 那么当前对话转为调试状态，你可以看

到命令提示符变为：“Debug>”，然后我们输入：“use logic e;”,执行后你可以看到对话自动打开一个调试窗口：



调试窗口里含有一颗树。不要小看这棵树，这里隐藏着 GMS 日后实现自动编程的关键。

所有逻辑在 GMS 内部被组装成了一棵树来执行，我们知道一棵树可以被任意重组，依然得到一棵树，它就像人类大脑里的神经树一样重要。

当 GMS 转为调试状态执行后，我们可以暂停条目下点击 Step 来单步执行，每执行一步，可以在输出窗口看到当前数据管道所拥有的数据。

用户稍后可以点击 run 重新转为正常执行，由于是无限循环，所以我们还有机会，点击 pause 又转为调试执行。

注意：点击条目上的 run 和对话工具条上的 run 稍微有点不同，我们知道逻辑是可以同时执行两个以上的循环，点击某个条目的 run 只能让条目所在的循环重新启动正常执行，不影响其他并联分支的暂停，但是，如果用户点击对话工具条上的 run，那么就是所有分支的 pause 都将取消。

另外用户也可以点击某个条目设置为 break，那么无论正常执行还是调试执行，都会在执行到此处后暂停。

## 数据表

要构建任意数据结构和实现对数据进行任意逻辑化操作都需要用到数据表。数据表是临时记忆的一种，用来临时存储数据，但是比电容这类更灵活。

我们已经多次提到数据管道:ePipeline，它除了在执行时充当执行管道，本身也是一个存储各种数据类型的容器，如果你查看 TheorySpace\Pipeline.h，可以发现，它能压入整数，浮点数，字符串，以及其他 ePipeline，这样，构建任意数据结构实际上就是组合一个任意嵌套的数据管道。

当任务执行时，有一个 ePipeline 始终贯穿整个逻辑树，为每一个节点提供数据，同时从节点取回执行结果数据（如果有的话），但有时我们需要对取回的数据做一些特别的处理，我们不能直接在当前数据管道上操作，那会影响当前数据管道里的数据，必须把数据管道里的数据转移到一个数据表上执行操作。

数据表操作涉及一下命令：

- 1) create table xxx;  
生成一个名字为 xxx 的数据表，并默认为当前焦点数据表。这个名字应该在当前对话保证唯一，系统会做检查。生成的表可以在 memory view 中查看。
- 2) import data;  
把当前数据管道里的数据导入到当前焦点数据表中
- 3) export data;  
把当前焦点数据表的数据转入到当前数据管道中。
- 4) insert data  
当前数据管道里的数据作为一行插入到当前焦点数据表中
- 5) get data  
把焦点数据表中指定一行的数据放入到数据管道中，行号从数据管道里的第一个数据获得。
- 5) modify data  
用数据管道里的数据替换焦点数据表中指定序列行的数据，行号从数据管道里的第一个数据获得。
- 6) remove data  
在焦点数据表中删除指定序列行的数据，行号从数据管道里的第一个数据获得。
- 7) get size  
返回数据表的行数
- 8) close table;  
关闭当前焦点数据表

举例说明，假设当前执行管道里含有如下数据：

```
ePipeline{  
    int32    0           //年龄;  
    string   "jack"      //名字;
```

```

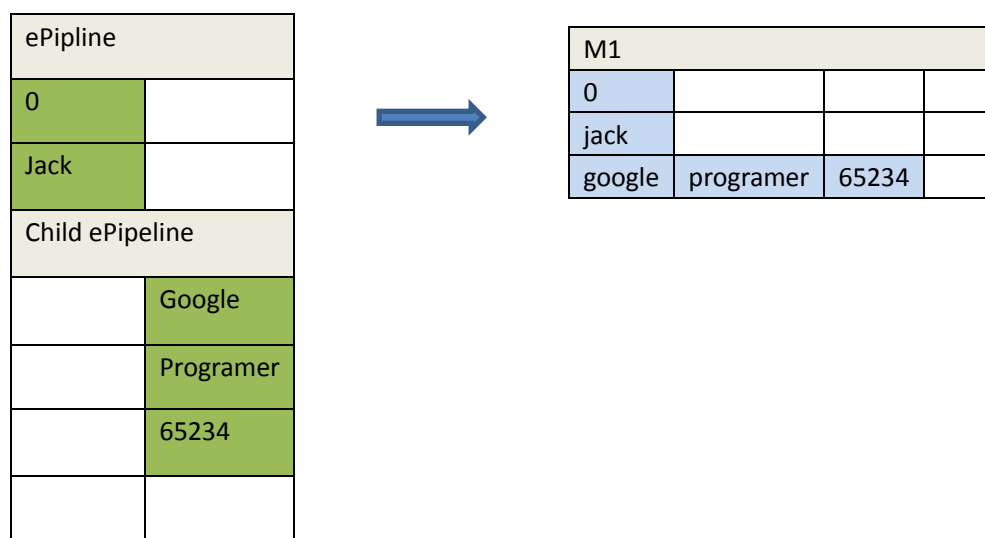
ePipeline{
    string  "google"    //单位
    string  "programer" //职位
    int32   65234       //工资
}
}

```

我们要导入一个名字为 **m1** 的数据表中，可以输入：

**Create table m1, import data;**

执行后会发生如下事情：



如图，执行管道里的每一个数据都会变成数据表 **m1** 里的一行，如果数据为 **ePipeline** 类型，里面含有多个数据，那么数据表 **m1** 里对应的行就含有多个数据。注意，转移后，原数据管道将不再保有数据，也就是为空。

相应的，如果想把数据表 **m1** 里的数据转移回到当前数据管道里，可以输入：**export data;** 此时，数据表每一行的数据会依次压入到数据管道中，如果这一行数据个数有多个，则会自动用一个 **ePipeline** 封装后压入。导出数据后，数据表则为空。

我们可以用 **get size** 命令得到 **m1** 含有的行数，通常，也用这个方法得到当前执行管道里的数据个数（注意：不包括子管道里的数据个数，比如上述例子答案为 3 而不是 5）。

如果当前数据管道含有数据：

```

ePipeline{
    int 4,
    int 5
    int 6
}

```

现在，我们想把此管道里的数据插入到 m1 末尾，可以输入：

*Use capacitor c1,Get size , reference capacitor c1,insert data;*

M1 变为：

M1			
0			
jack			
google	programer	65234	
4	5	6	

为什么要用到电容呢，因为我们需要把 m1 的当前行数放到当前数据管道里的作为第一个数据。

如果我们要返回指定行的数据，则可如下使用：

**Define int 2,get data;**

执行后，“google” “programer” “google”将作为三个字符串依次压入到当前数据管道中。注意：这里与 **export data** 区别，后者会把三个数据装入到一个 ePipeline 中压入当前数据管道。

其他命令类似于 **get data**，就不一一举例了。

剩下的问题是：我们如何构建出一个复杂数据呢？比如之前提到的数据：

```
ePipeline{
    int32  0           //年龄;
    string  "jack"      //名字;
    ePipeline{
        string  "google"    //单位
        string  "programer" //职位
        int32   65234       //工资
    }
}
```

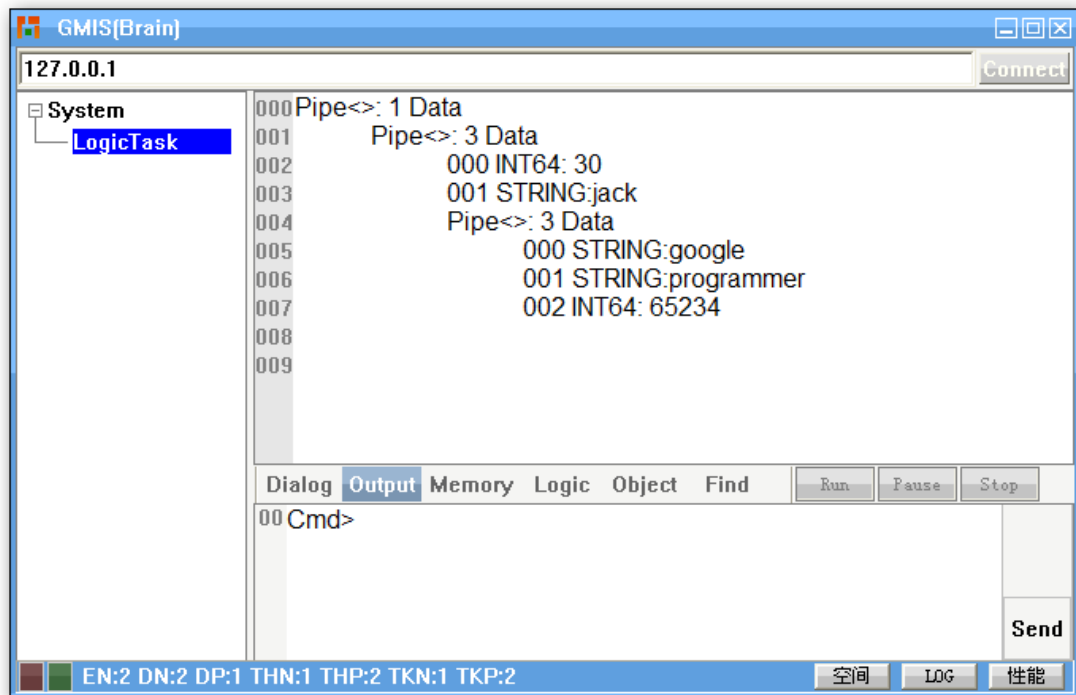
如果要人工构建，有了数据表，我们可以这么做：

*Think logic lg1;*

*create table m1,define int 30, define string jack, import data, Get size, define string "google",  
define string "programmer", define int 65234,insert data, export data;*

执行此逻辑可以看到输出为：





构建好的数据可以作为一个整体，插入到数据表中，反复如此，就可以构建出任意复杂的数据结构。

## 运行时改变逻辑

在调试执行一节，我们已经提到，GMIS 在内部把所有逻辑组装成一颗树来执行。这也意味着我们可以很方便的在执行期间在这棵树里插入一个分支，或者删除一个分支，从而改变整个执行逻辑。据我所知，这种能力目前是 GMIS 独有的。拥有这种能力，意味着机器人的行为可以随时根据环境的变化而改变。

运行时改变分为两种情况，一种是程序性改变，指在编程时就根据前半部分逻辑执行结果而决定下一步如何改变逻辑。比如，一个经理的工作逻辑负责给手下同事安排任务，并且监控其执行，一旦发现某个同事空闲就会再次给这个同事安排任务。经理无法预先知道谁会有空，只是肯定知道其中有人会有空，有空闲时该怎么办？于是会预先构建一个插入逻辑，当条件满足后就插入既定的行为。程序性改变只适合插入或移除逻辑。

另一种是临时性改变，这种改变需求通常来自外界而不是自身逻辑，此时逻辑已经完全编译执行，每个逻辑元素都有唯一的编号作为地址，因此，根据此地址添加或移除任何逻辑分支并不是一件难事。

目前我们只讨论第一种实现，即程序性运行时改变逻辑。

### 1) 插入逻辑

插入逻辑意味着在已经编译好的某个串联和并联体内插入某个分支,要做到这个,必须首先知道插入点的位置。

这个是通过命令: `Use logic lg1@InstancName` 来实现

在前面的示例中,我们已经使用了很多次 `use logic` 命令,但它的完整形式:

```
Use logic logicName@InstanceName:comment;
```

这里, `logicName` 当然是我们生成的临时逻辑名, `InstanceName` 则是这个临时逻辑名的实例名,因为我们在不同地方使用相同的临时逻辑时,实际上是使用了它的不同实例,只是通常我们并不需要这个实例名,所以可以省略。我们还可以在后面跟一个注释,简单解释一下这个临时逻辑是做什么的,同样是可选的。

比如有一个临时逻辑 `lg1`,我们可以这么使用:

`Use logic lg1@worklogic:循环逻辑;`

`Use logic lg1:循环逻辑;`

一个临时逻辑只有拥有实例名后,才可以在之后通过 `focus logic` 命令显式设置为当前焦点逻辑,然后我们才有可能针对这个逻辑实例执行其他操作,比如插入。

命令: `Insert logic xxx;`

这里的 `xxx` 是一个临时逻辑名,它的作用是把名字为 `xxx` 的临时逻辑,插入到当前焦点逻辑中,成为其 `child`。

来看一段示例:

```
think logic a;
define int 3;
think logic b;
define string "hello world";

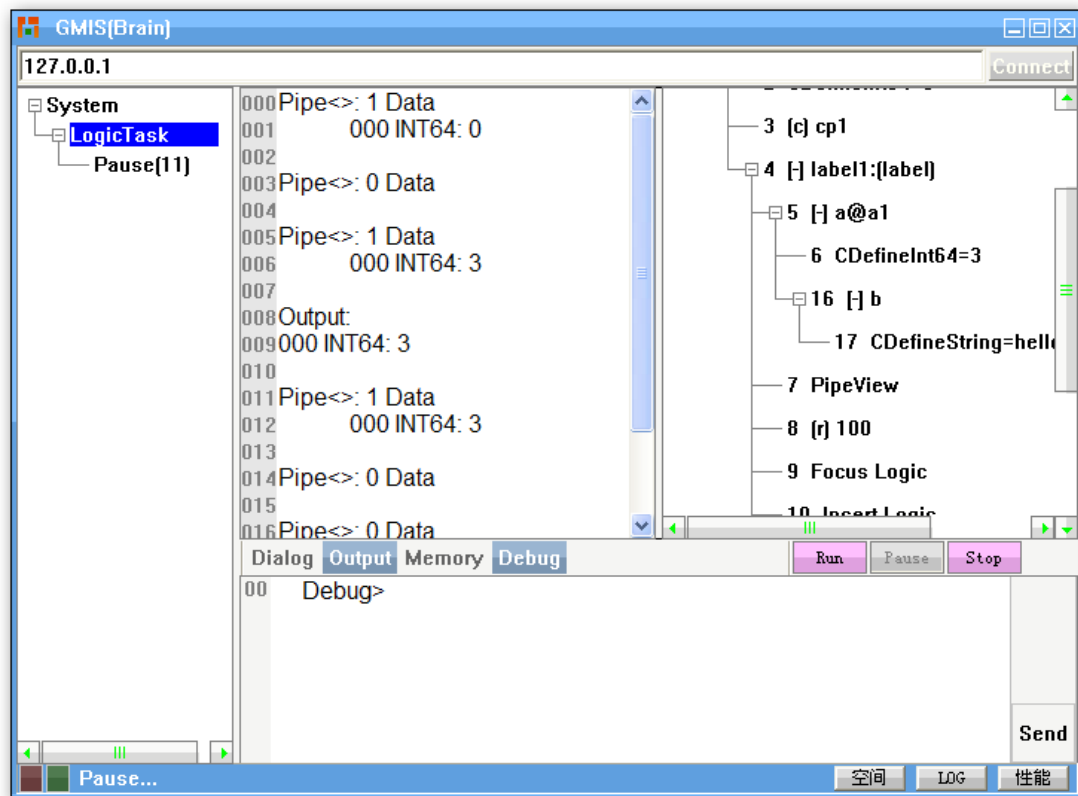
think logic c;
define int 0,use capacitor cp1,set label label1,use logic a@a1, view pipe,use resistor 100,focus
logic a1,insert logic b,reference capacitor cp1,use diode 0,define int 1,reference capacitor
cp1,goto label label1;
```

这段逻辑作用是先执行 `logic a`,然后把 `logic b` 插入 `logic a`,再执行 `logic a` 一次,结束。观察这段逻辑执行的最好方法是调试执行这段逻辑。

输入 “`debug;`”,执行,然后输入: “`use logic c;`”,执行,可见调试窗口。

然后单步执行,直到执行到 `ID=10` 的条目 (`insert logic`),注意观察 `ID=5` 条目的变化,当执行

完 10 后，条目 5 由只包含子条目 6，变成了如下：



可以看到，条目 5 多了一个 ID=16 的子条目，它正是我们插入的逻辑 b。

## 2) 删除逻辑

逻辑逻辑需要使用命令：Remove logic;

这个命令没有参数，它会向根部寻找当前焦点逻辑，然后删除自身所在的分支。

让我们紧接着上例，输入以下逻辑：

```
think logic d;
```

```
focus logic e1,define string "hello world",view pipe,remove logic;
```

```
think logic e;
```

```
use logic a,and use logic d;
```

```
think logic f;
```

```
define int 0,use capacitor cp1,set label label1,use logic e@e1,view pipe, use resistor 100,reference  
capacitor cp1,use diode 0,define int 1,reference capacitor cp1,goto label label1;
```

在这个例子中，先执行 logic e，显示结果，logic d 会删除自己，当再此执行 logic e，显示结果中会发现 logic d 已经不存在了。读者可以自行通过调试执行的方式一步一步的观察变化。

## 使用外部物体

和人类通过使用工具来扩展自己的能力类似，GMIS 也有一些本能可以帮助它使用外部物体——这里主要指的是其他用传统方式编程得到的可执行代码，从而实现在尽可能控制本能数量的前提下实现无限功能。

计算机技术发展到现在，可以执行的应用代码，包括各种语言编写，适合各种操作系统的，数以千万，GMIS 要能全部调用它们并不是一件简单的事。

我们既不可能要求别人大规模的修改代码去适应你的系统，也不可能完全一点不变的就直接使用这么多旧代码。

GMIS 的给出的解决方案是尽可能得把这种变化控制在自身能做的范围内，之前我们提到过执行器，首先，我们会尽可能的给现有代码配套一个执行器，你是 JAVA 代码我们就配 JAVA 的执行器，你是 C++的我们就配 C++的执行器。

然后，我们会简单的封装这个代码，封装的目的不涉及原有应用代码内部的执行，仅仅是赋予它从数据管道接受参数，以及把执行结果保存到数据管道的能力。

执行时，GMIS 会向空间请求执行某个外部物体，而空间会自动根据代码类型调用对应执行器执行指定物体，然后返回结果，对于用户来说，这一切都是透明的，用户无需关心如何执行，只需要知道会用到什么物体即可。

下面，我们以在 windows 平台执行 C++代码为例，我们把应用代码封装成一个标准格式的 class，然后编译成一个 DLL，这个 DLL 就是我们得到的外部物体。

### 1) 构建外部物体

如果你稍懂一点 C++，就会发现封装其实是一件很简单的事，最好的解释是独自看示例源代码，由于这里给出的代码涉及到其它基类，而我们又不想转移焦点去解释，所以只大致给出一个框架。如果你不懂，也不用过分担心，我们自己在现实中使用的东西，几乎都不是我们自己制造的，别人做好的，我们能用好就行了。现在这里假设你懂一点 C++。

首先，我们设计一个标准的 C++类，它承继自基类 Mass，在这个类里你主要需要实现的就是 `virtual bool Do(Energy* E)`，这个虚函数负责执行你所需要封装的其他代码。

```
class MyObject : public Mass {
```

```
virtual bool Do(Energy* E){
```

```
    ePipeline* Pipe = (ePipeline*)E;
```

```
    ...// 你可以从 Pipe 里取出数据执行任何你想做的事，然后把结果返回给 Pipe
```

```
    Return true; //or false  
}
```

然后，我们把这个类编译成一个 DLL，并让 DLL 导出如下四个标准函数：

```
extern "C" _LinkDLL Mass* CreateObject(std::string Name,int64 ID=0){ return new  
MyObjec(Name,ID); }  
  
extern "C" _LinkDLL void DestroyObject(Mass* p){ delete p; p = NULL; }  
  
extern "C" _LinkDLL DLL_TYPE __stdcall GetDllType();  
  
extern "C" _LinkDLL const TCHAR* GetObjectDoc(){ static const TCHAR* Text = "This is MyObject's  
document"; return Text;}
```

这里，前两个函数用户可以照搬，GetDllType() 主要是给出 DLL 类型，用来匹配对应的执行器，比如你连接的 C 运行时库是调试版的，那么就用调试版的执行器来执行，这也是一个标准函数，用户需要自行编写是 GetObjectDoc()——向未知用户介绍你所封装物体的实际功能和用法。

这样一个外部物体就基本完成了，显然它不仅仅可以用来承继旧世界，也同样可以用来生成新物体。也许有人会说 Final C 本身执行效率不够高，但大脑的作用不在于效率，而是灵活的控制，我们可以把需要极高效率的代码封装成外部物体来执行。

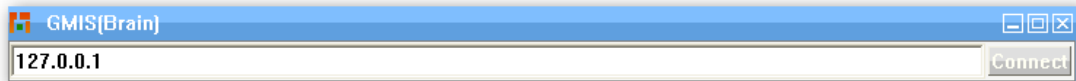
## 2) 把外部物体导入数字世界

还记得之前提过的空间吗？由于目前的 GMIS 实例并没有配备信息器官，不能感知现实世界，所以我们必须给它配备一个数字空间，这样它可以直接与这个数字空间交互。

现在的数字世界一片荒芜，我们需要把制造的外部物体导入到数字世界。

首先我们要把你所封装好的 DLL 打包成 ZIP 文件，虽然可能多数外部物体只是一个单文件，但也有些物体会会有自己的辅助文件，所以我们必须用一个压缩包的形式导入。也许未来我们应该给这种压缩包取一个单独的后缀名，就像 Android 对它的 APP 使用 APK 后缀一样，但现在一切从简。

然后，让 GMIS 实例连接到数字空间，默认地址为 127.0.0.1, 即所需要连接的空间入口程序在本机。



连接后，点击右下的“空间”按钮，会看到一个三维场景，这个是试验性质的，我们最终希望每一个人的空间都能相互连接，变成一个三维数字世界，目前无需理会。



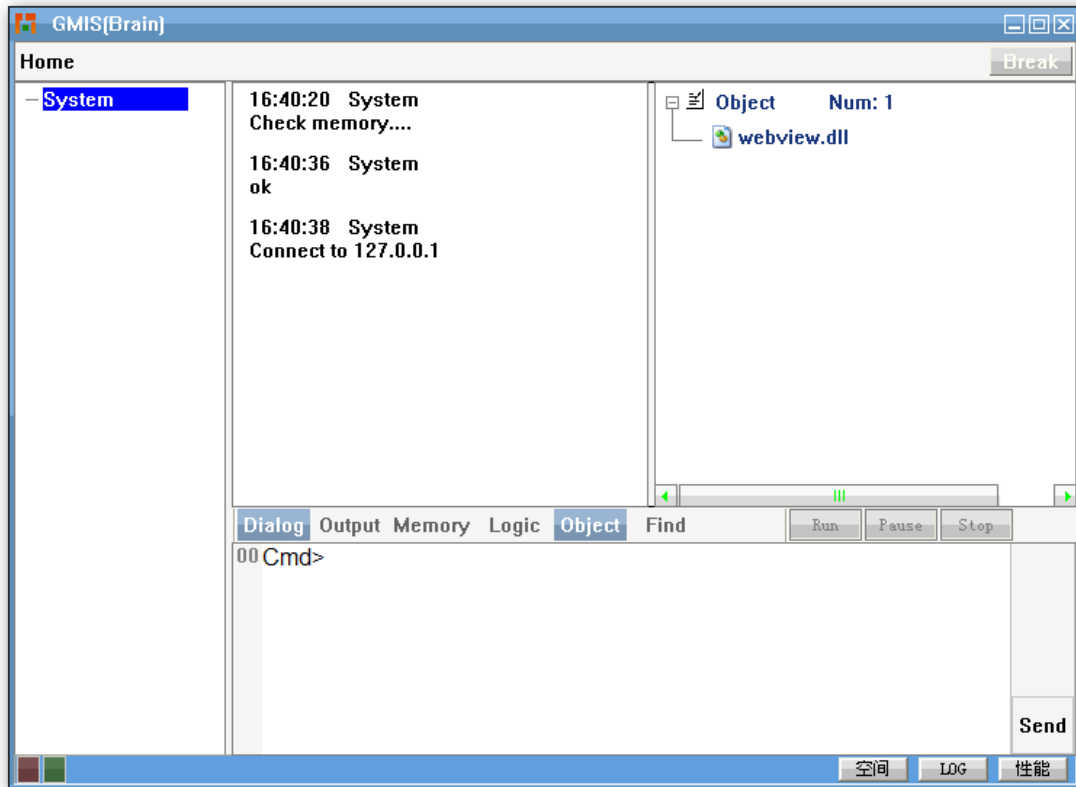
点击“ImportObject”，弹出一个 FileDialog，选择指定的 zip 包后，外部物体即被导入到当前空间中。系统会缺省生成一个三维物体来表示。

### 3) 使用外部物体

就像人使用外部物体，你首先需要在周围寻找，然后选定。GMIS 也同样需要这样做，只不过现在必须你帮它完成。首先我们在当前三维场景中点击鼠标右键，此时会有一个 ListView 显示当前房间内所有的物体列表，然后点击条目，会有一个工具条，点击工具条里的 select，那么此物体即被选中进入大脑的临时记忆里。



如上图，我们点击“Select”后，返回对话界面，打开 ObjectView 会看到：



我们可以在 ObjectView 里看到这个物体，接下来就可以直接使用了。涉及外部物体使用的本能命令有如下几个：

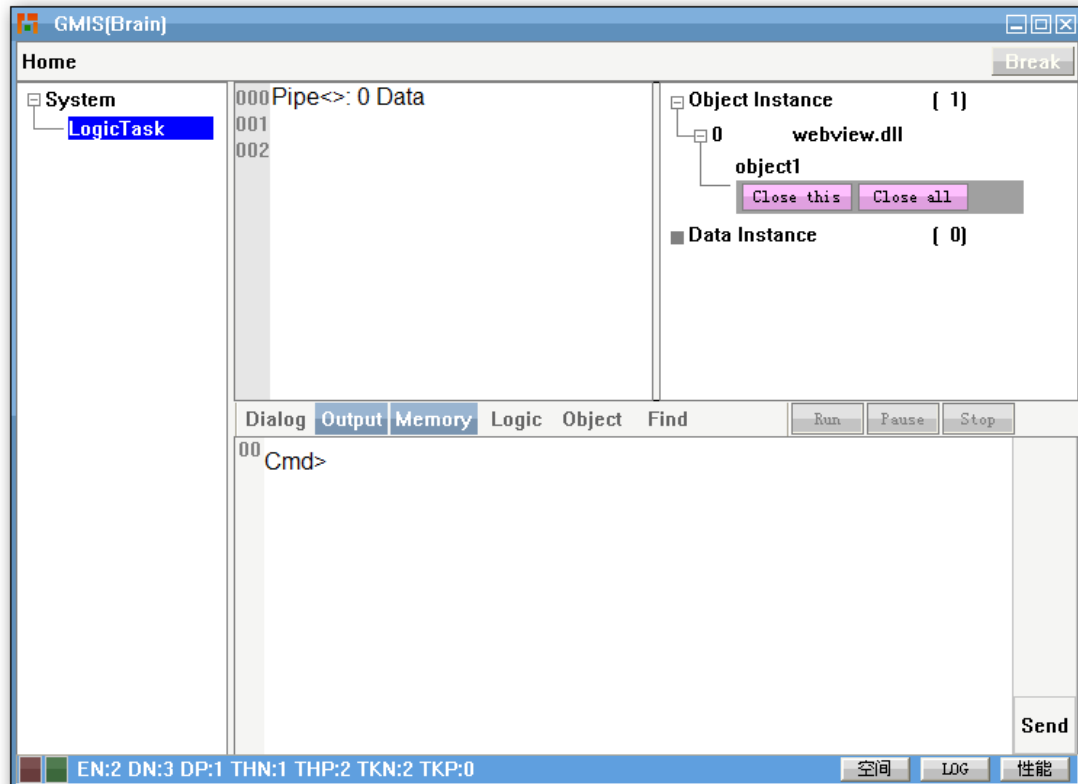
- 1) start object “ObjectName” ;
- 2) name object “ObjectInstanceName”
- 3) focus object “ObjectInstanceName” ;
- 4) Get object document;
- 5) use object;
- 6) close object;

start object 负责生成一个外部物体实例，与现实中的物体使用不同，GMIS 选中一个外部物体后，可能同时需要使用多个，每一个都是此物体的一个实例，为了能区别不同的实例，我们必须给这些实例命名，当 start object 执行后，生成的物体实例会成为缺省的焦点物体，然后我们紧接着使用 name object 就可以给这个实例命名。

比如输入并执行：

```
Start object “webview.dll”, Name object “object1” ;
```

你会在 memory view 看到：



Focus object 命名则用来显式的改变当前的焦点物体。之前在焦点一节已经提到过，焦点可以是静态的也可以是动态的，比如：

静态方式：Focus object “object1”；

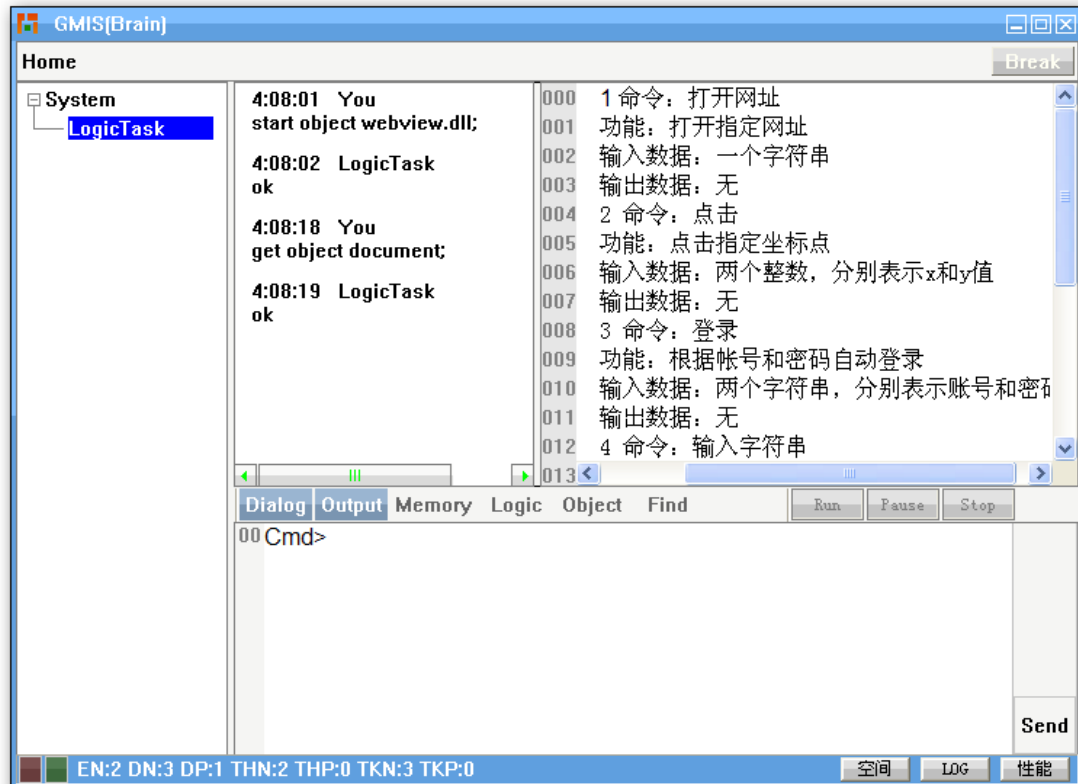
动态方式：define string “Object1”, focus object;

实际上，当我们 start object 后，新生成的物体实例将默认为当前焦点，所以我们才可以立即使用 name object 命令给当前焦点物体命名。当前焦点物体在下次改变之前都会继续存在。

现在的问题是，我们如何知道一个外部物体含有什么功能，特别是这个物体很可能不是我们自己制造的。

很简单，我们使用命令 “Get object document;” 来获得信息：





可以看到，在 output 窗口，我们得到了其所有功能的说明，以第一个命令：“打开网址”为例，我们可以如下使用：

```
define string “打开网址”, define string “www.baidu.com”, use object;
```

输入并执行上述逻辑后, 可以看到如下结果:



WebView.dll 实际简单封装了 WebBrowser 控件。

现在你可以随意用 Use object 命令来组合文档给出的命令。由于外部物体都是执行器执行的, 不管多么离谱的逻辑, GMIS 大脑都不会因此损坏。

GMIS 使用外部物体还有很多细节需要去完善, 目前只是想证明 GMIS 可以通过这种方式去扩展自己的能力。当前, 我们需要利用互联网上的已经有的开源应用去封装更多的物体, 以便不会 C++编程的人也可以直接尝试用 Final C 编程达到自己想要的结果。

## 与其他机器人合作

与其他机器人合作的原理和使用外部物体类似, 只不过是把外部机器人视为一个物体, 然后传递命令给它, 它像一般的外部物体一样, 执行完后返回结果。

不同的是, 对方可能会审查你的命令, 决定是否和你合作。由于这个需要多机联合测试, 留给下一版本去做, 这里不给出例子。

# 应用实例

## 定时点击网页

下面我们用 `webview.dll` 物体演示如何定时点击网页的某个链接或者按钮，你可以自行扩展做其他事，比如抢购秒杀，定时发微博，或者顶帖之类。

如果遇到新的本能命令，可以参考附录里的本能列表。如果对 `webview.dll` 功能不了解，可以参考之前的 `get object document` 命令来了解。

下面直接给出逻辑：

```
think logic lg1;  
start object webview,define string "view",name object,define string "打开网址",define string  
"www.ifeng.com",use object;
```

```
think logic lg2;  
input num "请输入 x 坐标:",input num "请输入 y 坐标: ",use capacitor point,input text "请输入  
时间: (比如 23:00:00)",use capacitor time,set label label1,use resistor 10,reference capacitor  
time,use resistor 0,and reference capacitor time,get time,use operator ==, use logic lg3,and use  
logic lg4;
```

```
think logic lg3;  
use diode 1,define string "刷新",use object,define string "点击", reference capacitor point,use  
object;
```

```
think logic lg4;  
use diode 0,wait second "0.1",goto label label1;
```

使用:`use logic lg1,use logic lg2;`

## 自动玩牌

为了更完整的演示 `GMIS` 的逻辑控制能力，我们让 `GMIS` 自动玩一款网上的德克萨斯牌游戏，当然我们必须把这款游戏封装成物体，来接受 `GMIS` 的控制，同时也用到了 `google` 开源的 `tesseract ocr` 库，用来识别牌。

我们的任务是，让 `GMIS` 控制多个帐号，同时让打开两个以上帐号，登录此游戏，自动进入用户选定的房间，自动玩牌，每一个帐号玩指定的局数后就退出，直到所有帐号都玩完。

如果花费更多精力，我们也可以让这些帐号把所用牌型通知 GMIS，在其统一指挥下决定如何出牌，从而显著提高胜率。目前我们只想简单的展示 GMIS 逻辑控制全局的能力，如果把这个玩牌物体替换成其他机器物体或者其他 GMIS 实例，那么 GMIS 就能让这些物体和机器人分工合作去完成特定任务。

## 传统编程方式

由于这是一个复杂的逻辑任务，在设计之前，我们最好简单的用其他高级编程语言习惯设计一下，然后再翻译成 Final C 逻辑。

```
vector<Player> PlayerList;
vector<User> UserList;

int PlayerIndex = 0;
bool bHasSite = true;
int RoomID = 100;
int PlayNum = 100;
int CurPlayNum = 0;

Start thread 1.do();
thread1::do()
{
    while(1){
        if(bHasSite){
            for(int PlayerIndex=0; i<PlayerList.size(); PlayerIndex++){
                Poker = PlayerList[PlayerIndex];
                if(Poker.m_IsIdle){
                    if(UserList.size()){
                        User = UserList.pop();
                        start thread 2.do()
                    }else{
                        return;
                    }
                }
                break;
            }
        }
        }else{
            sleep(100);
        }
    }
}
thread 2::do()
{

```

```

Poker = PlayerList[PlayerIndex];
Poker.m_IsIdle = false;
if(!Poker.Login(User)){
    Poker.m_IsIdle = false;
    return;
}

if(!Poker.goto(RoomID)){
    Poker.m_IsIdle = false;
    return;
};

if(!Poker.Sitedown()){
    Poker.m_IsIdle = false;
    return;
};

for(int CurPlayNum=0; i<PlayNum; CurPlayNum++){
    Poker.PlayOnce();
}
Poker.m_IsIdle = true
return ;
}

```

可以看出，如果用传统编程来实现这个任务并不复杂，它主要需要两个数据结构：用户帐号和实际玩家，分别构成两个数组，另外一些临时变量辅助控制，然后通过两个线程函数来配合完成，其中一个线程负责分配任务，一个线程负责执行任务。

## Final C 方式

用 Final C 翻译上述代码，有两个经验。

第一个经验：采用细分法来构建逻辑。先考虑整个任务大致分成几部分，每部分用一个子逻辑，在实现每一个子逻辑时，同样如此细分，直到所有逻辑都实现。

比如：这个玩牌任务任务会大致分成三部分，第一部分准备一些数据，第二部分准备玩牌物体实例，第三部分则是具体控制玩牌。那么我第一个逻辑就可以这么写：

Think logic lg1: 开始工作;

Use logic lg2:准备帐号,use logic lg3:准备玩牌实例,use logic lg4:工作逻辑;

这里,逻辑 lg2,lg3,lg4 都还没实现，但我们可以先声明。之后实现 lg2 时，如此重复，就可以把整个逻辑很自然的写完了，不像传统高级编程语言那样在不同头文件中跳转。

第二经验：给逻辑名字标识序号。通过序号我们可以大致判断一些子逻辑的相关性。越靠近根逻辑序号应该越大，这样子逻辑可以在父逻辑序号基础上递增。

比如，同样是上述逻辑，我们可以这么写：

Think logic lg1: 开始工作；

Use logic lg100:准备帐号,use logic lg200:准备玩牌实例,use logic lg300:工作逻辑；

如果第一个子逻辑 lg100 含有其他子逻辑，那么它的子逻辑可以像 lg101 这样递增命名，而我们则可以知道在 100~200 之间的子逻辑都与准备帐号相关。当然，如果逻辑层次过多，这样命名也无能为力。这里不得不提醒一句：Final C 不是为提高程序员效率而产生的另一门编程语言，我们的最终目标是让机器人能自动编程，在大脑内部，命名其实完全可以用一个唯一的 ID 来表示，只是现在不得不人工做这些事。

整个逻辑代码如下：

Think logic lg1;

Use logic lg100:准备帐号,use logic lg200:准备玩牌实例,use logic lg300@worklogic: 工作逻辑,use resistor 10,define string "game over",output info;

Think logic lg100;

create table UserAccountList: 帐号列表,

get size,define string "帐号 1",define string "密码 1",insert data,

get size,define string "帐号 2",define string "密码 2", insert data,

get size,define string "帐号 3",define string "密码 3", insert data,

get size,define string "帐号 4",define string "密码 4", insert data,

get size,define string "帐号 5",define string "密码 5",

insert data;

Think logic lg200: 启动五个玩牌物体;

start object poker, name object player1,

start object poker, name object player2,

create table ObjectList: 玩牌物体列表,

get size,define string "player1",insert data,

get size,define string "player2",insert data,

Create table RunFlagList: 物体使用标志,

get size,define int 0,insert data,

get size,define int 0,insert data;

Think logic lg300;

use logic lg301: 控制逻辑,and use resistor 0;

Think logic lg301;  
define string "0",use cp RoomID: 预定房间号,  
use cp CurRoomID:临时存储房间号,  
define string "0",use inductor ActualRoomID:保存实际返回的房间号,  
define int 1,use inductor bHasSeat:是否有空位,  
use cp UserAccount:当前用户帐号,  
define int 1, use cp bAlive:控制逻辑是否结束,  
define int -1,use cp ObjectIndex:指向 ObjectList 的序列号,  
use logic lg302: 循环;

Think logic lg302;  
Set label b1:控制循环标签,use resistor 100,reference cp bAlive,reference cp bAlive,and use  
resistor 0,use diode 1, use logic lg303: 检查是否有空位,goto label b1:控制循环标签;

Think logic lg303: 检查是否有空位;  
use resistor 10,Reference inductor bHasSeat, Use logic lg304: 没有空座位的处理, and use logic  
lg305: 有空座位的处理;

Think logic lg304: 没有空位;  
Use diode 0,define int 0,reference inductor bHasSeat,wait second 2;

Think logic lg305;  
Use diode 1,define int 0,reference inductor bHasSeat,reference cp ObjectIndex,use resistor 100,  
define int -1,reference cp ObjectIndex,Use logic lg306: 寻找空闲的玩牌物体;

Think logic lg306:循环检查是否有空闲的物体;  
set label b306:物体列表循环标签, reference cp ObjectIndex,define int 1,use operator +, use  
resistor 0, and reference capacitor ObjectIndex, use logic lg309: 得到循环终点值, use operator <,  
use logic lg310: 条件处理;

Think logic lg309;  
focus table ObjectList,Get size ;

Think logic lg310;  
Use logic lg311: 不符合条件,and use logic lg312: 符合条件;

Think logic lg311:循环结束;  
use diode 0, wait second 1;

Think logic lg312;  
use diode 1, use logic lg313:检查是否有空闲的物体;

think logic lg313;  
focus table RunFlagList,reference cp ObjectIndex,use resistor 0,and reference cp ObjectIndex,get

data,use logic lg314: 检查空闲标志;

think logic lg314;

use logic lg315,and use logic lg316;

think logic lg315:没有空闲继续寻找;

use diode 1, use resistor 100,goto label b306;

think logic lg316:有空闲;

use diode 0, focus table RunFlagList,reference cp ObjectIndex,use resistor 0,and reference cp ObjectIndex,define int 1,modify data, focus table UserAccountList, Get size,define int 0,use operator >, Use logic lg317:没有新账号,and use logic lg318: 有新账号;

Think logic lg317: 没有新帐号就关闭控制逻辑;

Use diode 0, reference cp bAlive,use resistor 10,define int 0,reference cp bAlive;

Think logic lg318;

Use diode 1,use resistor 10,define int 0, get data,reference cp UserAccount ,define int 0, remove data,define string "worklogic",focus logic ,insert logic lg400;

Think logic lg400: 自动玩牌逻辑;

use resistor 100,reference cp UserAccount,use cp MyUserAccount,focus table ObjectList, reference cp ObjectIndex,use resistor 0, and reference cp ObjectIndex,use capacitor MyIndex,and use resistor 0,get data,use capacitor MyObjectName,use resistor 10, use logic lg401: 打开游戏网址, use resistor 10,use logic lg402: 登录用户帐号, use resistor 100,use logic lg404: 进入指定房间,use resistor 10, use logic lg403: 取得物体名,focus object,define string "HasSeat",use object,reference inductor bHasSeat,use logic lg405: 循环玩牌;

think logic lg401;

use resistor 10,use logic lg403:取得物体名, focus object, define string "OpenUrl", define string "http://apps.renren.com/boyaa\_texas/index.php?origin=103", use object;

think logic lg402;

use resistor 10,use logic lg403:取得物体名, focus object, define string "Login", reference cp MyUserAccount, use object,use logic lg500: 循环检测是否进入游戏大厅;

think logic lg403:取得物体名;

reference cp MyObjectName,reference cp MyObjectName,and use resistor 0;

Think logic lg500;

define int -1,use cp cp500: 循环变量, set label b500:循环标签, reference capacitor cp500,define



int 1,use operator +,use resistor 0,and reference capacitor cp500,define int 10,use operator <,use logic lg501:不符合循环条件, and use logic lg502:符合循环条件;

Think logic lg501:循环完成;

use diode 0,define string "SearchLabel", define string "大厅", define float "90.0",use object, use diode 0,input num "还没登录到大厅, 请人工登录, 完成输入任意数字",use resistor 10;

think logic lg502;

use diode 1,define string " SearchLabel ", define string "大厅", define float "90.0",use object, use logic lg503:没有进入大厅,and use logic lg504: 已经进入大厅;

think logic lg503;

use diode 0,use resistor 10,wait second 2, goto label b500:循环标签;

think logic lg504: 刷新;

use diode 1,use resistor 10,define string "已经登录到大厅",output info, use logic lg403:取得物体名, wait second 5;

think logic lg404;

use logic lg610:定义房间号, use resistor 10,use logic lg620:进入房间, use logic lg630:验证房间号, use logic lg640:确定是否正确进入房间;

think logic lg610;

use resistor 10,reference cp RoomID,reference cp RoomID,and use resistor 0,define string "0",use operator ==,use logic lg611:房间号无效, and use logic lg612:当前房间号有效;

think logic lg611;

use diode 1,reference cp RoomID,use resistor 1,input text "请输入房间号", reference cp RoomID,and reference cp CurRoomID;

think logic lg612:把房间号存储在当前房间号里;

use diode 0,reference cp RoomID,reference cp RoomID,and reference cp CurRoomID;

think logic lg620: 自动进入指定房间;

use logic lg403:取得物体名, focus object,define string "ClickLabel ",define string "游戏场", use object,wait second 1,use resistor 10,

use logic lg403:取得物体名, focus object,define string "ClickLabel ",define string "初级场", use object,wait second 1,use resistor 10,

use logic lg403:取得物体名, focus object,define string "ClickLabel ",define string "牌场输入框", use object,wait second 1,use resistor 10,

use logic lg403:取得物体名, focus object,define string "InputText",reference cp CurRoomID,use object,wait second 1,use resistor 10,

use logic lg403:取得物体名, focus object,define string "ClickLabel ", define string "牌场搜索", use object,wait second 2,use resistor 10,

use logic lg403:取得物体名, focus object,define string "DbClickLabel ",define string "牌场选择",  
use object, wait second 5,use resistor 10;

think logic lg630;

use resistor 10,use logic lg631,and use logic lg632,use operator "==";

think logic lg631: 预定房间号;

reference cp RoomID,reference cp RoomID,and use resistor 0;

think logic lg632: 实际房间号;

use logic lg403:取得物体名, focus object,define string "GetRoomID",use object;

think logic lg640;

use logic lg641: 等于,and use logic lg642: 不等于;

think logic lg641;

use diode 1,define string "已经进入房间",output info;

think logic lg642: 与返回的实际房间值比较;

use diode 0,use logic lg643,and use logic lg632,use operator "==",use logic lg644,and use logic lg641;

think logic lg643;

reference inductor ActualRoomID,reference inductor ActualRoomID,and use resistor 0;

think logic lg644;

use diode 0,input num "请人工选择房间, 完成输入任意数字",use resistor 10,use logic lg632,reference inductor ActualRoomID,use resistor 10;

think logic lg405: 循环玩牌;

Define int -1, use capacitor Index, set label b405, reference capacitor Index,define int 1,use operator +, use resistor 0, and reference capacitor Index, define int 10, use operator <, Use logic lg406: 不满足条件,and use logic lg407: 满足循环条件;

Think logic lg406:玩牌循环结束;

use diode 0, use resistor 100, define string "玩牌结束",output info,use logic lg403:取得物体名, focus object,define string "ClickLabel",define string "站起", use object,use logic lg403:取得物体名, focus object,define string "ClickLabel",define string "大厅", use object,wait second "2.8",focus table RunFlagList,reference cp MyIndex,define int 0,modify data, use logic lg409:循环检查是否有座位, focus logic worklogic,remove logic;

Think logic lg407;

use diode 1, use logic lg408:玩一局牌, goto label b405;

*Think logic lg408;*

*use resistor 100,use logic lg403:取得物体名, focus object, define string "PlayRound", use object, output info, use logic lg403:取得物体名,focus object,define string "HasSeat",use object,reference inductor bHasSeat;*

*think logic lg409:检查座位;*

*set label b409,use resistor 10,reference inductor bHasSeat,reference inductor bHasSeat,and use resistor 0,use logic lg410,and use logic lg411,use resistor 10;*

*think logic lg410;*

*use diode 0,use logic lg403:取得物体名,focus object,define string "HasSeat",use object,reference inductor bHasSeat, and use resistor 0,use diode 0,wait second "2.4",goto label b409;*

*think logic lg411;*

*use diode 1,use resistor 10;*

注意：使用前请用你自己的实际帐号替换逻辑代码中出现的“帐号\*”和“密码\*”。

当你把 pokerd.dll 选择为当前物体后，输入：use logic lg1;

## 记忆与学习

当第一次启动 GMIS 原型时，你会看到一个初始化过程，因为最开始所有本能都是以 ID 形式存在，我们必须用至少一种自然语言去标记它，这样用户才有可能使用它，这就涉及到了大脑的记忆和学习。

记忆和学习其实是一个庞大的课题，目前的方案并不没有反映最新研究结果，也并没达到最终目标，但具有一定的实用性。

让我们先了解目前可用的部分。

## 记忆单词

如果一个 token 要被用来构建新命令，那么必须先让 GMIS 学习其词性，如此才有可能通过语法检查。显然，词性这类知识对于一个初级机器人来说太高级了，但为了规范用户的使用习惯，避免以后一个已经被训练好的机器人完全重新学习，我们只能强行置入。

命令：Learn [speech] “token”；

其中 speech 对应你所需要学习“token”的具体词性，比如 Learn verb “set”； 或 Learn noun “time” 。

目前允许的词性有：token、pronoun、adjective、numeral、verb、adverb、article 、preposition、conjunction、interjection。

实际上，我们目前只能用到 verb 和 noun， token 也是一种词性，用来记忆特殊符号。

示例：

```
learn verb “open” ;
```

```
learn noun file;
```

## 记忆文本

命令：

- 1) Learn text “string” ;
- 2) Learn text ;

有两个版本，第一个版本是直接记忆引号包围的字符串，第二个版本则是生成一个子对话要求你输入要记忆的文本。

这里的文本会被分解成单个字符，然后根据标点符号给出的逻辑层次记忆。

示例：

```
Learn text “hello world” ;
```

## 记忆逻辑

命令：Learn logic “xxx”;

这里的 xxx 是一个临时逻辑名。我们之前已经提到如何生成一个临时逻辑，GMIS 可以根据逻辑名存储指定临时逻辑为长期记忆。

为了之后回取使用，会继续要求用户输入文本来描述这个逻辑，GMIS 会把此文本记忆与逻辑记忆关联起来。

这个本能实际上还需要更多的研究，因为临时逻辑也分为几种情况，有些临时逻辑本身并不独立，依赖其他逻辑或者外部物体，那么记忆和回取使用时需要解决更多问题。

## 学习新行为

现在,我们可以用 Final C 编程得到各种临时逻辑,我们也可以把这些逻辑记忆为长期逻辑,但是要使用长期逻辑,需要我们搜索其描述,然后从搜索结果中选择正确的一个才能使用,这显然不够方便。

对于某些行为逻辑,它已经表达一个有意义的行为,那么我们希望它能像本能一样通过命令语句调用,这就需要先学习它。

当然,首先你需要先编写出这个临时逻辑,然后还要给这个临时逻辑命名一个行为语句,比如你有一个打开文件的逻辑,那么你可能想这么使用: open file, 接下来你就需要先记忆动词 open, 然后记忆名词 file, 最后把“open file”的记忆与临时逻辑的记忆关联。

命令: Learn action “xxx”;

这里 xxx 是一个临时逻辑名, GMIS 会先把指定的临时逻辑记忆为长期记忆, 然后要求你输入新的命令文字, 比如“open file”, GMIS 会验证是否符合谓语+宾语的语法, 如果没有问题, 记忆这个命令, 然后把这个命令与之前记忆的临时逻辑或命令 ID 关联。

随后, 你简单输入“open file”就可以执行一个复杂逻辑了, 这是一个非常重要的功能, 它能帮助用户去训练和提高 GMIS 的能力, 而不是由程序员提供一个千篇一律的复制品, 即使拥有同样的本能, 每一个 GMIS 实例的能力都可能不一样。

遗憾的是, 目前此命令的实现还需要进一步完善, 主要是外部物体依赖的问题, 如果你依赖外部物体, 那么你学习的命令就不会简单的被执行。当然, 如果对于一些专用机器人, 它不涉及外部物体的使用, 那么此命令还是具有实际意义的。

## 记忆物体

如果有需要, 我们可以让大脑记忆一个外部物体, 包括其位置, 以及使用描述, 这样可以在需要的时候通过搜索记忆即可直接使用。

使用命令: Learn object “ObjectName”;

与学习逻辑类似, 执行这个命令后, 会要求用户继续输入文本来描述这个物体, 最后大脑会把文本记忆与之前存储的物体关联起来。

如果这个物体是按前述规则生成的 DLL, 那么系统会自动调用这个 Object 的内置文档到当前编辑区, 以便用户可以这些文本为基础继续编辑作为对这个物体的描述。

注意: 当前这个命令意义不大, 用户最好忽略, 因为 GMIS 的记忆模型还没完全定型, 目前的实现仅仅是实验性质的。

## 学习记忆

学习记忆是指对已经存在的记忆用另一种记忆去标记,这种能力对于一个智能体来说非常关键。因为信息有不同的表现形式,比如视觉和听觉得到的信息不同,但可能都指向同一个物体,这个时候就需要把这两种信息关联起来。又如,我们学习外语时,会用新语言的单词去与自己母语中对应意思的单词相关联。

命令: Learn word memory “xxx”;

这里的 xxx 是已经记忆好的东西,目前只允许是一个单词,未来可以是命令,还可以只是一种描述。系统会自动分析 xxx 是不是一个单词,然后要求用户输入新的描述,新的描述应该和旧描述对等,比如你旧的记忆是单词,那么新的描述也应该只是个单词。

通过这个功能,用户可以自己用母语去替代系统缺省的英文命令。以用中文“定义 整数”学习一遍“define int ”为例,我们知道这里“定义”就是 “define”,“整数”就是“int”,那么我们可以依次执行:

Learn word memory “define”,然后按要求输入“定义”。

Learn word memory “int”,然后按要求输入“整数”。

这样,我们输入“定义 整数 2”也可以执行了。甚至我们可以这样输入:“Define int 2, 定义 整数 3, define 整数 4;”,同样可以正确执行。

注意,当前对中文这种无法靠空格分词的语言,用户必须自己用空格分词输入。

## 回取记忆

由于整个记忆原理设计并没定型,记忆的回取和记忆的存储一样,在将来都有可能发生改变。

就目前设计而言,记忆的回取已经给我们提供了一些惊喜,它能够通过控制记忆的颗粒度来筛选记忆,加快回取。

比如,红色的法拉利和红色的铅笔在大脑中显然记忆的颗粒度不同,如果不考虑这个,只用红色作为关键字搜索,那么可能事无巨细都列出来,但是如果大致设定一个颗粒度,那么先排除这个颗粒度范围之外的记忆,那么很快就能确定我们实际想搜索的目标,这让大脑像一个可调焦距的照相机,快速定位目标信息所在区域,仅此一点,用来改善当前市面上的搜索引擎也会带来巨大价值,我们希望在这个基础上继续前进,当前,仅提供以下基本命令:

```
find "text";
find logic "text";
find object "text";
```

这里 text 是你想搜索的关键词描述，其中用 or 和 not 控制搜索算法，这点和日常搜索引擎的使用差不多，比如：find “aaa bbb,or ddd ccc, not eee fff”；

GMIS 会在对话的 Find 窗口显示符合条件的长期记忆。后两个命令顾名思义分别专门用于搜索逻辑和外部物体。

要控制搜索信息的颗粒度可以在搜索之前使用命令：find pricision xxx；

其中 xxx 等于欲设定的搜索信息大小，它以时间秒为单位，缺省值=5。注意：本次设定后将影响其后所有搜索命令，除非你设定新值。

另外我们还可以设定搜索的记忆的时间范围，它可以进一步提高搜索结果的准确性。当前版本没有实现，主要是工作量问题。

示例：

Find “define”；

## 逻辑重入

逻辑重入是指 GMIS 可以把一个正在执行的任务暂停后，即使关闭系统和电脑，之后也可以在原来执行的基础上继续执行逻辑的能力，期间，任务存储为静态数据，不占用其他系统资源。

这种能力对于某些关键任务来说是必须的，比如某些远程机器需要升级系统或者重启电脑，但是又不想从头开始执行当前任务，那么 GMIS 的逻辑重入能力可以简单的解决这个问题。

从长远看，这也是智能发展所必须的能力。每一个人每天都会思考很多逻辑，其中多数逻辑可能只起了个头，因为各种原因，没有完整执行下去，但人类会把这些半途而废的任务存储为静态记忆，不给大脑带来任何负担，之后既可以在条件具备时继续执行，也可以为以后的任务提供参考。

让我们简单的来实验一下这个能力，首先输入：debug；执行后进入调试模式，然后输入：Define int 2,define int 3,define int 4；

执行后会停留在第二个行为上，此时，我们关闭 GMIS，如果你愿意，还可以关闭点电脑，然后重新启动执行 GMIS，这个时候，你会发现之前那个任务对话还在，点击为焦点后，会看到调试树，你可以接着 step 执行调试，或点击 run，任务会在之前执行的基础上完成。

对 GMIS 来说，逻辑重入实际上也是一种记忆和恢复行为，我们现在只是证明 GMIS 有这种

能力，善用这种能力的还有更多工作可以做。遗憾的是，目前让它具有使用价值还有更多工作需要做，比如，如果使用的是外部物体，那么仅仅在大脑内部恢复执行是不够的。不过对于专用机器人来说，如果所有行为都是本能，这种能力还是可行的。

# 附录

## 已学单词表

这里给出 **GMIS** 原型在第一次启动时缺省已经学会的单词（词性）,仅限英文和对应的中文，用户可以自行教给 **GMIS** 更多单词，或者给现有单词添加其他词性，也可以把本表里的单词翻译成自己母语对应的单词交给 **GMIS** 原型，这样就可以用你的母语编程驱动 **GMIS** 了。

词性	英文单词	中文单词
名词	int	整数
	float	小数
	string	字符串
	operator	操作符
	resistor	电阻
	rs	电阻
	inductor	电感
	id	电感
	capacitor	电容
	cp	电容
	diode	二极管
	dd	二极管
	label	标签
	pipe	管道
	text	文本
	num	数字
	second	秒
	table	数据表
	data	数据
	size	大小
	logic	逻辑
	date	日期
	object	物体
	dialog	对话
	token	符号
	pronoun	代词



	adjective	形容词
	numeral	数词
	verb	动词
	adverb	副词
	article	冠词
	preposition	介词
	conjunction	连词
	interjection	感叹词
	noun	名词
	text	文本
	action	行为
	time	时间
	end	终点
	precision	精度
	info	信息
	document	文档
动词	define	定义
	use	使用
	reference	引用
	wait	等待
	set	设置
	goto	转向
	view	查看
	input	输入
	create	制造
	import	导入
	export	导出
	focus	关注
	insert	插入
	modify	修改
	get	得到
	remove	移除
	close	关闭
	name	名字
	start	启动
	think	思考
	learn	学习
	run	运行
	debug	调试
	stop	停止
	pause	暂停
	step	单步

逻辑关系	find	搜索
	output	输出
	then	然后
	and	同时

# 本能列表

类型	命令	功能	输入\输出	备注
通用命令	Define int xxx; 定义 整数 xxx	把定义的整数 xxx 放入执行管道中	输入：无 输出：整数	命令第二行为中文版，下同
	Define float xxx; 定义 小数 xxx	把定义的浮点数 xxx 放入执行管道中	输入：无 输出：浮点数	须用引号包围小数
	define string xxx	定义一个字符串放入执行管道中	输入：无 输出：字符串	
	use operator xxx 使用 操作符 xxx	xxx 是操作符,可以是以下之一： + - * / % > < ! & ~   ^ >= <= != && == << >>	输入：根据操作符而定 输出：操作后的值	操作符目前与 C 语言的运算符相同
	use Resistor xxx 使用 电阻 xxx	XXX 为整数 n,从执行管道中删除 n 个数据, n 可以大于实际数据个数,表示全删	输入：无 输出：无	
	use inductor xxx 使用 电感 xxx	生成一个名字为 xxx 的电感:当执行管道有数据时,把数据转移到电感,当执行管道没有数据时,电感里的数据则转移到执行管道	输入：无 输出：见功能描述	
	use capacitor xxx 使用 电容 xxx	生成一个名字为 xxx 的电容:当电容为空时,执行管道里数据转移给它,当电容有数据时,把数据转移给执行管道	输入：无 输出：见功能描述	
	use Diode xxx 使用 电感 xxx	模拟二极管,xxx 为指定的整数,当执行管道里的第一个数据等于 xxx 则继续执行,否则当前执行分支失效	输入：整数 输出：无	

reference capacitor xxx 引用 电容 xxx	引用名字为 xxx 的电容	输入：无 输出：见电容功能描述	
reference inductor xxx 引用 电感 xxx	引用名字为 xxx 的电感	输入：无 输出：见电感功能描述	
set label xxx 设置 标签 xxx	设置一个返回标签，xxx 为标签的名字	输入：无 输出：无	
goto label xxx 转向 标签 xxx	当前执行转向标签 xxx	输入：无 输出：无	
view pipe 查看 管道	在运行时窗口打印显示当前执行管道里拥有的数据，不会改变执行管道里的数据	输入：无 输出：无	
input text [xxx] 输入 文本 [xxx]	启动一个子对话，要求用户输入一个字符串，其中 xxx 为可选的对话提示信息	输入：无 输出：字符串	
input num [xxx] 输入 数字 [xxx]	启动一个子对话，要求用户输入一个数字，其中 xxx 为可选的对话提示信息	输入：无 输出：整数或浮点数	
wait second xxx 等待 秒 xxx	让当前对话的执行等待 xxx 秒,xxx 为浮点数	输入：无 输出：无	
create table xxx 制造 数据表 xxx	生成一个名字为 xxx 的数据表	输入：无 输出：无	
focus table xxx focus table 关注 数据表 xxx	设置当前焦点数据表，如果 xxx 不是静态给出，就从执行管道中取第一个字符串为数据表名字	输入：见功能描述 输出：无	
import data 导入 数据	把当前执行管道的数据导入到焦点数据表中	输入：无 输出：执行管道为空	
export data 导出 数据	把当前焦点数据表中的数据导入到执行管道中	输入：无 输出：见功能描述	
insert data 插入 数据	把当前执行管道里的数据插入到当前焦点数据表中，其中执行管道离得第一个数据必须是整数，指向插入的 index	输入：见功能描述 输出：执行管道为空	
modify data 修改 数据	用当前执行管道里的数据修改当前焦点数据表里指定的某行，其中执行管道里的第一个数据必须是整	输入：见功能描述 输出：执行管道为空	

		数，指向修改行的 index		
get data 得到 数据		把当前焦点数据表里指定行数的数据放入到执行管道中，行数由执行管道里的第一个数据指出，必须是整数	输入：见功能描述 输出：见功能描述	
remove data 移除 数据		在当前焦点数据表里删除指定行的数据，行数由执行管道里的第一个数据指出，必须是整数	输入：见功能描述 输出：无	
get size 得到 大小		返回当前焦点数据表的行数，放入当执行管道中	输入：无 输出：整数	
close table 关闭 数据表		关闭当前焦点数据表	输入：无 输出：无	
use logic xxx 使用 逻辑 xxx		使用名字为 xxx 的临时逻辑	输入：无 输出：无	
focus logic xxx focus logic 关注 逻辑 xxx		设置当前焦点逻辑，其中 xxx 为指定逻辑名字，如果没有静态给出，则动态从执行管道中取出第一个数据作为名字，必须是字符串类型	输入：见功能描述 输出：无	
insert logic xxx 插入 逻辑 xxx		在当前焦点逻辑实例中插入名字为 xxx 的临时逻辑	输入：无 输出：无	
remove logic 移除 逻辑		在当前焦点逻辑实例中删除本命令所在的逻辑分支	输入：无 输出：无	
get date 得到 日期		得到当前日期	输入：无 输出：字符串	
get time 得到 时间		得到当前时间	输入：无 输出：字符串	
output info 输出 信息		从执行管道里取出第一个数据打印显示到运行时窗口，第一个数据必须是字符串类型	输入：字符串 输出：无	
start object xxx 启动 物体 xxx		启动一个外部物体实例，并设置为当前焦点物体，xxx 为外部物体的名字	输入：无 输出：无	
name object xxx name object 命名 物体 xxx		给当前焦点物体命名，如果不能静态给出的物体实例名字 xxx，则从执行管道中取出第一个数据作为物体实例的名字，必须是字符串类型	输入：见功能描述 输出：无	
focus object xxx		设置焦点物体，如果不是	输入：见功能描	

	focus object 关注 物体 xxx	静态给出焦点物体的实例名字，则从执行管道中取出第一个数据作为焦点物体的名字，必须是字符串类型	述 输出：无	
	use object 使用 物体	使用当前焦点物体	输入：见此物体 使用文档 输出：见此物体 使用文档	
	close object 关闭 物体	关闭当前焦点物体实例	输入：无 输出：无	
	Get object document 得到 物体 文档	得到当前焦点物体的使用文档	输入：无 输出：字符串	
内部独立命令	think logic xxx 思考 逻辑 xxx	使大脑进入思考模式，xxx为欲思考逻辑的名字	输入：无 输出：无	
	Run 运行	继续执行当前逻辑任务	输入：无 输出：无	
	Debug 调试	使当前大脑进入调试执行模式	输入：无 输出：无	
	Stop 停止	停止执行当前逻辑任务	输入：无 输出：无	
	Pause 暂停	暂停执行当前逻辑任务	输入：无 输出：无	
	Step 单步	单步执行当前逻辑任务	输入：无 输出：无	
	close dialog 关闭 对话	关闭当前对话	输入：无 输出：无	
内部非独立命令	learn token xxx 学习 符号 xxx	记忆 xxx 为 token 词性	输入：无 输出：无	
	learn pronoun xxx 学习 代词 xxx	记忆 xxx 为 pronoun 词性	输入：无 输出：无	
	learn adjective xxx 学习 形容词 xxx	记忆 xxx 为 adjective 词性	输入：无 输出：无	
	learn numeral xxx 学习 数词 xxx	记忆 xxx 为 numeral 词性	输入：无 输出：无	
	learn verb xxx 学习 动词 xxx	记忆 xxx 为 verb 词性	输入：无 输出：无	
	learn adverb xxx 学习 副词 xxx	记忆 xxx 为 adverb 词性	输入：无 输出：无	
	learn article xxx 学习 冠词 xxx	记忆 xxx 为 article 词性	输入：无 输出：无	
	learn preposition xxx 学习 介词 xxx	记忆 xxx 为 preposition 词性	输入：无 输出：无	
	learn conjunction xxx	记忆 xxx 为 conjunction 词	输入：无	

	学习 连词 xxx	性	输出：无	
	learn interjection xxx 学习 感叹词 xxx	记忆 xxx 为 interjection 词性	输入：无 输出：无	
	learn noun xxx 学习 名词 xxx	记忆 xxx 为 noun 词性	输入：无 输出：无	
	learn text xxx; learn text; 学习 文本	记忆一段文本 xxx, 如果没有静态给出文本, 则会交互式的要求用户输入文本	输入：无 输出：无	
	learn logic xxx 学习 逻辑	记忆名字为 xxx 的临时逻辑	输入：无 输出：无	
	learn object xxx 学习 物体 xxx	记忆名字为 xxx 的外部物体	输入：无 输出：无	
	learn action xxx 学习 行为 xxx	把临时逻辑 xxx 记忆为一个行为, 会交互式的要求用户输入行为短语和使用说明	输入：无 输出：无	行为短语所需单词必须提前记忆
	Find xxx 搜索 xxx	在记忆中搜索关键字 xxx	输入：无 输出：无	
	find logic xxx 搜索 逻辑 xxx	在记忆中的逻辑中搜索关键字 xxx	输入：无 输出：无	
	find object xxx 搜索 物体 xxx	在记忆的物体中搜索关键字 xxx	输入：无 输出：无	
	set memory size 设置 记忆 大小	设置搜索的记忆大小	输入：无 输出：无	