《Paracraft编程入门》 精简预览版

尊敬的读者：

我是本书的作者李西峙。这份电子文档**没有**包含本书第一部分的项目正文。如果您希望购买纸质完整版，可以联系我们商务人员。

李西峙
lixizhi@paraengine.com

前言

编程一直被误解为一件很难的事情。一个原因是几乎所有编程语言（工具，文档，开发者社区）都是英语文化圈下的产物，如果你英语不好，就无法真正融入其中。另一个原因是编程语言没有被教育者真正当成一种人类的语言去对待。

本书希望能够作为编程的入门教材，将正确的工具，学习方法介绍给希望真正掌握编程的你。本书的目标是通过我们原创的Paracraft工具，让你可以随心所欲的创作出任意复杂的3D动画与游戏。当你具备这种入门能力时，你仍然可以继续使用Paracraft开发专业的计算机软件或自学任何其它语言。

学习编程和学习外语很像，需要大量的**打字练习**。动手打字就如同学习外语中的发音一样重要。回忆一下，从出生开始，我们就在学习母语的发音，然后我们每天还要去使用它，长大后又系统的学习它。一个4岁的小孩已经能够用母语的发音表达自己的任何想法。相似的，本书希望营造一个类似的学习环境，让你可以在计算机世界中表达自己的任何想法。这也是当代希望从事科学与创造性工作的人的一项必备技能。拥有编程的入门能力并不困难，但也需要4年的时间或打5000行以上的代码。

1989年，7岁的我照着一本我父亲给我的书编写了我人生中第一个程序，并从此喜欢上了编程，小学期间我完成了大量个人作品，到了随心所欲的入门状态，这要感谢我的老师。本书也希望和大家分享我的学习经历。

李西峙
2018.2.5
创作于深圳大富配天集团NPL语言研发中心
lixizhi@paraengine.com
https://keepwork.com/

作者简介

李西峙：1982年出生于哈尔滨。2005年毕业于浙江大学计算机系（竺可桢学院）。大学期间他在国内外会议和刊物上发表游戏引擎，脚本语言技术，三维动画制作，CPU芯片设计相关论文和著作8篇。2004年出于长期对人工智能的兴趣和研究，他创立了NPL语言（Neural Parallel Language）和ParaEngine 3D游戏引擎，至今写了过百万的开源代码和各种开发工具。Paracraft也是基于NPL语言开发的一款3D创作工具。2007-2015年，他先后获得著名风险投资IDG和国内著名企业家的投资，并任CEO。

李铁才：1950年出生于上海，是我的父亲，《相似性和相似原理》作者。哈尔滨工业大学及深圳研究生院双基地教授，博士生导师，深圳航天科技创新研究院科技委主任。1996年获航天突出贡献专家称号；1997年获国家特殊津贴；获国家、省、部级科技成果奖18次；获中国发明专利30项，获美国发明专利3项；在国内外发表论文50余

篇。历时30年致力于相似性原理及其应用技术的研究。编写了本书第5章 相似原理与人脑仿真 。

> 创作本书时还有一个备选副标题：相似原理入门教程。**相似性和相似原理**是驱动人类大脑的基本工作原理，Paracraft也可以看成是使用相似原理进行创作的一个工具。我们会尽量将相似原理的思想融入到本书中，让你在学会编程的同时对人脑和宇宙万物的工作方式有一定的认知。程序员是虚拟世界的造物主，而虚拟世界与现实世界和人类大脑必然有相似之处。

本书的一些章节是由我的团队合作完成的，他们是：

- **刘远亮**：电梯调度算法，3D世界的编程模型，编程中的建模过程：乒乓球小游戏，编程中的建模过程：迷宫小游戏，编程中的建模过程：钢琴，编程中的建模过程：飞行的小鸟，编程中的建模过程：坦克大战，编程中的建模过程：跳一跳，由电梯调度算法了解编程思维，生命游戏，复杂编程中的建模过程：生命游戏的制作，BlockBot小游戏介绍，复杂编程中的建模过程：BlockBot核心部分的制作，BlockBot小游戏－3D UI，BlockBot小游戏－2D UI，BlockBot小游戏－复杂UI设计，人力资源游戏，如何学习Paracraft编程。
- **张大伟**：复杂编程中的建模过程：BlockBot核心部分的制作，BlockBot小游戏－3D UI，BlockBot小游戏－2D UI，BlockBot小游戏－复杂UI设计。
- **陈清华**：计算机体系结构，台式计算机模拟，狗狗陪护机器人模拟。
- **张磊**：CAD建模“桌子”，CAD建模“空心的盒子”，CAD建模“奥运五环”，CAD建模“杯子”，CAD建模“掷骰子游戏”。
- **罗阳平**：制作个人网站，创建课程包。
- **谭雯文**：创建方块，批量操作，电影方块，演员和动画，夏天游泳，制作简易动画开头，BMAX模型，简易BMAX小吉他，简易BMAX小动画，四足动物与镜头震动效果，密室教学，密室开头设计，星球运动仿真教学，BMAX简易骨骼与X文件应用，传送石，电影地震镜头与人物表现，电影方块与过山车，子母电影方块。

如何学习编程

在深圳大富配天集团NPL语言研发中心，我们每周六都有一个3小时的免费的Paracraft编程导引课，参加的人主要是小学生，偶尔还有家长和大学老师，总共已经30多次了。

来的大人们经常会问我：“应该如何学习编程？”；而小孩们总是迫不及待的运行电脑上的Paracraft，自己探索起来。

学习编程很简单。首先，你需要买一台带键盘的PC机或笔记本电脑。编程是需要打字的，因此需要鼠标和键盘。遗憾的是我发现50%的普通家庭只有手机，PAD或者15年前

无法开机的电脑。如果你已经有最新的笔记本电脑，可以再购买一个更大的显示器和独立的鼠标键盘，就可以将笔记本改造成台式机。

其次，你要学会在电脑上安装一个能编程的软件，本书的读者需要安装Paracraft（后面会教你）。

最后，你要允许自己的小孩儿使用电脑，哪怕是玩游戏。99%的家庭之前都做不到这三点，因此虽然大多数学校都开设了编程课，孩子也不可能学会编程。当你在家中做到这3点时，你会发现很多孩子是天生的程序员。孩子们会逐渐远离手机，更愿意将时间花在大屏幕上，去探索别人的作品或自我创作。这就好比孩子都喜欢模仿大人的语言和行为，当你给它一大堆积木，他们会先破坏再搭建。

如果你的孩子已经是一个手机游戏高手，那么不妨让他到电脑上玩Paracraft。Paracraft中有大量其它孩子，老师，程序员创建的3D游戏和动画作品，每个作品都可以随时查看背后的逻辑与代码，别人的作品就是最好的老师。

学习计算机语言和学习其它自然语言，如中文和英文，是一样的，你要不停的使用它，创造出自己的作品。

其实人类学习任何技能都是一样的，因为教育的本质就是让人保持思考和一直有事可做。

因此，我们还为Paracraft开发了一个学习平台，叫做KeepWork，官网是：

```
https://keepwork.com
```

KeepWork有2个字面意思：

- 保持(keep)有事可做(work)：人不能放弃工作和创作，大人小孩都一样。这个是教育的本质。
- 保存(keep)作品(work)：我们保存了你的所有作品和更改历史。作品是未来教育的重要评估方式，不再需要考试的分数。

只要你安装了Paracraft，后面的一切就可以教给孩子自己去探索、学习和创造了。而你只需要观察孩子是否一直保持有事可做即可。本书其实是Keepwork上内容的一个浓缩，启发孩子去探索和创作超过50个项目。出书的原因是为了让成年人也能快速和系统的了解Paracraft和编程理论，方便教育机构的教学，以及保护孩子的视力。但是真正入门编程，仍然需要大量在计算机上的打字练习和项目实践。

本书有什么

本书分为上，中，下三部：

- 上部：50个编程项目
  - 通过思维实验解决问题，理解AI和体验编程
- 中部：编程理论
  - 系统的讲解变量，函数等重要编程概念
- 下部：参考手册
  - 本书中所有的编程词汇，都可以查询

这三部分内容分别代表了程序员的3类重要行为：搜索项目、学习理论和寻找文档。下面来说说这3类行为:

- i. 搜索项目：程序员在解决一个问题时，通常会先去搜索别人的开源项目来参考。新人学习编程也是从项目开始的，而不是理论或语法。因此上部我们将大量的编程项目按照一定的次序归类到了几个章节中。你可以根据自己的能力和喜好从任何一个开始做起。这些项目大多来自我们过去每周六的编程导引课，因此都可以在2个小时内完成。
- ii. 学习理论：计算机程序已经深入到了人类科技的每个角落：物理，化学，生物，航天，材料，数学，动画，游戏等等。每个领域的程序员都需要学习对应的理论知识，然后用程序创造出符合相似规律的虚拟事物来。在本书的中部，我们会系统的讲解计算机编程的通用理论，这些理论和电子计算机的工作原理相关，因此几乎适用于所有现代计算机语言。而关于通用理论之外的理论，我们放在第一部分的每个项目中讲解。
- iii. 寻找文档：一般计算机语言的全部内置语法和词汇只有不到20个。因此对于已经精通一门语言的程序员，学习一个新的计算机语言大概只需1小时。但是编程是一个不断创造新词汇（函数）的过程，我们在编程时需要去使用其它程序员创造的词汇，这个词汇的数量从几百到上千，甚至几十万，每个领域都有自己的词汇，每个项目也有自己的词汇。程序员会给这些新词汇写使用说明书，我们叫做文档，由于文档数量太多，即使是资深程序员也只能靠模糊查询，而不是精确记忆。我们每写一行代码都可能需要先查看文档。在本书的下部中，我们列出了在上部的项目中使用到的所有词汇的文档。这些词汇已经足够开发任何你能想到的动画和游戏。

如何使用本书

对于自学者，我们希望用户从本书上部的任何一个项目开始学习编程。然后遇到任何不懂的地方可以在中部和下部中找答案。但是请你不要去记忆任何下部中的内容，只要有一个模糊的印象即可。即使是像我这样有30年编程经验的程序员，也经常在这3种行为之间切换，只不过专业程序员不会看书，而是从互联网中通过搜索去获取这些

内容。因此初学者可以先使用本书，然后逐渐脱离本书，直接到Keepwork网站或Paracraft软件中找答案。

本书也可作为编程教材，上部的每个项目都可以作为一个2小时的编程课。我们为每个项目制作了在线课程，目录中的项目ID为在线课程ID，方便老师在课堂教学，详见第一章的开始部分。老师可以根据课时数，按照任意自己喜欢的顺序和学生的能力选择课程项目，我们的网站上给出了几种参考性的教学计划。

本书的中部（理论部分）不建议在课堂上教学，但是老师可以安排一些答疑课，或者每节课留一些时间给学生提问。计算机语言中需要解释的知识点不到20个，更多的是需要大量实践。课堂上的教学：`应以完成项目制作为目标，而非彻底理解里面的所有细节`，鼓励学生间相互帮助。而贯穿整个学期的教学的大目标应该是`引导学生用课余时间完成自己的计算机项目`。我们建议老师在每节课开始或结束的时候，用5分钟的时间请班级中的学生展示一下自己在课余时间完成的作品，如果还没有这样的作品，可以使用我们网站上的学生作品视频。我们的网站上每月都举办编程大赛，里面有大量学生作品。

当学生开始创作自己的作品时，有能力的老师应该定期去阅读学生的作品代码，并给出修改意见。这就如同语文老师要批改学生的作文一样。这对老师的编码水平有一定的要求，但是如果学生将作品分享到KeepWork， 其他高水平的用户就可以提供这种帮助或服务。

在第6章`对未来教育的思考`中，我们给出了如何以本书为起点，构建从小学1年级到大学的编程学习体系。本书的`附录4`具体的讲解了如何开展Paracraft编程教学。

谁该阅读本书

具有阅读能力的人都可以阅读本书，本书争取只使用不超过20个专业术语（见附录3）去讲解计算机编程。 在我小学时，经常读一些大人门的科普书籍，总能有读懂的部分，也鼓励我去搞清楚看不懂的部分。哪怕是最高深的理论都应该能在生活中找到相似的比喻，如果找不到，很可能是这个理论本身有问题。

我心目中的读者包括：

- **喜欢编程的小朋友**：尽力去理解本书，但你不需要全部看懂，甚至开始你只需将本书看成50个编程实验的绘本，但需要你亲自动手去体验代码和计算机的创造力。
- **家长**：我理解大人们学习编程已经没有小孩儿那样容易了，因为你可能没有时间和兴趣去动手写代码，但是至少你可以快速的阅读本书，回答一些小孩儿提出的问题。

- **老师**：我鼓励所有年轻教师，无论你是教什么学科的，都能像小朋友一样去动手学习计算机编程，达到可以随心所欲的入门程度。当你掌握了Paracraft，你就打开了一扇门，你能创造的将不只是PPT（文字和图片），而是动画，逻辑与交互。未来教育需要大量教育工作者的创造；项目式和交互式学习将取代45分钟的课堂学习。在我们的官网（keepwork.com)我们为老师提供了创建项目式课程学习的编辑工具，本书中所有的项目已经被其它老师做成了交互式在线课程。我们期待更多，各个领域的教育工作者加入我们，用计算机技术改变未来的教育。我们为教育工作者提供培训。
- **程序员**：Paracraft是一个开源的软件生态，我们期待你的加盟，一起开发面向人工智能和未来教育的新应用。有经验的程序员也许也能从本书中获得一些编程方面的启发。如果你既是程序员又是一位家长，Paracraft是你最好的亲子读物了，你应该和孩子一起完成一个个项目并通过我们的官网分享出来。
- **大学生和大学老师**：Paracraft在过去几年中有很多作品，课程，CAD工业设计，书籍等都是由大学老师或大学生完成的，希望更多的大学生和老师选择Paracraft作为毕业设计的工具或题目。我们的官网还提供了人人可为老师的教学机制，希望大学生可以教小学生，并一同完成项目和学习编程：当别人的老师是最好的学习方式。

站在前人的肩膀上

Paracraft是使用NPL语言开发的。从2004年创立NPL计算机语言，到2005年制作ParaEngine分布式游戏引擎，到2007年发布儿童动漫创作平台，到2009年发行魔法哈奇3D创想乐园，到2012年发布Paracraft创作工具，到2015年NPL语言开源，到2018年发布Keepwork。期间我吸收了很多前人的思想和成果。

我无法罗列全部，但是最重要的思想是我父亲的《相似性和相似原理》，初稿是写于1982年，2015年后我也加入了该书的修改和补充工作。其中包含了对多组时空序列及其相似性的数学表达与大量研究案例。宇宙内部的相似性从易经开始，到亚里士多德，到后来，它已经被无数科学家研究过。可能是它太普遍，导致我们在使用它时，忽略了它的存在。在人工智能时代，我们有必要将它系统的作为一门独立的理论去研究。人类的大脑由记忆与单向连接构成。记忆就是时间序列，或者说是动画，我们很难去修改自己的某个记忆，但是我们可主观的选择一段记忆的时间起点在大脑中播放。

NPL语言的基础语法是基于Lua的，后来又受到Lisp语言的影响，使得它支持动态语法扩展。Lua的语法是全世界公认最简洁的。它被无数高级脚本语言采用作为基础语法；同时它拥有全世界最快的动态编译器Luajit，使得我们可以用C/C++去扩充它。

在本书中，第一步：我们要教会你如何随心所欲的创建任意复杂的三维时空序列，也就是动画。我们的网上有成千上万的小朋友自己创建的Paracraft动画片或电影供你学习和参考。第二步：我们要教会你如何用代码去控制这些动画的播放起点，你就像一个导演或音乐指挥家一样让你的动画在你代码的指挥下播放。当你可以随心所欲的掌握这2个技能时，你发现你已经可以像控制自己的思想和梦境一样去控制数字世界中的一切。

2007年和2009年当我用这种思想创建儿童动漫创作平台（KidsMovieCreator）和魔法哈奇时，还不知道后来风靡全球的一款游戏叫Minecraft。当我2012年深入研究它时，才发现它的魅力。但我是从相似原理的角度来看它的，在时空序列的数学表达中，世界应该是粒子化的，而且应该是其大无内，其小无外的。Paracraft将这种粒子化建模的思想发挥到了极致，用于任意的动画创作和编程。

MIT实验室的Scratch对少儿编程的贡献是巨大的，它让更低年龄层的用户可以自学编程。Paracraft后来也引入了类似Scratch的积木式编程，并且我们让它可以控制3D世界中的角色，并可以平滑的过渡到基于文本的编程。我们还提出并实现了一种面向记忆的编程模式。

无法罗列全部前人的成果，所以我们从最开始就在Github上开源了我们的所有成果，包括NPL语言和Paracraft。至今有上百位开发者参与其中，我们希望更多的程序员和教育工作者可以在我们的研究成果上继续探索。

致谢

感谢家人
感谢老师
感谢Paracraft的用户，尤其是奇仔，桃子，无心，阿杰。是你们的辛勤付出让我们的工具可以大放异彩。
感谢魔法哈奇超过500万注册用户近10年来的陪伴，很多用户从小学升到了大学，仍然没有完全离开这款3D社区和Paracraft。
感谢我过去和现在的团队，未来我们还有很长的道路要走。很多中途离开的同事依然在远程参与我们的开源社区。
感谢合作伙伴与多位教育工作者愿意在我们还不完善时，坚持使用我们的产品。
感谢资方，在最艰难的时刻依然给我们无私的赞助，能够让我们去实现一个10年以上的长期规划。

目录

- 项目20x134：子母电影方块

1.4 如何赋予虚拟人物智能？

- 项目6x18：代码方块教学1
- 项目6x19：代码方块教学2
- 项目6x20：乒乓球小游戏
- 项目6x22：迷宫小游戏
- 项目28x112：飞行的小鸟
- 项目17x74：坦克大战
- 项目20x79：星球运动仿真教学
- 项目6x24：双重机关与事件
- 项目6x25：制作图形界面
- 项目6x26：代码方块的输出
- 项目6x21：钢琴
- 项目17x120：制作钟表
- 项目35x130：CAD建模“掷骰子游戏”
- 项目28x112：跳一跳
- 项目28x107：电梯调度算法
- 项目28x108：由电梯调度算法了解编程思维
- 项目28x110：3D世界的编程模型
- 项目28x112：编程中的建模过程：乒乓球小游戏
- 项目28x114：编程中的建模过程：迷宫小游戏
- 项目28x115：编程中的建模过程：钢琴
- 项目28x125：编程中的建模过程：飞行的小鸟
- 项目28x126：编程中的建模过程：坦克大战
- 项目28x116：编程中的建模过程：跳一跳
- 项目34x123：生命游戏
- 项目34x124：复杂编程中的建模过程：生命游戏的制作
- 项目24x84：BlockBot小游戏介绍
- 项目24x95：复杂编程中的建模过程：BlockBot核心部分的制作
- 项目24x93：BlockBot小游戏 - 3D UI
- 项目24x94：BlockBot小游戏 - 2D UI
- 项目24x96：BlockBot小游戏 - 复杂UI设计
- 项目36x136：台式计算机模拟
- 项目36x135：狗狗陪护机器人模拟
- 项目33x122：人力资源游戏

1.5 保存并分享你的作品

- 项目29x118：制作个人网站
- 项目23x83：创建课程

中部：编程理论

2 基础编程理论

- 2.1 编程基本概念与语法
- 2.2 程序的本质
- 2.3 数字与数学
- 2.4 变量与名字
- 2.5 字符串与文字
- 2.6 表与数组
- 2.7.1 函数
- 2.7.2 内置函数
- 2.8 总结与对自学编程的建议

3 计算机辅助设计CAD简介
4 计算机体系结构
5 相似原理与人脑仿真
6 对未来教育的思考

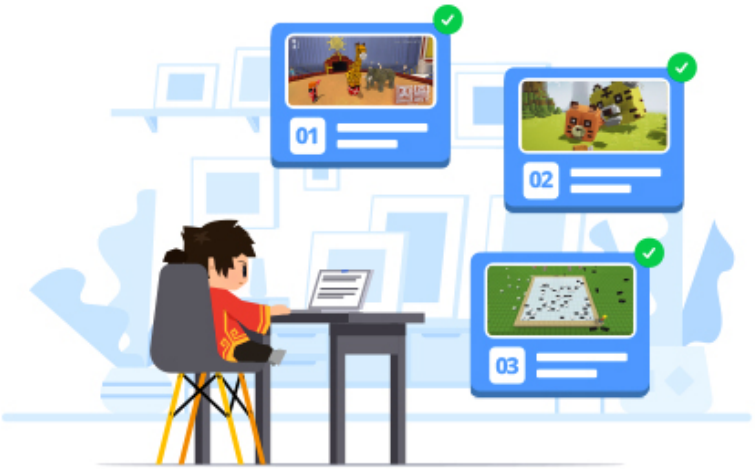
下部：参考手册

- 附录1：NPL常用语法 速查表
- 附录2：代码方块 函数速查表
- 附录3：术语表
- 附录4：如何学习Paracraft编程
- 附录5：推荐书目

上部：项目

> “编程要从玩开始，你需要体验足够多的项目”

上部:项目



编程要从玩开始，你要体验足够多的项目

1 编程项目

学习编程就是学习如何用计算机去创建和现实世界相似的虚拟世界。
本书中的项目将引导你动手去创造3类虚拟事物，分别是：

- 几何相似的3D世界：建造静态3D世界
- 随时间运动的角色：动画与电影
- 让角色拥有智能：程序与逻辑

本书的项目按照上面的顺序分成了3大类。但是你完全可以先将所有的项目都浏览一遍，然后跳跃着学习。

> 学习编程，第一步应该让小孩（包括大人）学会玩游戏，在游戏中东看看和西看看。看到好看的东西，看到好玩的东西，并且告诉他：这些都是小朋友自己创作和分享给大家的。

什么是Paracraft？

Paracraft(创意空间)是一款免费开源的3D动画与编程创作软件。你可以用它创建3D场景和人物，制作动画和电影，学习和编写计算机程序。与成千上万的用户一起学习和分享你的个人作品。本书中所有的动画与编程项目都使用Paracraft制作完成。

> 动画 + 编程 = Paracraft

Paracraft使用NPL语言开发完成。NPL语言全称Neural Parallel Language(神经元并行计算机语言)是本书作者于2004年为了解决基于相似原理的AI仿真问题研发的一个编程语言，语法与主流编程语言兼容，NPL社区通过github开源了200多万行引擎与NPL类库代码，期待编程爱好者的加入。

> NPL语言官网：https://github.com/LiXizhi/NPLRuntime

Paracraft模拟了人类大脑的工作方式。人脑具有下面几个核心能力：

- i. 对3D世界的抽象建模能力：我们生活在3D的世界中，而人脑天生对3D世界具有抽象建模能力。最近的研究发现，人脑中存在大量相似的神经元细胞具有和3D几何世界对应的空间关系。
- ii. 对动画的记忆能力：人脑的记忆不是静止的，而是随时间变化的动画片段。这些记忆一但形成，很难被修改。如果将一个人从出生到20岁看到和听到的一切用手机录制下来，大概需要6000GB，约等于15部512GB的手机容量，对计算机来说，并不是很多。
- iii. 对记忆的控制能力：人类的语言与行为其实是对过去记忆的重新剪接与播放。你的大脑仿佛是一个电影导演，指挥着很多动画记忆片段的播放。而驱动记忆重放的主要原则是相似性。

本书通过Paracraft软件，让你学会控制计算机去做类似的这3类事情。在Paracraft中：

- i. 我们主要用方块构建3D几何世界：人脑中的信号单元也是粒子化的，比如视网膜上有650-700万个视锥细胞可感受颜色和强光；而我们用手机拍摄的照片则由大概2000万个方块点构成。科学家观测到，在人脑深处，当我们从不同角度观察一个熟悉的环境时，某些神经元细胞构成的点阵，也会以相似的模式被依次激活；仿佛在我们脑海深处也有一个相似的由点阵组成的立体世界。这也许可以解释为何孩子对乐高积木特别喜欢，因为这种建模方式与人脑相似。
- ii. 我们用电影方块记录动画：本书中有相当的篇幅和项目是教你如何制作和播放3D动画片段。3D动画（也包括图片，声音）其实是编程的主要素材。我们在各类软件中看到的一切可操作的图形界面，或者游戏中会动的人物都需要先制作成可被计算机调用的动画素材。3D动画就如同我们的记忆一样。没有海量的记忆，人类无法思考；没有大量的动画素材，程序无法呈现。
- iii. 我们用代码方块控制动画：编程可以看成用逻辑去控制动画的过程。人类的思维也可以看成是通过相似匹配去控制记忆播放的过程。只不过人类还没有搞清楚这个相似匹配的全部规律。但是我们可以用人类语言去描述输入和输出的关系，从而控制在什么情况下，动画从哪里播放，到哪里结束。这种描述方法就是我们要学习的编程。在Paracraft中，我们提出一种简单易学的`面向动画`的编程方式。

总结：Paracraft致力于提供一个面向个人的3D动画与编程创作环境。我们探索了一种类似人脑的建模方式。无论你是小学生还是成年人，通过学习Paracraft，你可以随心所欲的创建3D动画，游戏以及专业计算机软件，并可以独立发布软件到Windows/MAC/Android/iOS等众多平台。

如何学习本书中的项目

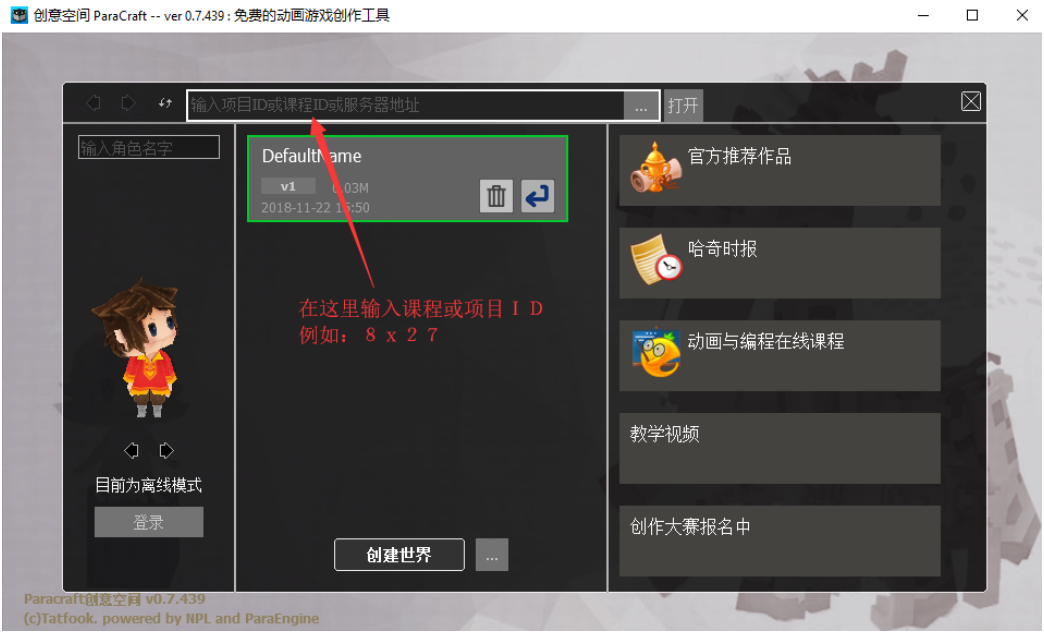
> 本书中的每个项目都对应了Paracraft中的一个3D世界。在Paracraft中输入项目课程ID例如`8x27`就可以进入项目对应的3D世界了。

你可以在Paracraft中将所有项目的3D世界都浏览一遍；选择你喜欢的项目，开始编程之旅。

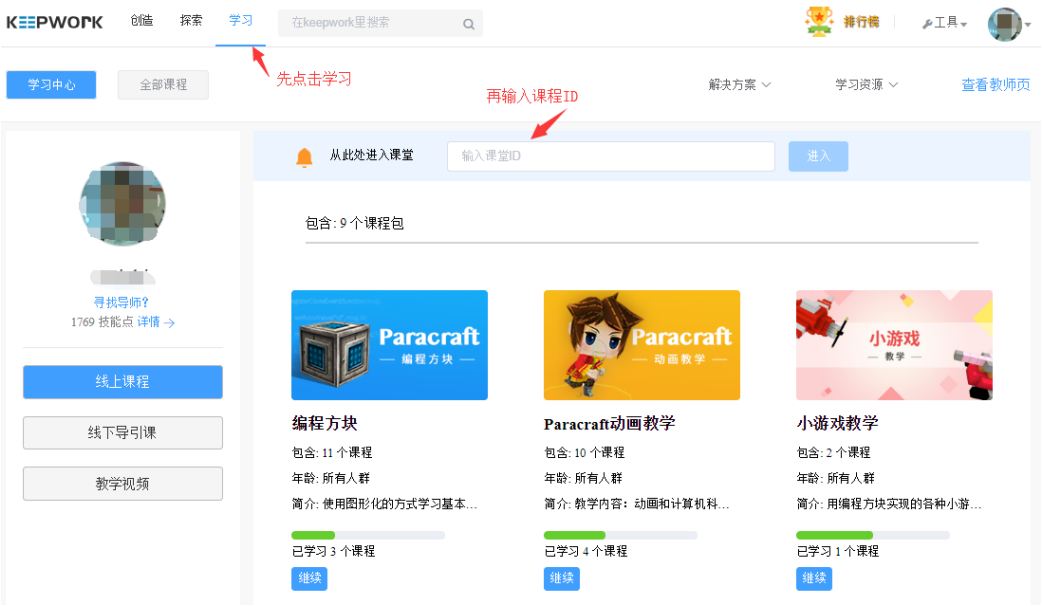
本书的第一部分包含了50个精选编程项目，它们都不需要你之前具有任何编程知识就可以完成。本书的每个项目给出了项目的文字描述和实现它的核心源代码。更多的项目指引可以通过在Paracraft或KeepWork网站中输入**课程ID**来查看。

每个项目的**课程ID**在每个小节的第一行以及目录的标题中，例如：第一个项目是`项目8x27：安装Paracraft和编辑模式`，则课程ID为`8x27`。

可以在Paracraft软件中输入这个ID，如图：

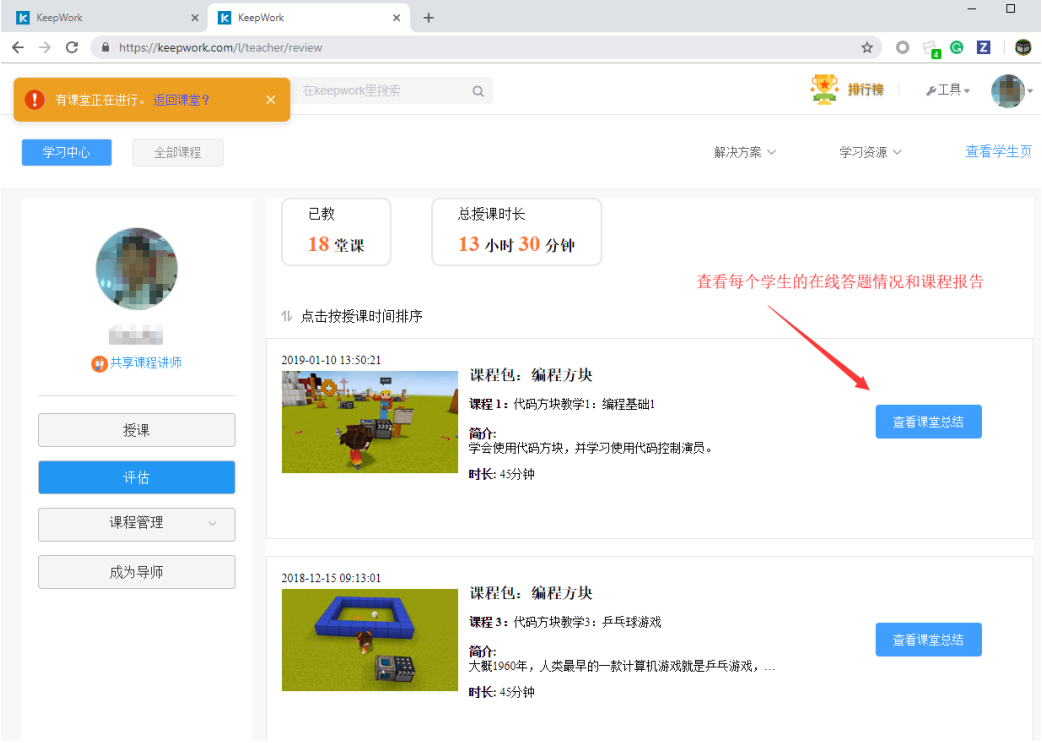


当然你也可以在KeepWork网站中输入这个ID，请先访问：https://keepwork.com 然后点击上方的`学习`，然后就可以输入课程ID了，如图：



所有的项目课程可以通过`学生自学`或`老师教学`模式完成。如果你希望在教室中授课，请点上图中右侧的`查看教师页`，切换到教师身份。教师身份下，你可以使用我们免费的在线编辑器创建课程。我们也鼓励老师和学生为自己的3D作品制作课程包，然后分享出来，供他人学习。详见`项目23x83：创建课程包`。

教师模式下的**课程包**还提供了在计算机教室中上课的各种辅助工具：包括备课，教师讲义，学生答题情况实时汇总，课堂总结，局域网内学生电脑桌面监控等，最重要的是它会自动安装课程所需的课件，3D世界，作业问题到每个学生的电脑上，让学生能迅速进入学习和创作状态。下图为教师页首页，展示了所有教过的课程。



好了，一切准备就绪，第一个项目我们会教你如何安装Paracraft，剩下的其它项目你可以自由选择。记住：学习编程是从玩游戏和写代码开始的。当你能够随心所欲的使用Paracraft创建任何你想做的动画和游戏时，你就入门了。对于“零”基础的人，这个过程可能需要几年的时间：需要你欣赏上百个别人做的Paracraft动画或游戏，动手完成数十个项目，写上5000行代码。不过这将是一个十分有趣和富有成就感的旅程。

项目8x27：安装Paracraft和编辑模式

课程ID：8x27
简介：了解Paracraft动画和游戏制作工具。观看由经验丰富的用户制作的动画短片。学习如何在3D世界中移动，播放动画，阅读它的源代码。

1 理论

今天，我们将观看一小段动画视频叫做：`What Do You Do With An Idea? 有了想法你怎么做？`

这个视频由Paracraft制作。Paracraft允许你能利用方块创建高级的3D动画和游戏。如果你已经玩过minecraft或者乐高，你将会喜欢它，但是Paracraft是一个更强大的3D动画和编程工具，并且是免费和开源的。你将要看到的动画是由像你这样的人仅仅使用Paracraft制作的。我们将学习阅读它的源代码来完成这一课。

2 游戏

学习如何从官方网站安装软件和工具是一项关键技能。许多人没有成功的学会编程是因为不知道怎样用家里的电脑下载和安装最新的软件。

首先你需要到官方网站安装Paracraft。

http://paracraft.keepwork.com/download

在浏览器中打开上面链接，然后点击下载，你需要根据你的操作系统安装相应的版本。

编程需要打字，我们强力建议你在最新的PC操作系统上安装`Windows版`。点击图中的下载按钮，下载完成后，你需要运行下载的`paracraft_full.exe`文件。

根据指引完成软件的安装，如果你的电脑出现安全警告或提示，请允许程序运行。

安装完成后，在你的桌面会生成一个图标 

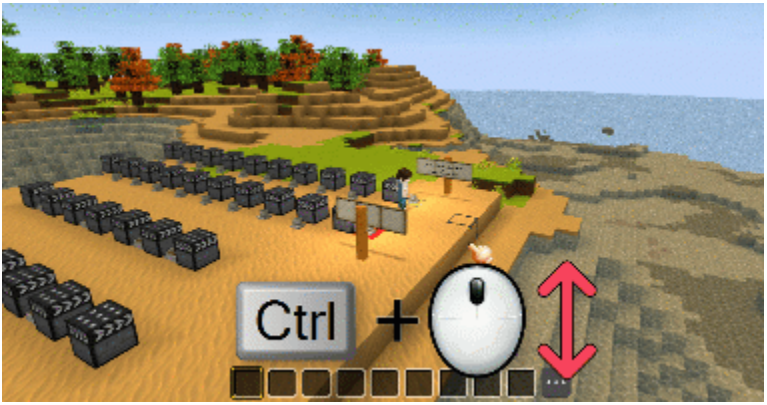
，点击它并完成软件的更新，就可以启动Paracraft了。

> 每次启动时，我们强烈建议您更新到最新的版本。

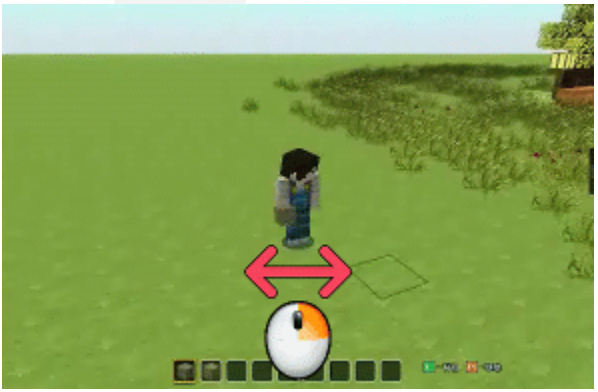
上面是软件的启动界面， Paracraft软件的版本号在窗口左上方和左下方都有显示。

- 学习用 `W A S D` 键来移动。

- 按住 `Ctrl` 键并滚动鼠标滚轮来放大和缩小视角。

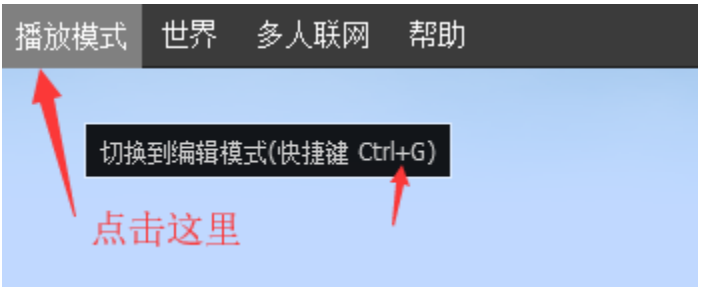


- 学习按住 `鼠标右键` 并拖动来改变视角。



当你打开一个用户创建的世界时，比如 `有了想法你怎么做？` ，你可以在游戏和编辑模式之间切换。在游戏模式中，你只能局限于创造者设定的游戏规则。在编辑模式下，你可以修改这个世界，并读取特殊方块中的源代码。

- 按 `Ctrl+G` 键切换到编辑模式，或者按下 `Esc` 键，在左上角点击播放模式切换模式。



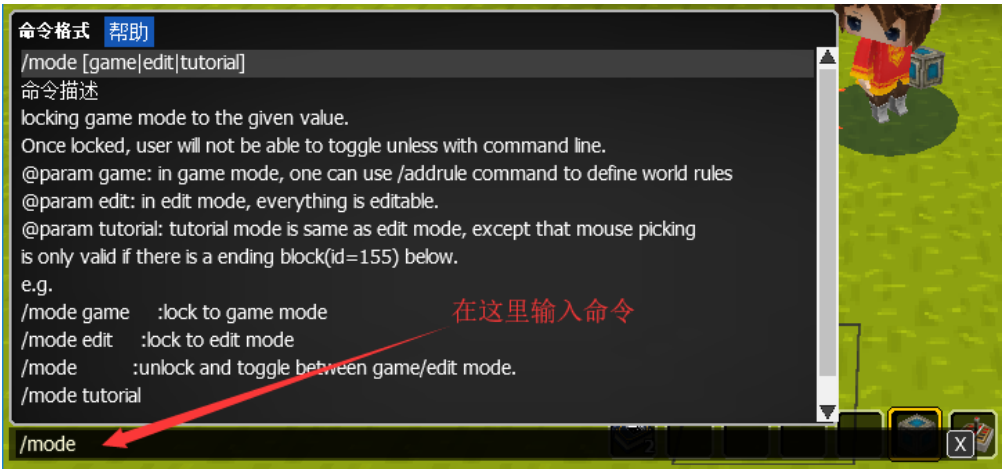
- 另一种切换模式的方式是通过命令。请记住：在大多数工具中，不仅仅是 Paracraft， `你能用鼠标和键盘做的所有操作都可以通过命令来完成，所有命令`

能做的事情也可以通过代码来完成。

- 命令就像是一种更加人性化的代码，具有输入和输出。许多专业程序员只使用命令与计算机交互，这样他们就可以几乎手不离开键盘的操控计算机。

- 因此，现在让我们像一个专业的程序员一样工作，只使用键盘来切换游戏模式。
  - 按下 `Enter` 或者 `/` 键 来打开在屏幕左下角的命令行窗口。
  - 然后输入 `/mode`。记住不要使用鼠标。现在再次按下 `Enter` 来确认命令。
  - 尝试组合输入 `/mode`，`Enter` 几次。恭喜你，你已经学会了第一个指令。
  - 按下 `Esc` 键(在键盘的左上角)来取消一个命令。

- 命令行可以用来做简单的编程，我们将会逐步学习。



在Paracraft使用中，右键单击一个方块意味着编辑或打开它。

- 在编辑模式下，`右键点击` 电影方块读取它的源代码。在这节课，你不需要理解它们。只要四处走走，尽可能多地探索电影方块内的代码。

> 用命令打字是非常重要的，试着提高打字的速度。

- 你可以通过单击上图所示的按钮来在编辑模式下播放动画。
- 如果你不小心进入播放模式，按下`Ctrl+P`键退出电影。

> 如果你离起点太远而走丢了，键入`/home`命令并按回车键将人物传送到出生点。

3 测试题

1. **如何在播放和编辑模式之间切换？**

   - A 按下 Ctrl+G
   - B 按下 Esc 键，并点击屏幕左上角的模式按钮
   - C 按下 Enter 键，并输入 /mode
   - D 按下 / 键，并输入 mode 来切换模式

2. **输入什么命令能让你回到出生点？**
3. **人物移动的快捷键是什么？**
4. **缩放视角的快捷键是什么？**
5. **如何调整视角？**

答案：

1. ABCD 模式切换的方式有很多种
2. /home
3. W A S D
4. Ctrl+鼠标滚轮

5. 按住并拖动鼠标右键

1.1 几何相似与构建相似的虚拟世界

本项目分类是一些和几何相似相关的项目，你将体验下列内容：

- 从几何相似开始，学习和了解几何意义下的放大、缩小、变胖、变瘦。几何相似与时间无关。
- 点阵图与真实图像具有相似性；
- 用方块可以表达二维或三维虚拟世界，虚拟世界是真实世界的相似体。世界由粒子组成，“其小无内，其大无外，以此可造万物”。
- 构建相似的虚拟世界，我们来构建一个地球尺度的虚拟世界。
- 练习打字。
- 用代码创建模型：计算机辅助设计(CAD)。

1.2 虚拟人物与虚拟人物的运动

虚拟人物怎么运动？ 本项目分类是一些和动画与角色运动相关的项目。方块创造的虚拟人物与真实的人物具有一定的相似性。你将体验下列内容：

- 运动需要关节，关节越多越复杂越逼真，真人有360多个关节，我们的虚拟人物通常只有少量关节，比如8个，运动起来已经像模像样了。
- 每个关节都有自己的三维坐标，在场景中运动，还需要场景地面三维坐标以及周围环境的三维坐标。这么多的坐标，随虚拟人物的运动、随时间都在变化，所以涉及极其复杂的坐标变换（要知道普通工业机器人只有6个关节，工作环境与虚拟人物所处环境比简单很多很多）。所以要描述虚拟人物的运动、编辑、存储、复现其运动是极其复杂和困难的世界性难题。目前人类只能通过“示教”来代替编辑。16个关节以上的机器人控制至今还没有人尝试并成功过，而我们的虚拟机器人可以有几十个关节。可以看得出你达到世界水平了。
- 相似原理可以描述多维虚拟人物的运动。而且它的基本步骤比较简单。
- 设计创造虚拟人物的运动、编辑、存储、复现其运动，你做到了。

1.3 构建我的电影世界使它可持续发展

你一定看过很多动画片， 创造动画与电影是计算机技术的一个重要研究方向。人脑的记忆其实可以看成是由很多动画片段组成的。 人类的记忆一但形成， 你很难去更改它。这一点和计算机中的图像与动画很像。我们后面会学习如何用电脑去操纵这些图像与动画，就如同我们的大脑可以按照某种相似的规律触发这些回忆，形成我们的思想，行为，以及我们的梦境。

但是说到底，我们还是要学会如何用电脑去创造出这些动态素材。这是程序员的一个必备技能。 我们认为**每个人都应该能够用计算机去创造一个动画短片作品**。 在Paracraft中你可以探索成百上千的用户创作的动画作品， 你可以去改编它们，或者去创造一个你希望表达的动画作品。

通过本书的学习， 希望你可以像我们的很多学生用户一样， 可以随心所欲的创造任何动画作品，并拥有自己学生时代的代表作。 我们呼吁每个老师和学生都去创造至少一个3D动画作品。作品就是你自己， 作品也是最好的老师。

本项目分类是一些和电影与动画短片相关的项目。**如果你是老师，此时应该启发学生用课余时间完整一个自己的动画作品。**

1.4 如何赋予虚拟人物智能？

计算机从诞生那一天起，就是为了能够代替人类的智能。在前面的章节中， 我们已经学习了如何用计算机创造静态的3D场景， 以及虚拟角色动画。 从本节开始，我们将开始正式通过写代码的方式来控制虚拟人物的运动。

人类的大脑是一台超级计算机。而人类智能的本质其实就是通过层层的相似性去控制自己的记忆（动画）。 你可以将我们记忆中的每个词汇，动作，概念，情绪都看成是虚拟角色。这种抽象能力是人类大脑的核心能力， 也是程序员的核心能力。 当我们用计算机去代替人脑解决问题时，我们要能够将问题抽象成虚拟角色，然后用代码控制它们，这些是本节你要去重点体会和学习的。

本项目分类包含了大量需要你输入代码的项目。 请将你的注意力放到完成项目上，而不要对那些代码追根问底。1989年，我在7岁的时候照着一本书，输入了20行代码，然后我看到电脑屏幕上的图形运动了起来。但是2年以后，我才能真正理解并运用那20行代码。所以你开始要做的仅仅是通过一个一个的项目去体验编程和打字，并争取让程序可以跑起来。

这就如同你小时候学习中文，开始是似懂非懂的模仿，一段时间后，你就可以自由的表达自己了。

在你完成了本节中的至少3个项目时， 你可以跳到第2章去阅读一些编程的基础理论， 同样你不需要完全弄懂，只要量力而行即可。

1.5 保存并分享你的作品

教育的本质就是让人保持思考和一直有事可做。

因此，我们还为Paracraft开发了一个学习平台，叫做KeepWork，官网是：

```
https://keepwork.com
```

KeepWork有2个字面意思：

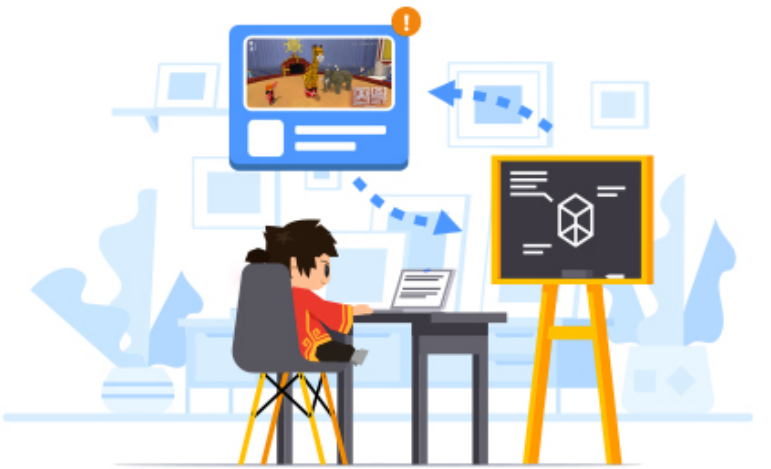
- 保持(keep)有事可做(work)：人不能放弃工作和创作，大人小孩都一样。这个是教育的本质。
- 保存(keep)作品(work)：我们保存了你的所有作品和更改历史。作品是未来教育的重要评估方式，不再需要考试的分数。

当你有了自己的项目或作品，你要做的是公开分享它。 开源和公开是互联网的本质，只有开放的内容才能被搜索引擎检索，才能彼此建立超链接，才能传承下去，让更多的人参与其中。 本项目分类将教会你如何通过个人网站和网页去分享你的作品。你将体验下列内容：

- 使用MarkDown语言创建网页
- 创建超链接
- 引用你的Paracraft作品
- 创建课程

> “在做项目时，你可能需要反复的查看编程理论，直到你真正理解它”

中部：编程理论



“在做项目时，你可能需要反复的查看编程理论，
直到你真正理解它”

2 基础编程理论

在本书的第二部分，我分成7个小节，给大家系统的讲解编程中的重要概念，它们分别是：语法，程序的本质，数字与数学，变量与名字，字符串与文字，表与数组，函数。

请大家跟着我一边打字，一边学习，积累自己的代码量。很多渴望学习编程的人，甚至计算机系的学生最终都没有成为程序员，是因为他们始终没有打过一千行代码。一个勤奋的新手程序员，每个月要打一千行以上的代码，10万行就是高级程序员了。Paracraft大概由50万行NPL代码组成。和学习中文，英文一样，学习计算机语言从7岁到70岁都是可以的。当你有兴趣写到1万行代码时，也许你已经是个不错的程序员了。Paracraft希望带给用户一个有趣的自学环境，但是如果未来你将编程作为职业，还需要更系统的学习计算机软硬件理论，而不只是编程语言本身。

如果你开始看不懂本章的全部内容，没有关系，请先挑选并动手完成本书前面的小项目，然后再反复的阅读本章相应的章节， 你一定会逐渐明白的。

未来的七个章节，我们将从实践的角度让大家学习编程中的基本概念。

2.1 编程基本概念与语法

程序看上去是一行行的代码，写程序时你需要十分的仔细，因为它有很严格的语法，你需要训练你的眼睛，开始会看上去有些难懂，但是随着你阅读和书写代码量的增多，你的眼睛很快会变得更加敏锐。

计算机语言看上去很接近人类的自然语言，但是更加严谨，不能有书写的错误。语言都有一定的语法，计算机语言的语法比大家使用的自然语言的语法要少很多也简单很多。

优秀的程序员可以一目十行，百行，甚至几秒钟浏览上千行的代码结构，我们现在来敲一些代码。

> Ctrl+A可以选中所有代码, 按Del键删除。

我们在场景中创建一个代码方块，右键代码方块，然后输入

```
-- 这里是注释
log("hello world")
```

双横杠`--`是注释，注释后面可以输入任意的文字， 注释并不会被计算机运行，它主要用来生成说明文档和告诉其他程序员后面的代码大概是做什么的。

然后我们在下面输入一个最简单的命令`log`，log是日志的意思，它会将括号中输入的内容显示在下面的日志窗口中。hello world是字符串，前后需要加双引号。关于字符串我会在`2.5节`中详细讲解，然后我们按上面的按钮运行，我们可以看到下方的日志窗口中就显示出了hello world。

我们也可以输出一些其他内容。比如说输出数字1`log(1)`，我们可以看到左下角又出现了1

```
-- 这里是注释
log("hello world")
log(1)
```

我们将上面hello world这行注释掉，我们再输入`一组数据`。

```
-- 这里是注释
-- log("hello world")
log(1)
log({0, 0.5, 1})
```

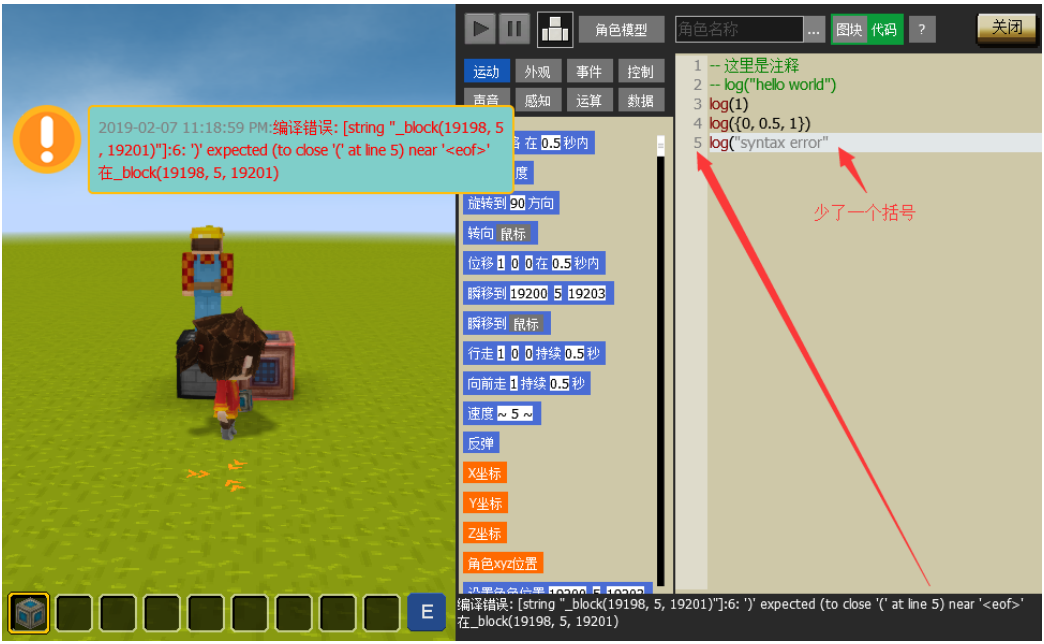
0, 0.5, 1 是一个数组，前后要加大括号，有关于数组，我会在`2.6节`中详细讲解。运行后我们可以看到这组数据就在下方显示出来了，同时大括号也显示出来了。由于hello world这一行已经注释掉了，所以没有输出，不显示。下面为日志窗口的输出：

```
1
{0,0.5,1,}
```

如果我们打错了，比如我们少打了括号：

```
-- 这里是注释
-- log("hello world")
log(1)
log({0, 0.5, 1})
log("syntax error"
```

运行后，下方会告诉我们有一条语法`编译错误`。
`[_block(19198,5,19201)]:6:')' expected to close '('at line 5`



中文的意思是在3D坐标为`19198,5,19201`的代码方块中的第6行：需要有一个右括号来关闭第5行的左括号。所以我们在后面加上右括号即可修复错误。

```
-- 这里是注释
-- log("hello world")
log(1)
log({0, 0.5, 1})
log("syntax error")
```

优秀的程序员可以很快地看出代码中的错误，所以我们要在不断的实践中训练自己的眼睛，不断的试错和实践。

下面我来给大家解释一下什么是`编译`？
因为计算机程序都是有严格的语法的，所以程序在运行前会检查所有的语法是否有错误，没有语法错误的高级语言会转换成更基础更底层的计算机指令给计算机执行。这种底层的计算机指令只和硬件也就是CPU（中央处理器）有关。

> 所有的计算机语言都需要被转换成这种底层硬件指令才能被执行，这个过程就是编译。

编译只在程序运行前执行一次。程序中的错误一般有两类，一类是在编译时出现的语法错误，也就是我们刚刚讲的`编译错误`。语法错误不修复程序是无法执行的。还有一类错误叫`运行时错误`，也叫Runtime error，是指我们的代码存在逻辑错误，导致程序可以执行，但并没有输出我们想要的结果或者程序运行的过程中出现了错误。编译错误是计算机可以自动帮我们找到的，所以很好修复。

例如刚刚第5行出现的错误，但是`运行时错误`则需要大量的时间反复运行调试，因为这是我们在编程时出现的逻辑错误。上面的log语句就是一种程序员们经常使用的寻找`运行时错误`的方法。程序员一般会在代码的关键位置添加一些log语句。程序员通过分析日志，可以很快的找到运行时错误出现的时间和位置，所以写日志(log)是优秀程序员的好习惯。比如我们可以写像下面这样的日志：

```
log("核心游戏逻辑加载成功")
log("地图正在加载...")
log("警告：地图加载失败...")
log("AI系统初始化完毕")
```

计算机语言有很多种，一个高级程序员一般可以精通其中的4到5种，并了解其他的十几种。

学习一门新语言需要记住的全部语法一般只有20个。例如我刚刚讲的代码中的`--`表示注释，
`()`小括号表示指令的输入，字符串前后要加双引号`""`，大括号`{}`表示一组数

据，其中的内容用逗号`,`隔开。以上这些都是计算机语言的语法。未来几节，我会给大家详细的讲解这些语法。

对于一个高级程序员，一般只要1到3天就可以掌握一门新语言，但是对于新手却需要很长的时间去训练自己的手,眼,大脑的协调性。不同的计算机语言的语法是有差别的，但对于高级程序员来说，它们大多是相似的。

NPL语言的语法与lua语言兼容，非常适合教学。Lua的语法是全世界人工智能，游戏开发等领域最受程序员喜欢的语法之一，因为它十分的简洁，又与C/C++语言很类似。同时NPL也是一种通用语言。所谓通用语言就是这门语言可以编写任何程序，而没有限制。

最后我来讲一下如何学习一门新的编程语言：

- 首先我们要多写代码，提高自己的打字速度，让自己的眼睛对语法更加的敏感。
- 第二点，多看像这样的入门教程，看完后自己设定一些目标来练习。
- 第三点要提高自己的英文能力。全世界的计算机语言以及他们的官方文档都是英文的。大家看完教程后应该尽可能的去看官方文档，只有这样才能向全世界的其他程序员一样获取知识以及寻求问题的答案。NPL语言的官网是:
  `https://github.com/LiXizhi/NPLRuntime`

如果你的英语足够好，懂得至少一门计算机语言，那么学习任何一门语言，只需要看官网就可以了。本书希望你通过Paracraft，首先学会NPL语言。

2.2 程序的本质

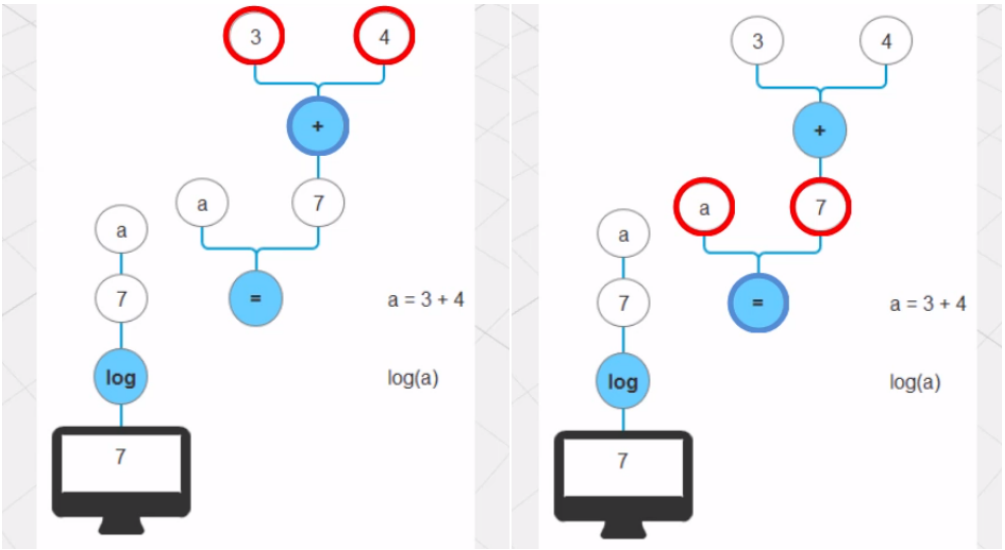
> 程序的本质就是输入和输出。

每一行语句就是将输入和输出连接起来，程序都是按一定顺序执行的，只有在上一行的输入和输出执行完毕后，才会启动下一行的输入和输出，也就是行与行之间是一种隐性的连接关系。

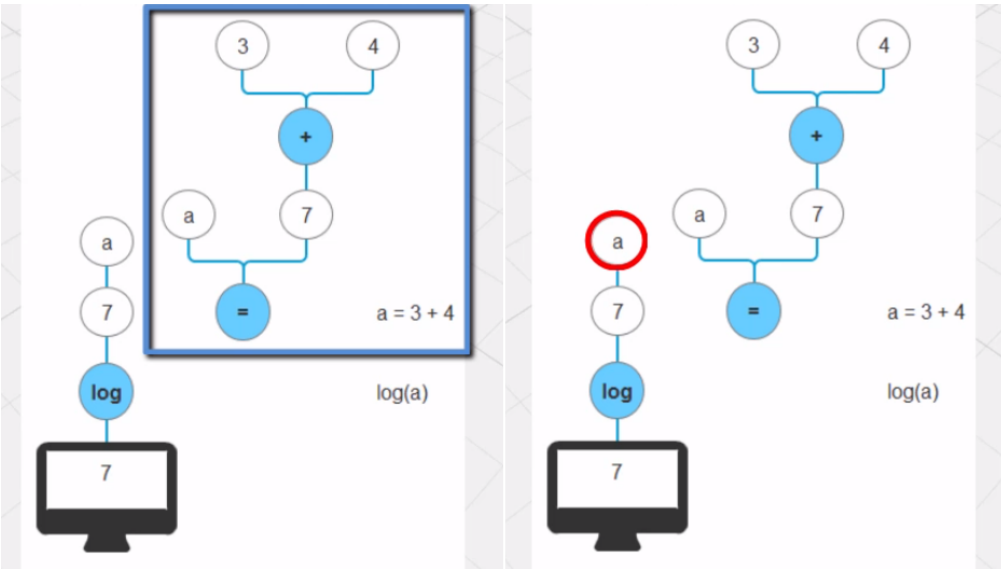
下面我们来看一个例子

```
-- 程序的本质
a = 3 + 4
log(a)
```

`a=3+4; log(a)` 运行后可以看到日志窗口中显示出了7。我们将右侧代码输入输出的连接关系以图形的方式展现出来。如下方左图:



在这里3和4是输入，加号 `+` 是一个运算函数。关于函数的概念，我会在 `2.7节` 中给大家详细讲解。这里大家先简单地理解为对输入3和4进行加法运算并输出结果。等号 `=` 与加号 `+` 类似，也是一个运算函数，它的作用是将等号右侧的输入，也就是加号函数的输出，赋给等号左侧的输入，也就是将右侧3+4的输出结果7再作为输入赋给左侧的变量a。如上方右图，等号的左右两侧都是等号函数的输入，等号函数并没有任何输出。有些语言中例如C语言中，等号函数还会输出赋值后的结果，但是NPL语言中是不输出的。所以我们看到一行很简单的，看上去很像数学表达式的代码 `a = 3 + 4`，其实是加号和等号两个函数输入和输出链接的结果，如下方左图。



`a` 是一个变量，在程序中没有加双引号的文字基本都是变量，纯数字和符号除外。更确切地说，由字母以及下划线组成的任意单词都是变量。如果在 `log(a)` 中将a的前后加上双引号 `log("a")`，那么a就变成了字符串，左下角的日志窗口就会显示字母a而不是数字7。

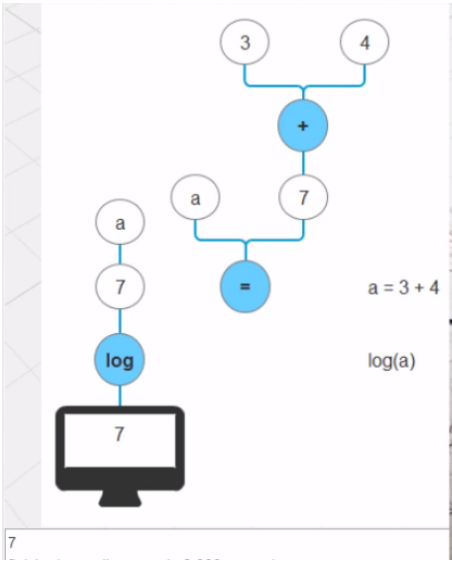
```
a = 3 + 4
log(a)
log("a")   -- 显示字符串a
```

关于变量我会在 `2.4节` 中详细讲解。这里大家先简单地理解为变量就是某个存储单元的名字，变量默认会输出它所代表的存储单元，等号赋值除外。例如在 `log(a)` 中变量a会输出数字7，7又成为了log函数的输入。log函数最终在日志窗口中输出数字7， `log(3+4)` 也会在左下角输出数字7，所不同的是3+4输出的结果7输入到log函数

后会马上被系统释放掉。但是在 `log(a)` 中变量a始终指向7，所以a对应的存储单元中的7不会被马上释放。

一般来说，所有没有被变量命名的输出结果都会很快被系统自动释放掉，这样就不会占用计算机的内存了。内存是计算机的存储单元，程序在执行时，所有的代码，也就是输入输出函数等都会变成内存中的存储单元。也就是说上图中所有的圆圈都对应着内存中的存储单元，我们写的程序建立了这些存储单元之间的输入输出关系。

对于初学者来说，从这种视角看程序，会有些繁琐和复杂，但是它的确可以帮助初学者真正的理解代码的结构，等你阅读了大量代码后，你的大脑会自动的去理解这些输入输出的关系，你甚至感觉不到它们的存在。

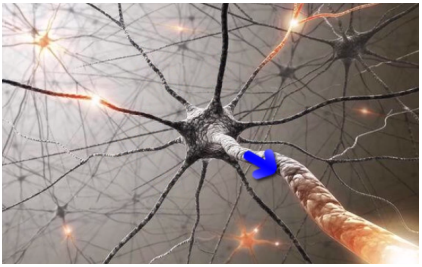



```
-- 2. 程序的本质
-- 程序的本质就是输入和输出。 每一行语句就是将输入和输出连接起

a = 3 + 4
log(a)

--[[ 程序就如同我们的大脑。
人类的大脑由上百亿的神经元组成，
每个神经元有上万个来自其它神经元的输入
和一个到其它神经元的输出，总共有百万亿的链接。
所以人脑就像一台超级计算机，
神经元的输入和输出连接关系就如果我们的代码。]]
```

程序就如同我们的大脑。人类的大脑有上百亿的神经元细胞组成，每个神经元细胞有上万个来自其它神经元的输入和一个到其他神经元的输出，总共有百万亿的链接。如下图。



所以人脑就像一台超级计算机，神经元的输入和输出的连接关系就如同我们的代码。NPL是一种高级语言，上面代码中的 `log(a)` 其实经历了很多你看不见的底层代码的输入和输出，最终才到达屏幕上你看见的由像素组成的数字7。

2.3 数字与数学

在 `2.2节` 中我们了解到，程序的本质就是输入和输出的连接。在这一节中，我们先来看一种最简单的输入到输出的关系，就是数字与数学。其实计算机语言只有最基本的几种内置数学函数，它们都只有两个输入和一个输出。这些数学函数的语法比较特殊，他们使用了我们熟悉的数学符号，符号的左右两侧是输入。

这些数学函数，包括 `加+` `减-` `乘*` `除/` 和一些比较操作符，包括 `大于号>` `小于号<` `小于等于号<=` `大于号等于号>=` 。其他高级的数学函数都是程序员通过这些简单的内置数学函数的组合实现的。所以计算机语言中的全部数学函数是小学生可以掌握的。下面我们逐一来看一下，我们来敲入一些代码，请大家在学习时也务必尽可能多的自己敲入代码来练习并积累代码量。现在我们敲入一行代码:

```
-- 数字与数学
log(25 + 30 / 6)  -- 30
```

运行后，日志窗口中出现了输出为30，这里除法函数会优先于加法函数，所以除法的输出为5，加法的收入为25和5，所以最后的结果是30。下面我们再尝试一些其他的写法。

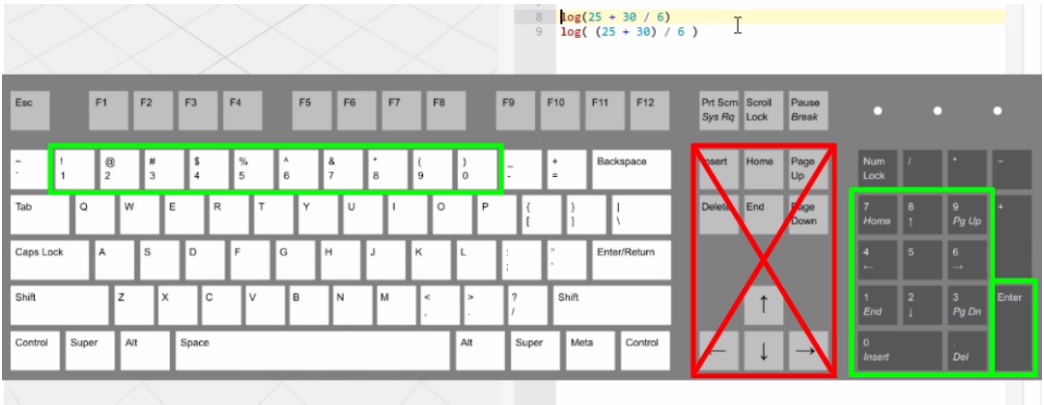
```
-- 数字与数学
log(25 + 30 / 6)
log( (25 + 30) / 6)  -- 9.16667
```

log括号25加30括号完毕除以6，和我们数学课上学到的语法很像。在程序中我们可以用括号 `()` 将一段代码括起来，强制这段代码作为一个整体输出结果。代码中的空格是不影响程序执行的。这里我们为了美观，在数字间统一加上空格，可以看到，虽然输入的数字和前一行代码相同，但是加号函数的输入却变成了25和30，括号中的代码的输出为55，除法的输入为55和6，所以最后我们看到结果为9.16667。下面我们修改一下代码，将前面的代码注释掉。

```
-- 数字与数学
-- log(25 + 30 / 6)
-- log( (25 + 30) / 6)
```

关于打字与快捷键

大家注意一下，我们在打代码时，应该尽量使用快捷键，按`Home键`可以迅速让光标移动到一行的开始。同理按`End键`可以移动到一行的结束。这些快捷键大家要反复练习来提高自己的打字效率，不能只依靠鼠标和方向键。这里推荐大家使用标准键盘右侧数字键盘区的方向键以Home和End键，数字则用左侧区域输入，这样会比使用中间区域提高更多的打字速度。



很多资深的程序员习惯完全不用鼠标编程，所以大家要多多练习打字与快捷键。下面是常用快捷键，掌握它们你将不再需要鼠标：

- Ctrl + 左右键：前进后退一个词
- Home, End, PgUp, PgDn:行首，行末，前翻页，后翻页
- Shift + Ctrl + 左右键： 以词为单位选择
- Shift + 上下左右键或Home或End： 选择
- Del和Backspace： 都是删除，shift选择状态下Del会更方便。
- Ctrl + C和V：复制和粘贴

```
-- 数字与数学
-- log(25 + 30 / 6)
-- log( (25 + 30) / 6)

log( (3+2) < 5.1)
```

下面我们继续来看一些其他的数学函数，比如(3+2)是否小于5.1，我们看到下方输出了`true`是一个常量，在程序中表示`真`。也就是说3+2真的是小于5.1的。我们再来看一个例子，1是否等于1？

```
-- 数字与数学
-- log(25 + 30 / 6)
-- log( (25 + 30) / 6)

log( (3+2) < 5.1) -- true
log(1 == 1)  -- true
```

等于等于号`==`是用来判断左右两侧的输入是否是完全相等的。如果是相等的，则输出`true真`。如果不是相等的，则输出`false假`。我们执行后发现上面两行代码都输出了true，我们用快捷键移动光标快速的注释掉刚刚的代码。我们再来看一下，2是否小于等于1，我们可以看到下面的结果是false。false是假，也就是二不是小于等于1的。

```
-- 数字与数学
-- log(25 + 30 / 6)
-- log( (25 + 30) / 6)

-- log( (3+2) < 5.1)
-- log(1 == 1)

log(2 <= 1) -- false
log(5 > -2) -- true
```

我们再来看`5>-2`。我们看到结果是真。

```
log(21%5) -- %是取模的意思，21%5的输出为1

log(2^10) -- 2*2*2*2*2*2*2*2*2*2*2=1024
```

还有一个特殊的函数操作符`百分号%`，在NPL语言中是取模的意思，也就是21除以5的余数是多少。我们看到输出的结果是1，所以`左侧输入除以右侧输入的余数`就是取模。我们看到注释`--`也是可以加到一行代码的后面的。

最后还有一个特殊的内置函数：`次方^`。例如2的10次方就是2×2x2，一直乘以2，要乘10次，结果是1024。以上就是NPL语言中你能用到的全部数学函数了。你可以在代码方块的`运算标签`下查看这些数学函数的文档和例子。

2.4 变量与名字

在`2.2节程序的本质`中，我们已经使用过变量。这一节我们来更深入的学习变量与名字。程序中的变量英文是Variable，与我们数学课上所说的函数变量不是一个意思。

> 程序中的变量只是某个存储单元的名字，它会直接输出程序执行的瞬间它所代表的存储单元。

```
a = 3 + 4
log(a)
```

上面的代码，我们运行一下，可以看到输出是7。在这里a是一个变量，它所代表的存储单元的数据是7。变量只是高级计算机语言中的一种重要手段，方便在代码中快速的连接输入和输出。实际上计算机执行的底层硬件（CPU）指令是不存在变量的，在硬件中只有存储单元的数字地址，所以变量的名字我们可以随意起。程序员喜欢用一些很长的易于理解的单词或单词组合作为名字。例如下面这个例子：

```
char_size = 200
scaleTo(char_size)
```

`char_size`等于200，scaleTo放大到char_size。我们运行一下，可以看到人物放大到了200%，在这里`char_size`是一个变量，我们从名字中就可以看出，它代表了人物的大小（character size）。同样的代码，我们也可以这样写:

```
s = 200
scaleTo(s)
```

很明显，使用char_size作为名字比使用s好得多。因为长名字更清楚明了，易于理解。

在NPL语言中，同一个变量可以在程序运行的不同时刻指向任何类型的存储单元。例如:

```
a = 1
a = "hello"
log(a)
```

我们把前面的代码删除，输入上面的代码并重新运行一下。`log(a)`输出的结果为hello，而不是1。变量在同一时刻只能代表一个存储单元。a等于hello执行后，变量a已经不再指向1，而是指向字符串hello所在的存储单元。所以`log(a)`输出的是当时a所指向的存储单元，也就是hello。

> 变量是有生命周期的，变量的生命周期在程序中叫做作用域(Scope)。

在NPL语言中，变量的作用域是我们可以使用这个变量名的代码区间。在NPL语言中有两种方式定义变量的作用域，一种是本地的变量（local），一种是全局的变量(默认的)。我们来看一个例子:

```
a = 1
local b
b = a + 2
b = b + 3
log(b)
```

每个代码方块其实是一个独立的文件。在NPL语言中，如果我们在文件中直接使用一个变量，例如上面的a等于1，则变量a的作用域默认是全局的。全局变量可以在所在

文件的任意位置使用，默认情况下也可以在其他文件中使用。关于如何将程序拆分为多个文件，我会在 `2.7节` 函数中讲解。这里大家简单理解为，文件就是函数，也有输入和输出，可以相互连接。

我们继续看第二行 `local b` 它定义了一个 `本地变量` 。这里 `local` 是一个系统内置的特殊名字，表示用它后面输入的变量名字重新定义一个新的本地变量。b的作用域是从local开始，直到所在文件或函数的结束。

变量b无法在local之前的代码中使用，也无法在其它文件中使用。所以一般来说，本地变量的生命周期是从local开始到文件或函数执行结束。

这里大家首先要明白，变量只是存储单元的一个名字，因此变量的生命周期和它所代表的存储单元的生命周期没有关系。一个变量在程序运行的过程中可以代表不同的存储单元，就像例子中这样；同样一个存储单元也可以有多个不同的变量名字。例如：

```
a = "hello"
b = a
log(a)
log(b)
```

b和a指向的是同一个字符串存储单元即hello。注意有一些编程语言中，等号函数会复制一份新的相同内容的存储单元，但是NPL中，a和b是指向同一个存储单元的，所以 `log (a)` 和 `log(b)` 的输出都是hello。

在NPL中当一个存储单元没有任何变量指向它时，存储单元会被系统自动释放掉。所以很多时候变量和它所代表的存储单元的生命周期是几乎一样的。所以我们应该尽量多用局部变量，让变量的生命周期尽可能短一些，这样可以节约内存，代码也执行的更快。

> 写代码时，能用局部变量就绝不用全局变量

我们再回到最初的代码。

> `Ctrl+Z` 可以回滚我们刚刚敲入的代码。

这里我们在代码中加入一些log语句。

```
a = 1
local b
log(b)  -- nil
b = a + 2
log(b)  -- 3
b = b + 3
log(b)  -- 6
```

我们先看第一个`log(b)`。此时b没有指向任何存储单元，所以输出的结果为nil。

> nil在NPL中是一个常量，代表一个无效的存储对象。

然后我们将a+2输出3赋给b，此时第二个`log(b)`的输出结果为3，我们也可以合并一下，直接写成`local b = a + 2`。

```
a = 1
local b = a + 2
log(b)  -- 3
b = b + 3
log(b)  -- 6
```

同理下一行`b = b + 3`有加号和等号两个函数先后执行。b开始会输出3，3+3输出了6，等号函数又将6赋给了b所以最后一个`log(b)`输出了6。我们可以看到在程序运行过程中，本地变量b所代表的存储单元发生了变化。NPL中的存储单元也叫对象。

> 对象的类型只有7种，分别是数字，字符串，函数，表，true，false和nil。

变量可以给所有7种对象起名字，上面代码中的`log`实际上也是一个全局变量。当变量指向一个函数时，我们习惯性的将变量名说成是函数，所以log函数也是log变量。代码中的1，2，3是数字，hello是字符串， 等号和加号是数学函数。

在程序中没有加双引号的文字都是变量，更确切的说，代码中由文字和下划线组成的任意单词都是变量（只有少数几个例外，比如local）。在NPL中变量区分大小写，因此小写a和大写A是两个不同的变量，和数学中的变量命名不同，程序员一般会给变量起一个很长的便于识别的名字，并习惯遵守一些人为的规则。例如`count, char_size, length, walkRadius`都是不错的变量名。NPL支持使用中文作为变量名，但是我们不建议大家这样做，一个原因是打字时你需要不停的切换输入法影响打字速度，最重要的原因是你很难用中文创造新的不存在的短词汇。

下面我们来看变量在程序中的用途。我们用代码方块编一段代码:

```
say("hello! ".."xxx")
tip("hello! "..."xxx")
```

我们运行一下，可以看到人物说"hello! xxx"。这段代码其实是用say函数让角色说一句话，再用tip函数在屏幕中央打印一行文字。

`..` 和 `+` 函数类似，是将左右2个字符串合并，并返回结果。关于 `..` 函数和字符串我们在 `2.5节中` 会详细讲解。这里 `"hello! "..xxx"` 会返回一个新的字符串 `"hello! xxx"` 。

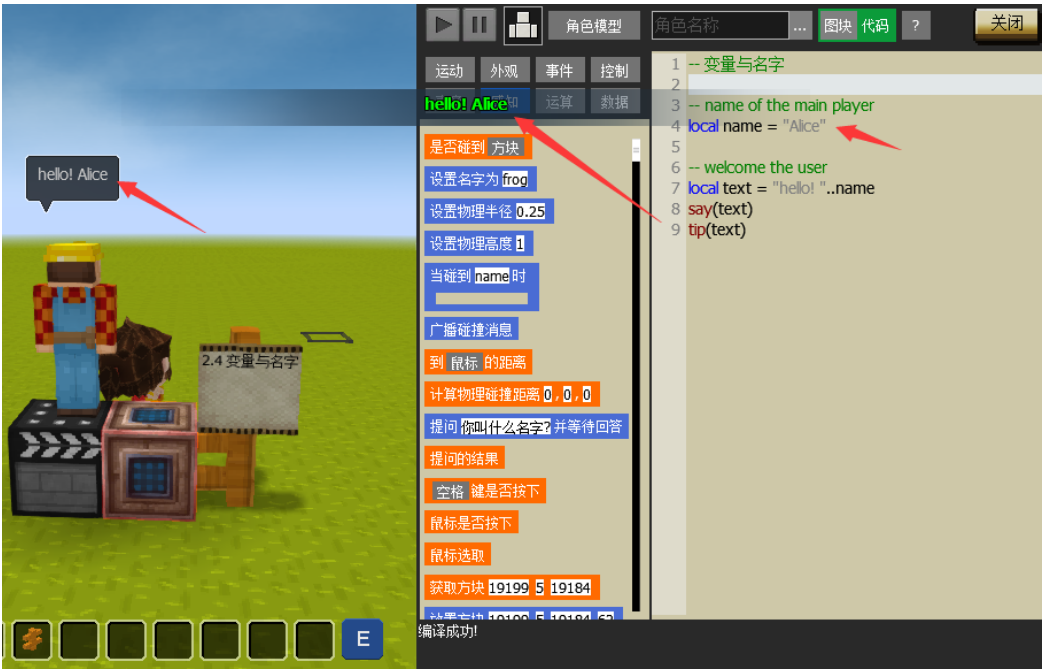
下面我们再用变量改写一下上面的代码：

```
-- name of the main player
local name = "xxx"

-- welcome the user
local text = "hello! "..name
say(text)
tip(text)
```

先定义一个本地变量name等于"hello"，我们在前面加上一行注释，也就是角色的名字。
然后再定义一个本地变量text，将 `"hello! "..name` 的结果赋给它。 然后我们再分别调用say和tip函数。我们运行一下，结果是一样的。

加入变量name，代码变得更加容易理解和方便修改了，我们只需要看到注释和变量名，就基本理解了下面一段代码是做什么的，并且还可以直接修改变量的输入，来影响后面整段代码的输出。例如我们将name改成Alice，重新运行，可以看到2个地方都变成了"hello! Alice"。如下图:



变量是编程中最难理解的概念，程序员在编程时有一大半的时间都在思考代码中需要哪些变量，是本地的还是全局的，以及用什么通俗易懂的名字。后面我们还会看到函数名本身也是变量，所以变量无处不在。

> 变量也可以看成是计算机语言的词汇。写代码的过程其实就是在不断创造新的词汇，并用这些词汇去描述我们要解决的问题。

代码的90%都是各种变量名。高级和初级程序员代码的主要区别就是在变量的使用和命名上。优秀的程序员能够创造大量简单易懂的词汇去描述和解决问题。初级程序员往往只用少量晦涩难懂的词汇解决问题。和用自然语言写文章一样， 我们要多读，多写才能写出优美的程序来。另外就是鼓励大家学好英语，因为世界上99.99%的程序的变量名都是英文的，所以只有学好英语才能读懂别人的代码，创造出良好的变量名。

2.5 字符串与文字

在计算机语言中，字符串(string)是一种最常用的存储单元的类型。例如 `log("123")` 中 双引号中的 `123` 代表一个长度为3的字符串。

> 字符串是一定长度的二进制数据。

NPL语言中，字符串的长度单位是字节（Byte）。一个Byte包含8个Bit，也就是8个0或1的组合，因此一个Byte最多可代表2的8次方256种不同的组合。长度为1024的字符串就代表1KB的数据，1024KB=1MB（也就是1兆），1024MB=1GB

字符串的一个最大用途是用来代表自然语言中的文字。 字符串的每个Byte对应到文字的映射规则叫做编码。NPL中默认的编码规则叫做UTF8，UTF8是全世界使用最广泛的编码规则，几乎互联网上所有的文字都是这种编码。这种编码将每个英文字母或数字映射到一个Byte，将中文或其它特殊字符映射到2个或多个Byte。例如字符串"123"中每个数字字符都对应一个Byte，也就是对应256种0,1组合中的一种。

我们看到，代码本身就是由文字组成的。 在NPL中，所有在`""`或`''`的中的文字都是utf8编码的字符串。变量可以指向字符串，例如：

```
local a = "Hello"
local b = 'World'
```

这里介绍一个特殊的内置函数`..`，例如`log(a..b)`：

```
local a = "Hello"
local b = 'World'
log(a..b)
```

`..`函数可以将左右两侧输入的字符串合并，并输出一个新的合并后的字符串。`..`的作用类似`+`号函数，只不过输入是字符串。所以我们看到`log(a..b)`在左下角的输出为Hello与World合并后的新字符串，也就是"HelloWorld"。

除了`..`函数，NPL中还有很多其它操作字符串的函数。例如，在一个字符串中查找或匹配另外一组字符串。 但是这里我只介绍如何用大家熟悉的log函数来操作字符串。

log其实是一个多输入的函数。如果它的第一个输入是字符串，并且字符串中包含`%d,%s`等特殊占位符的文字，则它后面的输入会依次替换前面对应的占位符，并输出替换后的字符串结果。例如：

```
local a = "Hello"
local b = 'World'
log(a..b)
log("x=%d, y=%s, z=%d", 1, "hello", 2)
```

这里log的第一个输入是字符串，内容是`"x=%d,逗号空格 y=%s, 逗号空格 z=%d"`。这里双引号中的所有文字，例如空格，等号，逗号都不是代码，而是字符

串中的数据， 占一个Byte。log的第二个输入1会替换字符串中的第一个%d，第三个输入"hello"会替换%s，第4个输入2会替换后面的%d，所以最后的输出为`"x=1, y=hello, z=2"`。

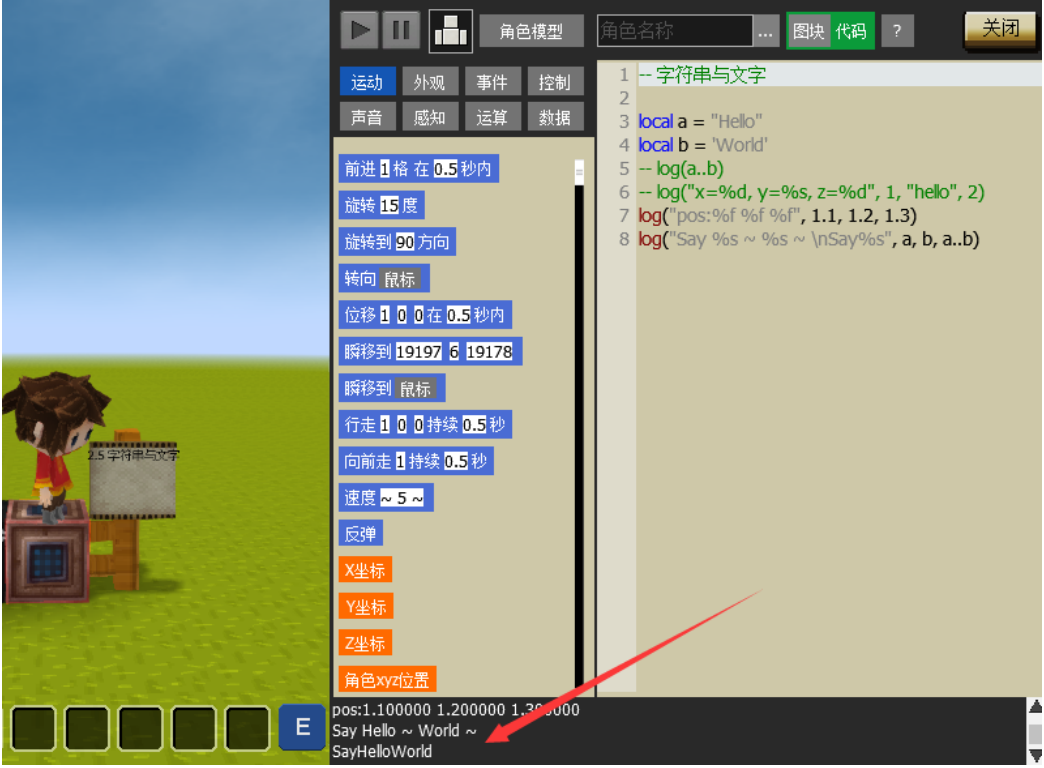
`%d`代表替换的对象是一个十进制的整数(decimal)，也就是1和2；`%s`代表替换的对象是字符串，也就是"hello"。注意这种替换规则只是log函数内部的逻辑，和字符串本身和NPL语言无关。只不过在很多语言中都有类似规则的函数。

下面我来随意练习一下，先注释掉之前的例子。我们敲入

```
local a = "Hello"
local b = 'World'
-- log(a..b)
-- log("x=%d, y=%s, z=%d", 1, "hello", 2)
log("pos:%f %f %f", 1.1, 1.2, 1.3)
```

`"pos:%f 空格%f空格 %f"`是一个字符串，%f之外的内容可以随意输入，中英文都可以，这里pos的中文意思是位置。
`%f`和`%d`一样，也是特殊占位符，只不过`%d`代表整数，`%f`代表浮点数(float)，默认会显示小数点后6个有效数字。我们再继续敲入：

```
local a = "Hello"
local b = 'World'
-- log(a..b)
-- log("x=%d, y=%s, z=%d", 1, "hello", 2)
log("pos:%f %f %f", 1.1, 1.2, 1.3)
log("Say %s ~ %s ~ \nSay%s", a, b, a..b)
```



这里`"Say %s ~ %s ~ \nSay%s"`是一个字符串，其中`\n`是换行字符，占一个Byte，也是字符串的一部分，`\n`会使log在屏幕上的输出另起一行。变量a代表字符串"Hello"，它会替换第一个%s；变量b代表字符串'World'，它会替换第二个%s；`a..b`的输出是字符串"HelloWorld"，它会替换第三个%s，所以log函数最后输出的是`Say Hello ~ World ~`，第二行就直接是`SayHelloWorld`，中间没有空格。对于不清楚的内容，请大家自己写一些代码，进行大量的练习。

这里再介绍一个非常用的操作字符串的函数format。format函数的输入和log函数非常类似，只不过它的输出为一个字符串。例如下面的代码和之前的结果是一样的。

```
local a = "Hello"
local b = 'World'
-- log(a..b)
-- log("x=%d, y=%s, z=%d", 1, "hello", 2)
local msg = format("pos:%f %f %f", 1.1, 1.2, 1.3)
log(msg)
msg = format("Say %s ~ %s ~ \nSay%s", a, b, a..b)
log(msg)
```

字符串在计算机语言中使用十分广泛。 字符串中的文字可以用来代表文件名，变量的名字，屏幕上的文字或任何2进制数据。这些字符串的用法我会在后面的章节中继续讲解。

2.6 表与数组

我们之前讲过， NPL语言中的全部数据类型是： `数字，字符串，表，函数，true/false/nil` 。
我们只剩下表和函数没有讲了。 本节我们讲解：表(Table)。

> 表是数据到数据的映射关系。

表就如同文件夹或字典，例如下面从中文字符串到英文字符串的映射就可以用一个表来存储。这里是从左向右映射：左侧的数据叫做关键字(key)，右侧的数据叫做数值(value)。

我们可以用 `{}` 来创建一个空的表。例如:

```
a = {}
```

此时变量a代表了一个空的表。 下面我们向表a中插入上述数据的映射关系。

```
a = {}
a["苹果"] = "Apple"
a["右侧"] = "Right"
a["正确"] = "Right"
```

`a[key]=value` 是一种特殊的函数形式，它有3个输入，分别是a，key，value。它的作用是向表a中加入一个从关键字key到数值value的映射。关键字key可以是除了nil外的任何数据类型，所以我们也可以插入Apple到苹果的映射。

```
a = {}
a["苹果"] = "Apple"
a["右侧"] = "Right"
a["正确"] = "Right"

a["Apple"] = "苹果"
```

`表`最重要的功能是，输入任何一个关键字key，快速的输出对应的数值value。即使表中有成千上万的映射，也可以快速的返回结果。例如:

```
a = {};
a["苹果"] = "Apple"
a["右侧"] = "Right"
a["正确"] = "Right"

a["Apple"] = "苹果"

log(a["苹果"]) -- aple
```

`log(a["苹果"])` 会输出Apple

上面的`a[key]`中的`[]`其实是一个特殊的内置函数， 它有2个输入，分别是左侧的表a和方括号中的关键字key，它的输出是表中关键字所对应的数据，如果不存在这个映射，则输出nil。例如

```
a = {}
a["苹果"] = "Apple"
a["右侧"] = "Right"
a["正确"] = "Right"

a["Apple"] = "苹果"

log(a["苹果"]) -- Apple
log(a["橘子"]) -- nil
```

如果关键字是字符串，并且符合变量的命名规则， 则`a[key]`以及`a[key]=value`函数在NPL中还有一种更简单的形式 `a.key`和`a.key=value`。 `.`的左右侧的输入分别是表和关键字。例如:

```
a = {}
a["苹果"] = "Apple"
a["右侧"] = "Right"
a["正确"] = "Right"
```

```
log(a["苹果"]) -- Apple
log(a.苹果) -- Apple
```

`log(a["苹果"])`和`log(a.苹果)`都会输出"Apple"，这样我们可以省去`[""]`，让代码更容易阅读。

```
a = {}
a["苹果"] = "Apple"
a["右侧"] = "Right"
a["正确"] = "Right"


log(a["苹果"]) -- Apple
log(a.苹果) -- Apple

a["橘子"] = "orange"
a.橘子 = "orange"
log(a.橘子) -- 橘子
```

同理`a["橘子"]="orange"`与`a.橘子="orange"`是等价的。`log(a.橘子)`会输出orange。

你也许会问，使用`.`函数访问数据的方式和变量很像。 没错，其实NPL中所有的变量，都是通过表来存储的。

> `变量`就是变量名(字符串)到变量所代表的对象的映射。

在NPL中所有的全局变量都存在一个全局表变量`_G`中。例如，下面3行语句是等价的:

```
_G.a = {}    -- 其它地方可以用 a = {}
a["苹果"] = "Apple"
a["右侧"] = "Right"
a["正确"] = "Right"

log(a.苹果)    -- Apple
log(_G["a"]["苹果"])   -- Apple
log(_G.a.苹果)   -- Apple
```

上面三个log都输出Apple。所以访问一个全局变量a其实是调用了函数`_G["a"]`，本地变量（local variable）情况特殊一些，但是基本原理是一样的。

> 注意：代码方块中的全局变量被存在了另外一个独立的表中，而不是默认的全局表。如果你希望表a可以在多个代码方块中使用， 你需要用`_G.a = {}`，也

> 就是向全局表_G中插入字符串"a"到空表{}的映射，或者调用`set("a", {})`函数。 在常规的NPL代码中，你可以直接用`a = {}`向默认全局表中写数据。

计算机如何通过变量的名字在某个表中找到它对应的对象（存储单元）是一个很比较复杂的事情。 在NPL语言中，这个过程是通过一个叫做`MetaTable`(原表)的概念实现的。通过Metatable，程序员可以自定义`[]`和`.`函数针对某个表的输入/输出映射规则。但是这个概念对初学者太复杂了，这里就不讲解了。在代码方块中，大家并不需要知道原表，每个3D世界中所有的代码方块共用一个名字为`_G`的全局表。 最后，大家记住一个规则，就是尽量不要用全局变量。如果你一定要用，你要清楚它存在了哪个全局表中，因为全局表可能不只一张。

一个表对象中的所有关键字必须是彼此不同的， 然而数值可以相同，并且可以是任意的数据类型，包括其它表。
例如，变量_G中包含了字符串"a"到表a的映射，表a又包含了从"苹果"到"Apple"的映射。

可以说代码方块中的数据(全局变量，系统函数等等)都在_G表中， 所以计算机程序其实是保存在一个由表构成的树型结构中，类似文件夹一样；最上面的一层就是_G表。通过这张表，我们可以通过关键字，找到程序中的所有全局数据。

最后我们再介绍一种创建表的方法，它可以让你的代码更简短。

```
a = { ["苹果"]="Apple", Apple="苹果", 右侧 = "Right", ["正确"] = "Right" }
```

如上，我们可以直接在{}中用逗号分隔每个数据映射。这样就不用一个一个插入了。为了美观，我们也可以加入空格和回车。

```
a = {
  ["苹果"] = "Apple",
  Apple = "苹果",
  右侧 = "Right",
  ["正确"] = "Right"
}
```

细心的人会发现，其实如果等号左侧的关键字是符合变量命名规则的字符串，则可省去`[""]`，例如Apple和右侧我们就没有加`[""]`。

这里要注意的是， 表中的映射是不记录添加的先后顺序的。所以下面的写法也是等价的。

```
a = {
  右侧 = "Right",
```

```
  ["正确"] = "Right",
  ["苹果"] = "Apple",
  Apple = "苹果",
}
```

如果关键字不是字符串， 而是连续的从1开始的整数，例如：

```
a = {
  [1] = "one",
  [2] = "two",
  [3] = "three",
}
log(a[2]) -- 会输出"two"
```

此时，有一个简单的写法，可以忽略前面的整数关键字,方括号以及等号，写成

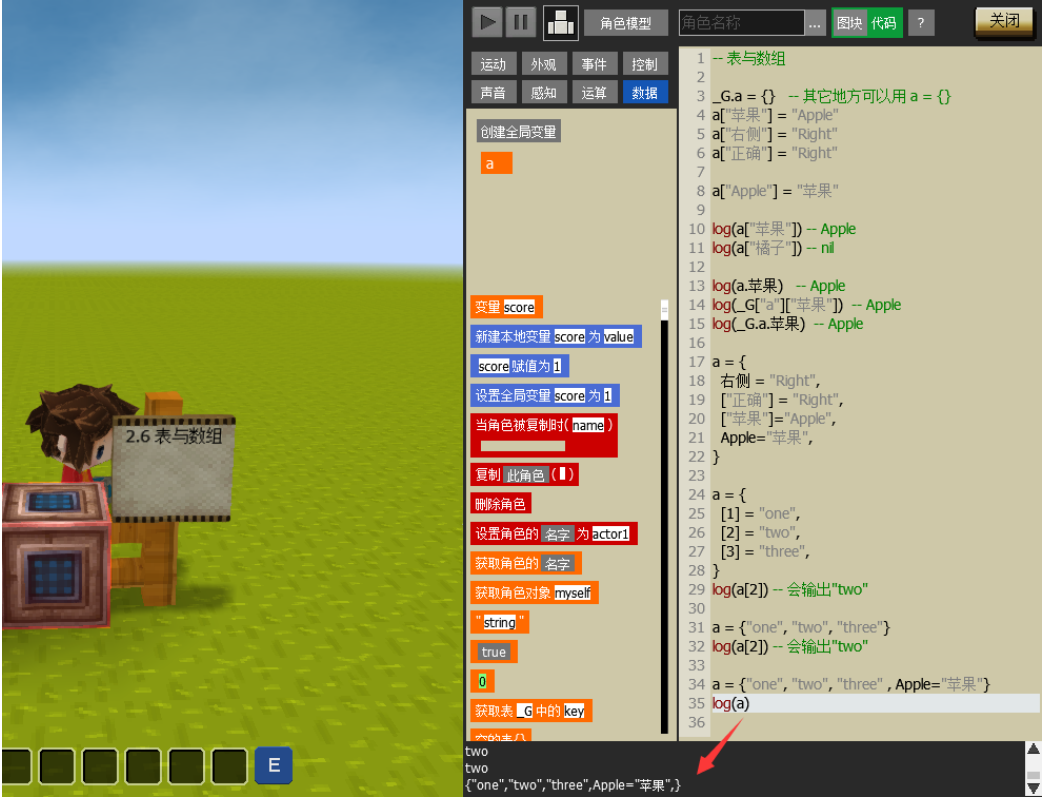
```
a = {"one", "two", "three"}
log(a[2]) -- 会输出"two"
```

这样的表，通常叫做数组。

我们也可以混合两种写法，例如：

```
a = {"one", "two", "three", Apple="苹果"}
log(a)
```

此时`log(a)`会输出整张表的内容。如下图：



了解了表和数组的概念，我们来看一个复杂一点的例子。

```
moveTo(19202, 5, 19168)
turnTo(90)
```

moveTo函数可以让人物瞬移到指定位置。它的输入是3个数字坐标， 关键字1,2,3映射的数据分别代表x,y,z的坐标。

```
-- moveTo(19202, 5, 19168)
local pos = {19202, 5, 19168}
moveTo(pos[1], pos[2], pos[3])

turnTo(90)
```

下面我们先注释掉moveTo这行， 换一种写法。先创建一个本地变量pos，再建立表中的3个映射，分别将1,2,3的位置映射到数据19202, 5, 19168。再输入`moveTo(pos[1], pos[2], pos[3])`。我们重新运行，可以看到人物瞬移到的位置是一样的。所以注释掉的代码和我们新加入的使用变量pos的代码效果是一样的。

下面我们看turnTo函数，它让演员转向到某个角度，代码中是90度。下面我们来用一个变量来记录角色的位置(pos)和方向(facing)

我们先注释掉之前的代码， 换一种写法。

```
local params = {}
params.pos = {x=19202, y=5, z=19168}
params.facing = 90

moveTo(params.pos.x, params.pos.y, params.pos.z)
turnTo(params.facing)
```

`local params = {}` 先创建一个本地变量params，让它指向一个空的表。
`params.pos = {x=19202, y=5, z=19168}` 再向params中加入pos到另外一张表，也就是包含x,y,z坐标的映射。
`params.facing = 90` 我们再加入一个字符串facing到90的映射。
然后我们再使用moveTo函数，将三个坐标 `params.pos.x, params.pos.y, params.pos.z` 输入给它；以及使用turnTo函数，将 `params.facing` 输入给它。

运行后，如上图，人物瞬移到19202, 5, 19168并转向90度的方向。虽然代码变长了， 但是我们将数据和命令通过变量分离开来了。 这样做有很多好处。比如，我们可以根据变量动态的计算人物的位置和方向，比如下图中的 `params.pos.y + 1` 和 `params.facing + 45` 。注意下面的代码我还用了一个前面讲到的更简洁的初始化表的写法。

```
local params = {
   pos = {x=19202, y=5, z=19168},
   facing = 90,
}
moveTo(params.pos.x, params.pos.y + 1, params.pos.z)
turnTo(params.facing + 45)
```

最后我们总结一下：表是一组有映射关系的数据的集合， 是NPL语言中唯一的复合型数据， 几乎所有复杂的概念都需要通过表来表示，例如一个3维坐标， 一组复杂的输入； 我们写的所有的代码其实都存在表中。 表可以通过关键字(一般是字符串和数字)快速的输出它对应的数据。

2.7.1 函数

这一节我们来讲解NPL语言中的最后一种数据类型: 函数function。

在2.2中，我们讲过程序的本质是输入和输出。而函数正是建立从输入到输出关系的方法。

其实，在之前的章节中，我们已经使用过很多系统函数例如log, moveTo, turnTo, 操作符例如 `+-*/ ..  == > <` 以及 `=` 也是函数的一种特殊形态。未来我们还会学到例如 `if, else, while, for` 等内置函数。

这一节，我们重点介绍如何定义新的函数。注意：计算机语言中的函数，和数学语言中常常提到的函数不是一个概念。前者有着更宽泛的含义。

函数是代码的主要形态，甚至可以说是唯一形态。可以说代码就是由函数构成的。如果说中文，英文是由文字和单词构成的， 那么函数就如同自然语言中的文字或单词。 只不过，自然语言中的词汇是单向串联起来的，并且文字和单词的总量是基本固定的。但是计算机语言不同，我们在解决一个问题时，需要定义许许多多新的词汇（也就是函数），再利用这些新的函数（也就是词汇）去描述某个领域中的输入和输出关系。

所以每个程序，都有大量仅属于自己的词汇。好的程序员能够更科学的定义这些词汇（也就是函数），让代码朗朗上口， 逻辑清晰，简洁而优美。 初级程序员由于不善于定义词汇，代码往往冗长，晦涩，不易阅读。

在NPL中，我们可以用关键字function去创建一个新的函数。例如：我们定义一个变量JumpForward（中文是向前跳的意思），将后面的整个函数（function）赋给它。

```
local JumpForward = function(distance, maxHeight)
   -- 这里是函数的内容
end
```

函数的定义是以`function()`开始，以`end`结束。`()`内为函数的输入，多个输入用`,`分开，每个输入需要指定一个局部变量名。上面的例子中就是distance（距离）和maxHeight（最大高度）。在函数被调用时这2个局部变量会被赋值。

函数的内部可以有任意其它的函数。每个代码方块或文件中的代码都在一个我们看不见的函数内部。当代码方块被激活或文件被加载时，函数里面的代码就会被执行。

函数还可以用return（返回）来输出一个结果（我们后面会讲解）。 函数本身和字符串，数字一样是一种数据类型，存在于一个固定的存储空间，如果你定义的函数没有任何变量指向它，则函数会被释放掉。 因此我们需要将新定义的函数，马上用`=号函数`赋给一个变量，未来我们就可以用这个变量来调用函数。

下面是一种更友善的定义函数的语法，和上面是完全等价的， 但是省去了等号。写法是这样的：

```
local function JumpForward(distance, maxHeight)
    -- 这里是函数的内容
end
```

内部是函数内容，也就是function和end之间的代码会在`JumpForward`这个函数被调用时执行。

无论哪种写法，其实`JumpForward`都是一个本地变量，当然我们也可以用全局变量，去掉上面代码中的local，例如这样：

```
function JumpForward(distance, maxHeight)
end
```

或者像这样：

```
JumpForward = function(distance, maxHeight)
end
```

对于代码方块， 全局变量需要存放在_G表中，像下面这样

```
function _G.JumpForward(distance, maxHeight)
end
-- 或者
_G.JumpForward = function(distance, maxHeight)
end
```

甚至，我们可以用多个不同的变量指向同一个函数，例如：

```
function JumpForward(distance, maxHeight)
end
local DoJump = JumpForward
```

但是通常我们都只用一个固定的变量名来指向一个函数。因此，这个变量名也被叫做函数名。

> 调用函数的语法为 `函数名(param1[, param2, ...])`

例如： `JumpForward(2, 1)`
我们之前使用到的系统函数moveTo，turnTo等都是用这种方式调用的。只不过这些系统函数是在系统代码中定义的，在你的代码运行前，已经被加载了。

下面我们来看一个完整的自定义函数并调用这个函数的例子。

```
local function JumpForward(distance, maxHeight)
    local time = distance / 5;
    move(distance, maxHeight, 0, time)
    move(distance, -maxHeight, 0, time)
end
```

distance，maxHeight是JumpForward函数内部的局部变量，当每次JumpForward函数被调用时，它们会代表不同的输入。在函数的内部我们调用了系统函数move两次。move函数有4个输入， 前3个是相对当前角色位置的x,y,z位移，第4个输入为消耗的时间。所以第一个move让人物向斜上放运动， 第二次move让人物向斜下方运动。运动的距离，高度，和用的时间由distance，maxHeight决定。

下面我们来调用JumpForward函数：

```
local function JumpForward(distance, maxHeight)
    local time = distance / 5;
    move(distance, maxHeight, 0, time)
    move(distance, -maxHeight, 0, time)
end

JumpForward(1, 1)
JumpForward(1.5, 1.5)
JumpForward(2, 2)
```

我们运行一下上面的代码，可以看到，我们通过调用自定义的JumpForward函数让人物向前跳跃了3次，每次的距离分别为1，1.5，2米。

我们看到函数隐藏了内部的输入和输出细节，并用一个变量名代替了内部看不见的逻辑关系。 好的程序员会为每一个功能写函数，并命名函数，使得代码的可读性大大增加， 并可以重复利用相同的逻辑关系。

> 写代码有一个黄金原则是`绝对不要写重复的代码` 。

写代码的过程中，程序员会不断的将重复的逻辑封装到函数变量中，这个过程叫做代码的`重构`。 宏观上看， 函数使得代码有了层级关系， 每个层级上仿佛都有程序员自己定义出来的一套新语言（和新词汇）。 一个复杂的程序可能会定义成千上万的函数变量。

我们再来看一个有返回值的函数，叫做平方函数。

```
local function sq(x)
   local result = x * x;
   return result
end
```

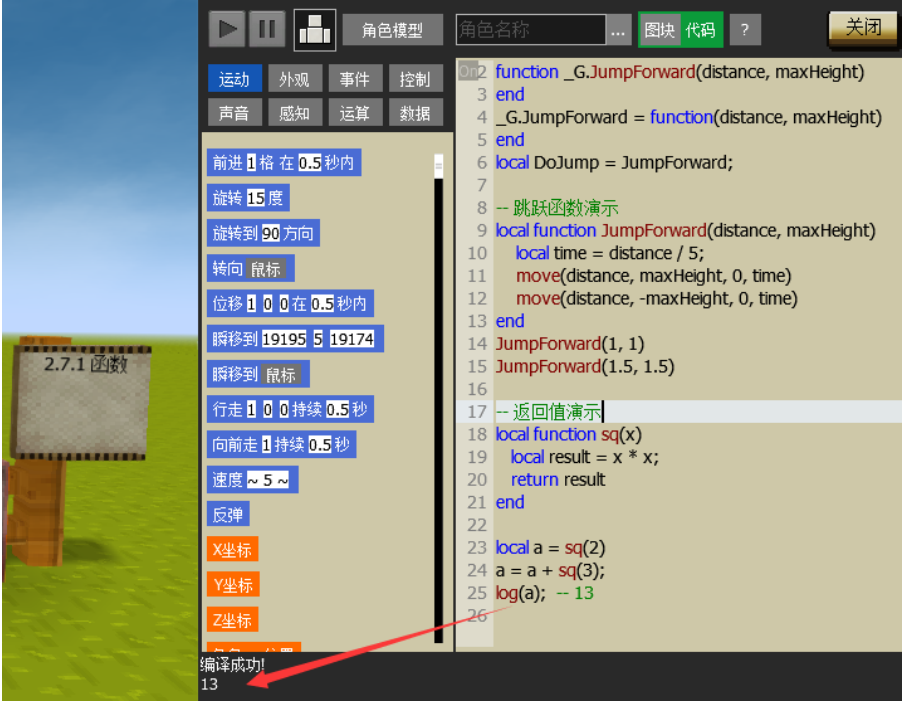
我们在函数内部定义了一个局部变量result。它的作用域是到end结束。这里result首先被赋值为x * x。
return函数代表了函数的输出，也就是sq(x)的输出。return函数后面的代码不会被执行。

下面我们来调用这个函数：

```
local function sq(x)
   local result = x * x;
   return result
end

local a = sq(2)
a = a + sq(3);
log(a);   -- 13
```

因为程序是按顺序执行的，执行到这里，此时a的值已经是4+9=13。我们用log函数输出，运行一下， 可以看到此时log(a)的输出为13。



下一节我们讲解一些常用的系统内置函数`for, if, while, and, or`。

2.7.2 内置函数

这一节我们来看一些常用的系统内置函数，包括`and, or, if, for, while`。这些内置函数与我们自己定义的函数本质是一样的，只是语法不同.
它们有一个共同的特点就是在一定条件下改变代码的执行路径，代码不再是顺序执行。

下面我们分别来看下。

先来看and（和）函数。

```
local result = (left) and (right)
```

它将代码分成了左右两个部分（left）and（right）。
它会先执行左侧的代码，如果左侧的代码的返回值为false或nil，则整个and函数返回左侧代码的输出，右侧代码不会执行。
如果左侧代码的返回值不是false或nil，则右侧的代码会执行，并且整个and函数返回右侧代码的输出。

我们看个例子

```
local function left_code(a)
   log("左侧执行了")
   return a > 10;
end
local function right_code(a)
   log("右侧执行了")
   return a > 5;
end
```

我们先来定义一个左侧函数left_code。这个函数会输出`左侧执行了`，它会返回一个值，如果输入大于10的话，它会返回true，否则会返回false。我们再来定义一个右侧函数(right_code)，它会输出`右侧执行了`。如果右侧输入大于5的话，它会返回true，否则返回false。现在我们来使用and函数：

```
local t = left_code(10) and right_code(10);
log(t); -- false
```

and函数左侧代码为left_code(10)，然后是and和right_code(10)。此时我们输出t，我们运行一下，可以看到执行的结果为`左侧执行了`。由于左侧10并不大于10，所以返回了false。因此整个and函数会返回左侧代码的执行结果，也就是t为false，而右侧代码并没有执行。

下面我们将左侧输入变成11，右侧输入为10不变。

```
local function left_code(a)
   log("左侧执行了")
   return a > 10;
end
local function right_code(a)
   log("右侧执行了")
   return a > 5;
end

local t = left_code(11) and right_code(10);
log(t); -- true
```

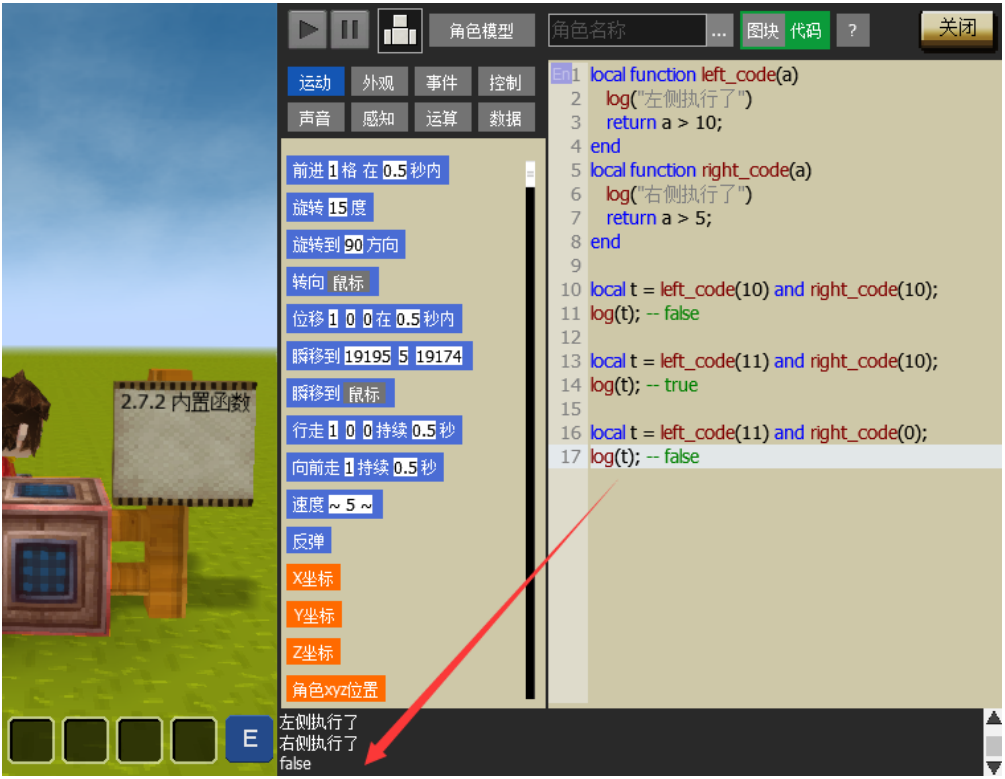
此时我们再次运行，我们可以看到左侧代码的输入11是大于10的，所以返回了true。左侧代码执行了，此时and函数会继续执行右侧的代码。因为右侧代码的输入是大于5的，所以右侧代码返回了true。因此整个and函数返回true。

其实and函数左侧的代码永远会执行，只不过根据它的返回值的不同，决定了是否执行右侧代码，进而决定整个and函数的返回值。

如果我们将右侧代码的输入改成0，我们再次执行。

```
local t = left_code(11) and right_code(0);
log(t); -- false
```

可以看到左侧代码和右侧代码都执行了，但是右侧代码的返回值为false，因为0没有大于5。 如图：



下面我们来看or（或）函数，它也是将代码分成左右两个部分，但它与and函数的执行结果基本相反。也就是如果左侧代码的返回值不是false或nil，则整个or函数返回左侧代码的输出，右侧代码不会执行。如果左侧代码的返回值是false或nil，则右侧代码会执行，并且整个or函数返回右侧代码的输出。

下面我们来看一个例子，同样还是这两个函数，我们这里改为了or函数。我们运行一下，可以看到左侧的输入，同样是10，左侧代码返回了false，因为是or函数，所以

右侧代码会执行。右侧的输入10大于5，右侧代码返回了true，所以左右代码都执行了，并且整个or函数返回了true。

```
local function left_code(a)
   log("左侧执行了")
   return a > 10;
end
local function right_code(a)
   log("右侧执行了")
   return a > 5;
end

local t = left_code(10) or right_code(10);
log(t); -- true
```

那么我们现在将左侧的输入改为11，右侧的输入改为0。

```
local t = left_code(11) or right_code(0);
log(t); -- true
```

再来运行一下。我们看到只有左侧的代码执行了，并且因为11大于10返回了true，而右侧的代码并没有执行。如图：

下面来看if函数。 if中文是如果的意思，它需要配合then和end来使用，也就是如果(if)那么(then)的意思，我们用一个例子来说明它的用法。

```
function testword(a)
     if (a=="hello") then
          log("a是hello")
     end
end
testword("hello")
testword("xxx")
```

我们先来定义一个函数textword，如果（if） a等于hello，也就是a和字符串hello完全相同；那没(then)我们输出`a是hello`。下面我们来调用testword函数两次，输入分别是hello和xxx。

```
function testword(a)
     if (a=="hello") then
          log("a是hello")
     end
end
testword("hello")
testword("xxx")
```

我们运行一下，可以看到它只输出了`a是hello`，也就是说if函数会根据括号中函数的返回值来决定是否会执行then和end之间的代码。
比如上面的代码中，如果a等于hello，那么中间的代码才会执行，log才有输出。否则输入是xxx那么log这行代码并不会执行。

我们同样还可以使用else关键字。else是否则的意思。我们在它后面加上`log("a不是hello")`。

```
function testword(a)
    if (a == "hello") then
        log("a是hello")
    else
        log("a不是hello")
    end
end
testword("hello") -- a是hello
testword("xxx") -- a不是hello
```

那么这段代码的意思是：如果a和字符串hello完全相同，则执行then和else之间的代码，否则将执行else和end之间的代码。

此时我们再运行，可以看到输出了两行：

- `testword("hello")` 输出了 `a是hello` 。
- `testword("xxx")` 输出了 `a不是hello` 。

我们看到if函数是编程语言中唯一一个有多种形态的特殊函数，它可以由多个像 `then end else` 这样的关键字构成。

比如它还可以加入elseif关键字，如下面：

```
function testword(a)
    if (a == "hello") then
        log("a是hello")
    elseif(a == "world") then
        log("a是world")
    else
        log("a不是hello, 也不是world")
    end
end
```

`elseif` (否则如果)a `==`等于等于 world，那么输出 `a是world` ；再用else关键字，也就是否则输出 `a不是hello，也不是world` 。也就是前两个括号中的函数返回false时才会执行最后一个else和end之间的代码。

整体来说，if函数中至少要有then和end，同时还可以有任意多个elseif和一个else。if函数最终的实现效果是依次执行括号中的代码，直到有一行代码返回真则执行后面的代码。换句话说，上述由关键字隔开的三段代码永远只有一段会执行。

下面我们再加一行 `testword("world")` ，运行一下，可以看到输出了三行结果。

```
function testword(a)
  if (a == "hello") then
     log("a是hello")
  elseif(a == "world") then
     log("a是world")
  else
     log("a不是hello, 也不是world")
  end
end
testword("hello")
testword("world")
testword("xxx")
```

- `testword("hello")` 输出了 `a是hello` 。
- `testword("world")` 输出了 `a是world` 。
- `testword("xxx")` 输出了 `A不是hello也不是word` 。

当然我们也可以不使用elseif，用两个if函数来写。例如在第一个if函数的else和end之间再加入另一个if函数，那么结果也是一样的。 如下：

```
function testword(a)
  if (a == "hello") then
     log("a是hello")
  else
     if(a == "world") then
          log("a是world")
     else
          log("a不是hello, 也不是world")
     end
  end
end
testword("hello")
testword("world")
testword("xxx")
```

为了避免嵌套，让逻辑更清晰，我们还是用第一种写法。if, then, elseif, else, end 是系统内置的 `关键字` ，他们可以共同的十分灵活的定义若干输入和输出之间的条件触发关系。if函数在计算机语言中十分的常见，但是它也会破坏代码的可读性。在自然语言中，例如我们在用中文讲课或写文章时，我们很少用：如果怎么样，那没怎么样。即使我们平时说话时使用了 `如果` ，在如果和那么之间的文字也不会很长，也很少嵌套。同样的原则对于计算机语言同样适用，我们应该尽可能的让我们的代码看上去是顺序执行的。

初级程序员的代码到处都是冗长和嵌套的if函数。下面我们介绍一些降低if函数复杂度的方法。

- 第一种方法是将then和end之间的代码放到一个新的函数中。
- 第二种方法是将各种输入和输出都放入一个table表中。

下面我们来看一个例子，首先我们先将then和end之间的代码放到一个函数中，这里我们需要创建三个新函数，它们分别 `a_is_hello, a_is_world, a_is_others` ，分别对应我们之前if,end中间的代码。在实际使用中，这里面的代码可能是很多行的。

```
local function a_is_hello()
   log("a是hello")
end
```

```
local function a_is_world()
   log("a是world")
end
local function a_is_others()
   log("a不是hello, 也不是world")
end
```

然后我们会创建一个table，比如叫wordtable，它建立了多个字符串和函数之间的对应关系，也就是字符串hello到a_is_hello这个函数的映射；以及字符串world到a_is_world函数的映射。如下面所示：

```
local function a_is_hello()
   log("a是hello")
end
local function a_is_world()
   log("a是world")
end
local function a_is_others()
   log("a不是hello, 也不是world")
end

local wordtable = {
   hello = a_is_hello,
   world = a_is_world,
}

function testword2(a)
     local result = wordtable[a] or a_is_others
     result()
end
```

这时我们再定义一个testword2函数，此时我们就可以避免出现if和end,这样来写：我们将对条件的判断改为对table对象的查询。那么如果没有查询到的话，我们则返回`a_is_others`变量。此时result是一个函数变量，我们用`result()`调用这个函数。

现在我们来测试一下。

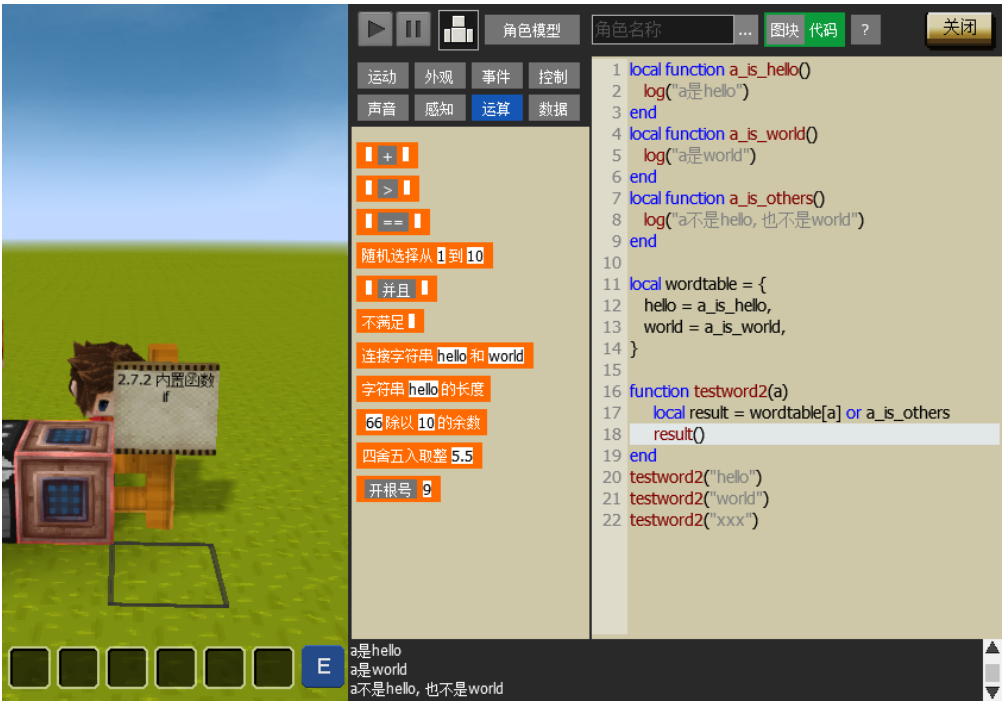
```
local function a_is_hello()
   log("a是hello")
end
local function a_is_world()
   log("a是world")
end
local function a_is_others()
   log("a不是hello, 也不是world")
end

local wordtable = {
```

```
    hello = a_is_hello,
    world = a_is_world,
}

function testword2(a)
     local result = wordtable[a] or a_is_others
     result()
end
testword2("hello")
testword2("world")
testword2("xxx")
```

调用testword2函数3次。第一次的输入为hello，第二次的输入为world，第三次的输入为xxx。运行一下，可以看到输出同样是这三个。通过这样的方法，我们避免了使用if函数。



下面我们来看while函数，它是一个循环函数，while是循环的意思，它同样需要配合do和end两个关键字来使用。我们来举个例子。

while函数会不停的执行while和end之间的代码， 直到（）中的代码输出不是false或nil。

```
local a=0
while(a<3) do
   a = a + 1;
   log(a)
end
```

如上面的代码中，第一次执行时a等于0。0小于3，括号中的代码返回true。所以它下面的代码继续执行。那么0+1等于1，log(a)输出结果是1，那么do和end之间的代码会执行三次，会有三个输出结果。
我们运行一下，可以看到输出结果为123，也就是当a大于3时，后面的代码将不再执行。

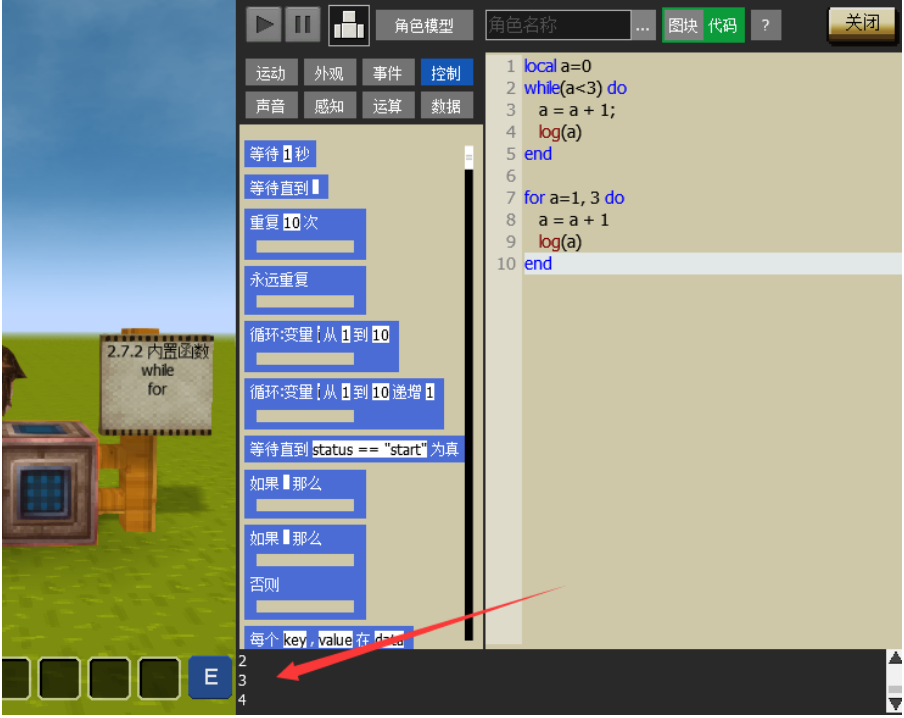
最后我们再来看另外一个循环函数for，它的语法是这样的。 我们用一个例子来说明：

```
for a = 1, 3, 1 do
   a = a + 1
   log(a)
end
```

for和while类似，只不过它会定义一个局部变量a并设置一个初始值1 一个结束值3和一个递增值1，并重复执行do和end之间的代码，也就是第一次执行时a等于1，然后a会不停的加1。 最后一次执行时a等于3，每次log(a)会输出不同的a的数值。我们运行一下，可以看到输出结果为234。当然如果递增的值是1的话，我们也可以不写，例如这样

```
for a = 1, 3 do
   a = a + 1
   log(a)
end
```

我们再运行，结果是一样的。



好了， 到今天为止， 我们就讲完了NPL语言中的全部语法。 无论多么复杂的程序都是由我们学到的这些最基本的函数构成的。 可见， 目前的高级计算机语言相比自然语言要简单很多， 一般只有十几个最基本的函数构成， 但是如何运用这些函数去创造成千上万更复杂的函数却需要大量的读写练习。NPL/Paracraft提供了一个很好的练习环境。你可以通过代码方块学习计算机编程， 最终编写出和Paracraft一样复杂的程序。

2.8 总结与对自学编程的建议

我们在前面的章节中通过Paracraft的代码方块中的例子，系统的学习了NPL语言的基本语法。这一章我们对学习编程语言做一个总结，并给出进一步深入自学编程的一些建议。

1. 语法
2. 程序的本质
3. 数字与数学
4. 变量与名字
5. 字符串与文字
6. 表与数组
7. 函数

上图是我们的大纲，编程的本质是掌握一门新的语言。计算机语言主要是描述各种输入和输出关系的，从而构建出各种有一定智能输入输出的软件。人脑也是由输入输出组成的智能系统。
那么人脑是如何学会编程的呢？其实人脑学习编程和学习任何知识一样，是一个创作的过程。人脑可以被动的将我们的感受和编写过的代码都记录下来，并且根据相似性自发的建立相互之间的连接。随着我们创造的代码量增多，我们的大脑内会形成一些关键的链接，如同是顿悟， 这些关键链接是真正的知识，它们大幅提升我们在编程中的创造力 ，当这些关键连接积累到一定数目，我们的大脑就对编程形成了一个完整的联通的知识体系。也就是说通过大量的编程练习和反思，我们在自己的大脑中创造了一个超级程序，它使得我们可以随心所欲的输出代码。
遗憾的是还没有程序员能描绘出知识在人脑中的具体连接形态，但一个经验丰富的程序员是可以在脑海中用关键词创作一个完整的编程体系的链接图，也叫做思维导图，是每个人脑海中知识的一个投影，也是唯一作为老师可以分享给学生的。
所以学习编程没有捷径，你需要不断的编写程序，使大脑不断形成新的记忆和连接，直到你也能够自信的完整的阐述什么是编程。 我们对编程的探索和阐述是没有止境的，只能尽可能接近真理，至少目前我们还无法写出具有人类学习能力的软件，我们还无法确定大脑中的知识是通过哪些变量，以及通过什么样的输入输出关系构成的。

中文，英文，计算机语言是现代科学工作者必备的3种语言。 前2种语言我们一生都在使用的创作工具。如果你未来5-10年，希望更多的使用计算机语言去创作，那么本节将给你若干学习编程语言的建议。

- 第一个建议是保持兴趣。 保持兴趣意味着你可以给自己设定一个又一个目标。能够不断的找到想到新的程序去编写，或者不断的升级和完善自己的作品。如果有一天你无法知道自己下一步想编写什么，可以去 `KeepWork.com` 上看看别人的作品。从小学到大学的学生阶段，虽然我们业余时间不多，但却是最自由的，这10年是最适合积累代码量的。
- 第二个建议是系统学习。系统学习的目的是构建尽可能完整的知识体系。知识越完整越不容易忘记。拥有完整知识的人更加自信，也更有创造力。所以当你完成了一些作品， 发现自己对编程越来越有兴趣时， 你需要抽时间系统的看3本书或理解相关知识。分别是：
  - (1) 计算机体系结构: 了解计算机的硬件构成和工作原理
  - (2) 操作系统： 了解比如linux,windows是如何编写出来的，甚至看下他们的源代码。
  - (3) 编译原理： 总有一天你会发明自己的编程语言，那么了解下编译原理，至少在你需要时，你不会认为这是一件过于困难的事情。

> 这里顺便向大家推荐一个叫LISP的语言，它很古老，发明于1960年， 后来一个分支成为面向过程和对象的语言（C/C++/C#/Java等）， 另一个分支成为了更

> 高级的类汇编语言。

最后，对于已经是程序员的读者，我想说程序员和科学家一样都是需要追求真理的人， 我们写的每个软件，都是在创造中探索着输入，输出间最简洁，高效的连接方式。好了， 以上就是NPL基础编程理论部分的全部教学内容，未来我们还会在keepwork.com上推出更多编程课程和实例。

3 计算机辅助设计CAD简介
4 计算机体系结构
[5 相似原理与人脑仿真]

6 对未来教育的思考

我7岁开始学习编程，到上大学时已经编写了数十个软件项目。大学期间出于长期对人工智能的兴趣和研究，我开始研发NPL语言和ParaEngine游戏引擎，一直到今天已经快15年了，写了过百万的代码和各种工具。

虽然我目睹了很多同龄人和我一样，以优异的成绩考入了大学。 但是回想起来，我儿童时代的学习经历是非常独特的，甚至是未来教育的一个重要参考。

目前我们的团队在用NPL语言做一件很重要的事情，就是建立人类未来教育的基础平台。

教育的本质

我最喜欢我父亲说的一句话：`教育就是让人保持一直有事可做`
现在想来，这句话无论对于小孩，还是成年人都是非常困难的。
让学生一直保持在做有意义的事情是未来教育的唯一目标。我很庆幸，自己从有记忆开始，就一直在这种状态中。

自学与基于项目的学习是最容易达成这个目标的教育形态。 常常看一些教育工作者在争论课堂学习的必要性，或批判STEAM等基于项目学习的科学性，在我看来，唯一评判的标准就是我父亲的那句话。

从我个人的经历看，包括我观察班级里学习最好的学生，他们很少认真听讲，而我自己更是用尽一切办法避免在课堂上浪费时间。

未来教育的形态一定是让像我这样喜欢自学和做自己的事情的学生更加如鱼得水，减少我们的痛苦。而我学生时代最大的痛苦就是感觉白天在浪费时间。

未来教育的形态

在我看来，未来教育并不难达成，甚至无需改变教育所用的教材，书籍。

我们要做的是培养兴趣，推荐和欣赏优秀的项目，用作品的评比代替考试，剩下的时间留给学生。 而教育工作者更多的时间是花在展现自己的知识和作品给周围的人。

整个教育的氛围应该同社会中的企业类似，充满着基于创新和作品的沟通,对完美的追求，对合作的渴望，对用户的责任，以及作品之间的攀比。

我们过去曾经用NPL语言开发过一个名叫魔法哈奇的3D网游，有500多万的注册用户，我们运营了9年，这是一个非常复杂的虚拟社会。用户是自愿加入的，每个用户拥有

自己的3D家园，可以和同伴一起战斗，获得好看的服装和宠物。未来教育的形态和我们开发的这个游戏也是有相似性的。

一切从动画与编程教育开始

未来教育其实是在构建一个全新的虚拟社会。这个虚拟社会有自己的规则，用户自愿加入，从开始的几个人，到500万人，到上亿人。

无需关注外面的人怎么看，加入的人会在里面生活5年，10年，甚至一辈子。
这个社区我们叫做keepwork。并且我们从动画与编程教育开始做，理由是

- 我们有自己研发15年的创作工具和编程语言的积累
- 动画和游戏是这个社区中作品的主题，大量的用户会很感兴趣
- 我7岁到20岁之间开发的软件也基本都是游戏。而我在用同样的方式学习其它学科，并同样有效。为了能用大量的业余时间开发我自己的游戏， 我不得不用最快的速度自学所有其它学科，同时精通英文。
- 人工智能时代，所有学科都需要计算机来辅助。这是一个最基础的学科也是一个工具，它可以将你学到的任意知识变成某种可分享的作品。

世界观

和游戏一样， 我们要做的未来教育的虚拟社会是有世界观的. 这个`世界观`我们把它称为`并行世界ParaWorld`。每个人的动画和游戏作品都呈现在一个开源的3D虚拟世界中，用户通过这些作品学习，交流，形成社会关系。

并行世界是我们全部渠道，平台，内容和工具的总称。 我们以NPL语言为圆心和半径，建立一个联通的编程教育文化圈。 如同中国文化的半径是中文，我们每天接触到的视频，文章，APP都是这个文化圈最外围的作品。 同理每个计算机语言，比如C/C++，Java也都有自己的原点和半径。

文化都是伴随语言的，我之所以能学会编程，是因为我最初的12年，一直呆在了微软的VC语言的生态圈中，我探索的过程就是在不断向圈的原点靠近，甚至有一天我萌生了创造属于自己的语言的想法。

我这样说其实是要反对市场上那些什么语言都教的教学平台。 那样的教育社区是没有共同语言的，也不可能像魔法哈奇那样成为一个默契的虚拟社会。 我不想罗列NPL语言的诸多好处。只是做为有近30年经验的程序员，我客观的觉得NPL语言拥有最适合教育同途的开发工具和语法，并且可以终身使用，它能够开发出像Paracraft，keepwork,魔法哈奇这样的复杂程序。

未来编程教育的要素

- 简单和强大并存的单一编程语言和创作工具。
- 个人网站创造工具
- 拥有海量开源软件项目的社区，并且所有项目都使用同一语言和工具。
- 能够记录和分享知识领域的个人知识引擎
- 能够基于项目撮合老师和学生的搜索引擎
- 能够评比所有用户作品的评估与认证机制

10年之后的编程教育

下面的大多数预言将在2035年前发生，有些其实已经在小范围内发生了。

> 预言1： 编程教育生态将以若干像Paracraft这样的工具和在线社区构成。

其实整个计算机领域都是无数这样的生态圈构成的，比如CAD软件，操作系统，数学仿真，游戏开发，动画电影制作，Web浏览器等。可惜的是目前每个领域的生态圈的原点都不在中国。 我们希望通过Paracraft/NPL构建教育领域的生态圈，并覆盖更广泛的年龄层，直到终生学习。

> 预言2： 中文，英文，计算机语言将成为人类教育的基础。 所有其它学科都将以它们为基础。

数学，物理，化学，生物，历史等等都是建立在人类语言的基础上的。 而计算机语言将像中文和英文一样大量的出现在其它学科的教材中。 甚至像数学，物理，化学，生物的主要教学语言都将被计算机语言取代。 其实在美国90%的小学生数学课已经全部数字化，这种趋势将逐渐发生在其它学科。也将深刻的改变这些学科的学习和考试的方式。

> 预言3： 编程将渗透到高考的各个学科中，而不是作为一个独立的学科。

编程作为独立学科是无法用传统的考试去评估的。 一个原因是它语言生态众多，人类每天都在发明新的计算机语言；另一个原因是语言本身太简单，成年人只需3个小时，就可以背下所有编程的知识点，并考取满分。 所以编程不太可能作为独立学科进入高考，即使会，也是暂时的。 但是其它学科就不一样了：数学，物理，化学等都会逐渐引入用计算机去解决问题的题目。 前提是要允许学生带着笔记本电脑参加高考。当然我们最希望看到的是，到那时，高考已经不是教育评估的主要方式，平时学习的过程和个人作品成为更重要的手段。

> 预言4： 个人作品将逐渐代替学分，成为教育评估的主要手段。

由于计算机技术的普及， 几乎所有学科的成果都可以通过数字化的计算机作品来表现， 并通过互联网来永久保存，和通过类似区块链的技术做认证。无论是大学招生，还是企业招人，将越来越信赖这种基于个人作品的评估方式。 在国外已经有一些政府项目试图用计算机记录学生从小学到大学的所有创造内容；并有包括MIT，哈弗在内的20多所高校加入了这个联盟，承认这种评估手段。 其实我们开发的Keepwork就是这样的系统：它用GIT（区块链）一样的格式去存储用户的作品，让用户可以方便的创建个人作品网站，并有大量的面向未来教育的辅助功能。

> 预言5： 老师将成为导师和终身学习者。

当AI取代了人类的大多数工作后， 每个人类都有更多的时间花在学习和育人上。 在终身学习的未来， 每个人都有机会成为别人的老师：每个父母都有更多的时间去教育自己的孩子，每个大学生会成为小学生的老师。

而全职老师的角色也将不仅是在课堂上教书，而更多的是作为某个领域的专家去启发学生。 同时自己的时间将更多的花在学习和创造上。 未来的师生关系将回到孔子那个时代， 成为一种重要的持续终身的社会关系或合作关系。

> 预言6： 基于项目的学习将取代基于教室的学习

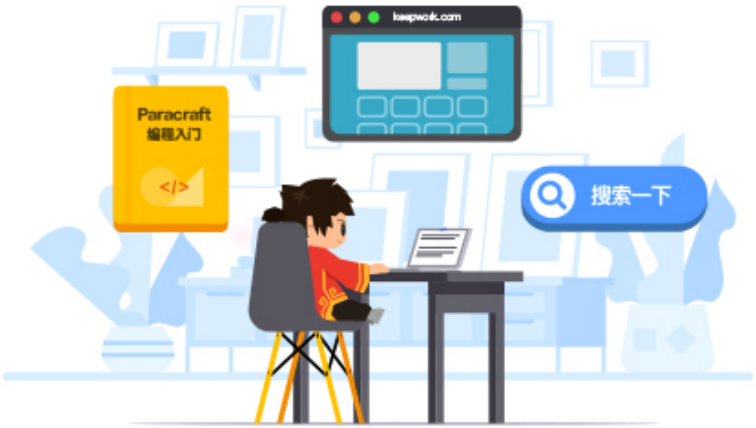
大多数优秀学生获取知识并不是在课堂的教室中， 而是在课外的自我体验中。基于45分钟一节课的教学模式将逐渐被更自主化的项目式学习取代。 在项目式学习中，学生有更多的时间去集中思考问题和解决问题。其实这才是更科学和更主流的学习任意学科的方式。探索，学习，创造是未来教育中，学生的主要行为；其中创造是占很大比例的。

> 预言7： 个人电脑是主要创作工具

编程学习需要使用人类最强大，最方便的输入输出设备。 虽然手机平板看上去更便捷，但是目前鼠标，键盘，大屏幕仍然是人类发明的最适合编程的设备或创作工具。有些消费类设备的进步，在人类历史上也都是昙花一现，比如触屏手机。 我很期待人类发明出更适合编程的输入，输出设备，但是绝不是目前的手机或平板。 作为教育工作者，应当让学生尽早的接触PC，目前手机阻断了很多青少年学习编程的道路。

> “不要记忆手册里的东西，因为它们都在软件或互联网中”

下部：参考手册



> 不要记忆手册里的东西，因为它们都在软件或互联网中

附录1：《NPL常用语法》速查表

NPL语言的语法100%兼容lua，并有扩充。下面仅包含本书理论部分提到的常用语法。

```
-- 两个横岗代表单行注释

--[[
    首尾加入两个 [ 和 ] 可以为多行注释
]]

----
-- 1. 变量与控制
----
local num = 42;  -- 所有的数字都是浮点数(double)

s = '相同的字符串在内存中只有一个Copy'
t = "双引号也可以"; -- `;` 每行代码的结尾可以加入`;`也可不加
u = [[ 在开头和结尾
       的两个中括号
       代表多行文本字符串.]]
-- 'nil' 为空的意思. 例子中t不再指向任何存储单元.
-- 当没有任何变量指向某个存储单元时，存储单元很快会被从内存中释放。
t = nil

-- do 和 end 之间的代码是一个代码区间
while num < 50 do
  num = num + 1
end

-- 如果:
if (num <= 40) then
  log('小于等于 40')
elseif ("string" == 40 or s) then
  log('任意类型的变量之间都可以比较');
elseif s ~= 'NPL' then
  log('if语句也可以不加前后的括号()')
else
  -- 变量默认为全局变量，代码方块中情况特殊
  thisIsGlobal = 5  -- 注意变量是区分大小写的

  -- 如何定义一个本地变量:
  local line = "这个变量只能在下个`end`或文件结束前使用"

  -- 连接2个字符串用 .. 函数:
  log("第一个字符串" .. line)
end

-- 从来没有赋值过的变量会返回nil
-- 下面代码并没有语法错误
foo = anUnknownVariable  -- 现在 foo == nil.

aBoolValue = false  -- 真true与假false
```

```lua
-- 对于if语句只有nil 和 false 是假的; 0 和 ''都是真!
if (not aBoolValue) then log('false') end

-- 'or' 和 'and' 函数返回最近的输入
ans = (aBoolValue and 'yes') or 'no'  --> 'no'

local nSum = 0;
for i = 1, 100 do  -- i包含1和100是本地变量
   nSum = nSum + i
end

-- 用 "100, 1, -1" 可以递减:
-- 区间的三个输入分别是 开始值, 结束值[, 递增增].
nSum = 0
for i = 100, 1, -1 do
   nSum = nSum + i
end

-- 另一种罕见的循环, 循环直到nSum == 0:
repeat
  nSum = nSum - 1
until nSum == 0

----
-- 2. 函数
----
function fib(n)
  if n < 2 then return 1 end
  return fib(n - 2) + fib(n - 1)
end

-- 函数的返回值, 函数调用, 和赋值语句都支持多个输入
-- 不匹配的输入, 值为nil;
-- 多出来的输入会被自动忽略.

x, y, z = 1, 2, 3, 4
-- 现在 x = 1, y = 2, z = 3, 但是 4 会被忽略.

function bar(a, b, c)
  log(string.format("%s %s %s", a, b or "b", c))
  return 4, 8, 15, 16, 23, 42
end

x, y = bar("NPL")  --> 输出 "NPL b nil"
-- 现在 x = 4, y = 8, 数字 15..42 被忽略.

-- 函数也是变量, 可以是全局或本地的.
-- 下面定义函数f的方式是等价的:
function f(x)
  return x * x
end
f = function (x)
  return x * x
end
```

```
-- 下面的方式也是等价的
local function g(x) return math.sin(x) end
local g; g  = function (x) return math.sin(x) end

-- 当函数只有一个输入时，也可以不加括号:
log 'hello'  -- 正确的语法.

----
-- 3. 表.
----
-- 表是NPL语言中唯一的复合数据结构;

-- 默认情况下关键字为字符串类型: "key1", "key2"
t = {key1 = 'value1', key2 = false}

-- 可以用.来引用表中的数据:
log(t.key1)  -- 输出 'value1'.
t.newKey = {}  -- 可在运行过程中可随时插入新的数据.
t.key2 = nil  -- 将key2从表中删除.

-- 任何不是nil的数据类型都可以为表的关键字
u = {['@!#'] = 'blabla', [{}] = 1982, [3.14] = 'pi'}
log(u[3.14])  -- 输出 "pi"


for key, val in pairs(u) do  -- 获取表中的每个数据.
  log(key, val)
end

-- _G 是一个特殊的表，里面有所有的全局变量.
_G.test = 1
log(test == 1)  -- 输出 "true"

-- 表可以当成列表或数组使用:

-- 默认为从1开始递增的整数为关键字:
v = {'value1', 'value2', 1.21, 'gigawatts'}
for i = 1, #v do  -- #v 代表v中整形关键字的数据的数目.
  log(v[i]) -- 注意数组从1开始，不是0
end

-- Have fun with NPL!
```

附录2：《代码方块》 函数速查表

详细的参考手册请到Paracraft软件或官方网站中寻找。你也可以在Paracraft中访问项目`530`，直接在3D世界中运行各种测试代码。

运动

前进 `1` 格 在 `0.5` 秒内: moveForward(1, 0.5)

```
-- 例子1:
turn(30);
for i=1, 20 do
    moveForward(0.05)
end
```

旋转 `15` 度: turn(15)

```
-- 例子1:
turnTo(-60)
for i=1, 100 do
    turn(-3)
end
-- 例子2:点击我打招呼
say("Click Me!", 2)
registerClickEvent(function()
    turn(15)
    play(0,1000)
    say("hi!")
end)
```

旋转到 `90` 方向: turnTo(90)

```
-- 例子1:
turnTo(-60)
wait(1)
turnTo(0)
-- 例子2:三轴旋转
turnTo(0, 0, 45)
wait(1)
turnTo(0, 45, 0)
wait(1)
turnTo(0, nil, 45)
-- 例子3:
while(true) do
    setActorValue("pitch", getActorValue("pitch")+2)
    say(getActorValue("pitch"))
    wait()
end
-- 例子4:
while(true) do
    turnTo(nil, nil, getActorValue("roll")+2)
    wait()
end
```

转向 `鼠标` : turnTo(“mouse-pointer”)

```
-- 例子1:转向鼠标,主角,指定角色
turnTo("mouse-pointer")
moveForward(1, 1)
turnTo("@p")
moveForward(1, 1)
turnTo("frog")
moveForward(1, 1)
-- 例子2:面向摄影机
while(true) do
    turnTo("camera")
    wait(0.01)
end
-- 例子3:面向摄影机
-- camera yaw and pitch
while(true) do
    turnTo("camera", "camera")
    wait(0.01)
end
```

位移 `1` `0` `0` 在 `0.5` 秒内: move(1, 0, 0, 0.5)

```
-- 例子1:
turnTo(0)
move(0.5,1,0, 0.5)
move(1,-1,0, 0.5)
say("jump!", 1)
```

瞬移到 `19178` `5` `19209` : moveTo(19178, 5, 19209)

```
-- 例子1:
moveTo(19257,5,19174)
moveTo("mouse-pointer")
moveTo("@p")
moveTo("frog")
```

瞬移到 `鼠标` : moveTo(“mouse-pointer”)

```
-- 例子1:瞬移到主角, 鼠标, 指定角色
say("current player", 1)
moveTo("@p")
say("mouse-pointer", 1)
moveTo("mouse-pointer")
say("the frog actor if any", 1)
```

```
moveTo("frog")
-- 例子2:瞬移到角色的某个骨骼
-- block position
moveTo("myActorName")
-- float position
moveTo("myActorName::")
-- bone position
moveTo("myActorName::bone_name")
```

行走`1 0 0`持续`0.5`秒: walk(1, 0, 0, 0.5)

```
-- 例子1:
walk(1,0) -- x,z
walk(0,1) -- x,z
walk(-1,0,-1) -- x,y,z
```

向前走`1`持续`0.5`秒: walkForward(1, 0.5)

```
-- 例子1:
turnTo(0)
walkForward(1)
turn(180)
walkForward(1, 0.5)
-- 例子2:恢复默认物理仿真
play(0,1000, true)
moveForward(1, 0.5)
walkForward(0)
```

速度`~ 5 ~`: velocity("~ 5 ~")

```
-- 例子1:
velocity("~ 10 ~")
wait(0.3)
velocity("add 2 ~ 2")
wait(2)
velocity("0 0 0")
```

反弹: bounce()

```
-- 例子1:遇到方块反弹
turnTo(45)
while(true) do
    moveForward(0.02)
    if(isTouching("block")) then
```

```
        bounce()
    end
end
```

X坐标: getX()

```
-- 例子1:
while(true) do
    say(getX())
end
```

Y坐标: getY()

```
-- 例子1:
while(true) do
    say(getY())
    if(getY()<3) then
        tip("Game Over!")
    end
end
```

Z坐标: getZ()

```
-- 例子1:
while(true) do
    say(getZ())
end
```

角色xyz位置: getPos()

```
-- 例子1:
local x, y, z = getPos()
setPos(x, y+0.5, z)
```

设置角色位置`19178` `5` `19209`: setPos(19178, 5, 19209)

```
-- 例子1:
local x, y, z = getPos()
setPos(x, y+0.5, z)
```

> 方向: getFacing()

```
-- 例子1:
while(true) do
    say(getFacing())
end
```

外观

> 说 `hello!` 持续 `2` 秒: say(“hello!”, 2)

```
-- 例子1:
say("Jump!", 2)
move(0,1,0)
-- 例子2:点击我打招呼
say("Click Me!", 2)
registerClickEvent(function()
    turn(15)
    play(0,1000)
    say("hi!")
end)
```

> 说 `hello!`: say(“hello!”)

```
-- 例子1:在人物头顶说些话
say("Hello!")
wait(1)
say("")
-- 例子2:点击我打招呼
say("Click Me!", 2)
registerClickEvent(function()
    turn(15)
    play(0,1000)
    say("hi!")
end)
```

> 提示文字`Start Game!`: tip(“Start Game!”)

```
-- 例子1:
tip("Start Game in 3!")
wait(1)
tip("Start Game in 2!")
wait(1)
tip("Start Game in 1!")
```

```
wait(1)
tip("")
```

显示: show()

```
-- 例子1:显示/隐藏角色
hide()
wait(1)
show()
```

隐藏: hide()

```
-- 例子1:显示/隐藏角色
hide()
wait(1)
show()
```

播放动作编号 `4`: anim(4)

```
-- 例子1:
anim(4)
move(-2,0,0,1)
anim(0)
-- 例子2:常用动作编号
-- 0: standing
-- 4: walking
-- 5: running
-- check movie block for more ids
```

播放从 `10` 到 `1000` 毫秒: play(10, 1000)

```
-- 例子1:播放电影方块中的角色动画
play(10, 1000)
say("No looping", 1)
-- 例子2:点击我打招呼
say("Click Me!", 2)
registerClickEvent(function()
    turn(15)
    play(0,1000)
    say("hi!")
end)
```

循环播放从 `10` 到 `1000` 毫秒: playLoop(10, 1000)

```
-- 例子1:播放电影方块中的角色动画
playLoop(10, 1000)
say("Looping", 3)
stop()
```

骨骼 `Root` 从 `10` 到 `1000` 并循环 `true`: playBone(“Root”, 10, 1000, true)

```
-- 例子1:
playBone("Neck", 2000)
-- regular expression supported
playBone(".*UpperArm", 5000, 7000)
playBone(".*Forearm", 5000, 7000)
play(0, 4000)
```

播放速度 `1`: playSpeed(1)

```
-- 例子1:
playSpeed(4)
playLoop(0, 1000)
say("Looping", 3)
playSpeed(1)
stop()
```

停止播放: stop()

```
-- 例子1:播放/暂停角色动画
playLoop(10, 1000)
wait(2)
stop()
turn(15)
playLoop(10, 1000)
wait(2)
stop()
```

放缩百分之 `10`: scale(10)

```
-- 例子1:
scale(50)
```

```
wait(1)
scale(-50)
```

放缩到百分之`100`：scaleTo(100)

```
-- 例子1:
for i=1, 20 do
    scale(10)
end
scaleTo(50)
wait(0.5)
scaleTo(200)
wait(0.5)
scaleTo(100)
```

观看`此角色`：focus(“myself”)

```
-- 例子1:
focus()
moveForward(2,2)
focus("player")
-- 例子2:
focus("someName")
focus(getActor("someName2"))
```

摄影机距离`12`角度`45`朝向`90`：camera(12, 45, 90)

```
-- 例子1:
for i=1, 100 do
    camera(10+i*0.1, nil, nil)
    wait(0.05)
end
```

放缩尺寸：getScale()

```
-- 例子1:
while(true) do
    if(getScale() >= 200) then
        scaleTo(100)
    else
        scale(10)
    end
end
```

动画时间: getPlayTime()

```
-- 例子1:
playLoop(10, 2000)
while(true) do
    if(getPlayTime() > 1000) then
        say("hi")
    else
        say("")
    end
    wait(0.01);
end
```

设置电影频道`myself`为:`0`,`0`,`0`: setMovie(“myself”, 0, 0, 0)

```
-- 例子1:不传参数代表与代码方块相邻的电影方块
hide()
setMovie("main")
playMovie("main", 0, -1);
-- 例子2:myself代表当前代码方块的名字
setMovie("myself")
playMovie("myself", 0, -1);
-- 例子3:指定电影方块的坐标
local x, y, z = codeblock:GetBlockPos();
setMovie("main", x, y, z+1)
playMovie("main", 0, -1);
```

播放电影频道`myself`从`0`到`-1`毫秒: playMovie(“myself”, 0, -1)

```
-- 例子1:播放与代码方块相邻的电影方块
hide()
-- -1 means end of movie
playMovie("myself", 0, -1);
stopMovie("myself");
```

循环播放电影频道`myself`从`0`到`-1`毫秒: playMovie(“myself”, 0, -1, true)

```
-- 例子1:播放与代码方块相邻的电影方块
hide()
playMovie("myself", 0, 1000, true);
```

停止播放电影频道`myself`: stopMovie(“myself”)

```
-- 例子1:
playMovie("myself", 0, -1);
stopMovie();
```

> 设置电影频道`myself`的属性`播放速度`为`1`:
> setMovieProperty(“myself”, “Speed”, 1)

```
-- 例子1:
setMovieProperty("myself", "Speed", 2);
playMovie("myself", 0, -1);
stopMovie();
```

事件

> 当演员被点击时``: registerClickEvent(function()
> end)

```
-- 例子1:
registerClickEvent(function()
    for i=1, 20 do
        scale(10)
    end
    for i=1, 20 do
        scale(-10)
    end
end)
```

> 当`空格`键按下时``: registerKeyPressedEvent(“space”, function(msg)
> end)

```
-- 例子1:空格跳跃
registerKeyPressedEvent("space",function()
    say("Jump!", 1)
    move(0,1,0, 0.5)
    move(0,-1,0, 0.5)
    walkForward(0)
end)
-- 例子2:任意按键
registerKeyPressedEvent("any", function(msg)
    run(function()
        say(msg.keyname)
    end)
    if(isKeyPressed("e")) then
```

```
        return true
    end
end)
-- 例子3:鼠标按钮
registerKeyPressedEvent("mouse_buttons",function(event)
    say("button:"..event:buttons())
end)
-- 例子4:鼠标滚轮
registerKeyPressedEvent("mouse_wheel",function(mouse_wheel)
    say("delta:"..mouse_wheel)
end)
```

当方块`10`被点击时``: registerBlockClickEvent(“10”, function(msg)
end)

```
-- 例子1:任意方块被点击
registerBlockClickEvent("any",function(msg)
        local blockid = msg.blockid;
        x, y, z, side = msg.x, msg.y, msg.z, msg.side
    say(blockid..":"..x..","..y..","..z..":"..side)
end)
-- 例子2:某个方块被点击
registerBlockClickEvent("10",function(msg)
        local blockid = msg.blockid;
        x, y, z, side = msg.x, msg.y, msg.z, msg.side
    tip("colorblock10:"..x..","..y..","..z..":"..side)
end)
```

当动画在`1000`帧时``: registerAnimationEvent(1000, function()
end)

```
-- 例子1:
registerAnimationEvent(10, function()
    say("anim started", 3)
end)
registerAnimationEvent(1000, function()
    say("anim stopped", 1)
end)
registerClickEvent(function()
    play(10, 1000)
end);
say("click me!")
```

当收到`msg1`消息时(`msg`)``: registerBroadcastEvent(“msg1”,
function(fromName)
end)

```
-- 例子1:
registerBroadcastEvent("jump", function(fromName)
    move(0,1,0)
    wait(1)
    move(0,-1,0)
end)
registerClickEvent(function()
    broadcastAndWait("jump")
    say("That was fun!", 2)
end)
say("click to jump!")
```

广播`msg1`消息: broadcast(“msg1”)

```
-- 例子1:
registerBroadcastEvent("hello", function(msg)
    say("hello"..msg)
    move(0,1,0, 0.5)
    move(0,-1,0, 0.5)
    say("bye")
end)
for i=1, 2 do
    broadcast("hello", i)
    wait(0.5)
end
```

广播消息`msg1`(``): broadcast(“msg1”, “”)

```
-- 例子1:
registerBroadcastEvent("hello", function(msg)
    say("hello"..msg)
    move(0,1,0, 0.5)
    move(0,-1,0, 0.5)
    say("bye")
end)
for i=1, 2 do
    broadcast("hello", i)
    wait(0.5)
end
```

广播`msg1`消息并等待返回: broadcastAndWait(“msg1”)

```
-- 例子1:
registerBroadcastEvent("hi", function(fromName)
    say("hi,"..tostring(fromName))
    wait(1)
```

```
    say("bye")
    wait(1)
end)
for i=1, 2 do
    broadcastAndWait("hi")
end
```

当代码方块停止时``: registerStopEvent(function()
end)

```
-- 例子1:只能执行马上可返回的代码
registerStopEvent(function()
    tip("stopped")
end)
```

当收到网络消息`connect`(`msg`)时``: registerNetworkEvent(“connect”, function(msg)
end)

```
-- 例子1:
registerNetworkEvent("updateScore", function(msg)
   _G[msg.userinfo.keepworkUsername] = msg.score;
   showVariable(msg.userinfo.keepworkUsername)
end)

registerNetworkEvent("connect", function(msg)
   broadcastNetworkEvent("updateScore", {score = 100})
end)

registerNetworkEvent("disconnect", function(msg)
   hideVariable(msg.userinfo.keepworkUsername)
end)

while(true) do
   broadcastNetworkEvent("updateScore", {score = 100})
   wait(1);
end
```

广播网络消息`score`(`{}`): broadcastNetworkEvent(“score”, {})

```
-- 例子1:
hide()
becomeAgent("@p")

registerNetworkEvent("updatePlayerPos", function(msg)
```

```
   runForActor(msg.userinfo.keepworkUsername, function()
      moveTo(msg.x, msg.y, msg.z)
   end)
end)

registerCloneEvent(function(name)
    setActorValue("name", name)
end)

registerNetworkEvent("connect", function(msg)
    clone(nil, msg.userinfo.keepworkUsername)
end)

registerNetworkEvent("disconnect", function(msg)
   runForActor(msg.userinfo.keepworkUsername, function()
      delete();
   end)
end)

while(true) do
   broadcastNetworkEvent("updatePlayerPos", {x = getX(), y=getY(), z=getZ()})
   wait(0.2);
end
```

发送网络消息给`username`,`title`,`{}`:
sendNetworkEvent(“usernames”, “title”, {})

```
-- 例子1:发送消息给指定用户
registerNetworkEvent("title", function(msg)
   tip(msg.userinfo.keepworkUsername)
   wait(1)
   tip(msg.a)
end)

sendNetworkEvent("username", "title", {a=1})
-- 例子2:发送原始消息给指定地址(无需登录)
-- __original is predefined name
registerNetworkEvent("__original", function(msg)
   log(msg.isUDP)
   log(msg.nid or msg.tid)
   log(msg.data)
end)

sendNetworkEvent(nid, nil, "binary \0 string")
-- given ip and port
sendNetworkEvent("\\\\192.168.0.1 8099", nil, "binary \0 string")
-- broadcast with subnet
sendNetworkEvent("\\\\192.168.0.255 8099", nil, "binary \0 string")
-- UDP broadcast
sendNetworkEvent("*8099", nil, "binary \0 string")
```

执行命令`/tip hello`: cmd("/tip hello")

```
-- 例子1:
cmd("/setblock ~0 ~0 ~1 62")
cmd("/cameradist 12")
cmd("/camerayaw 0")
cmd("/camerapitch 0.5")
-- 例子2:关闭自动等待
set("count", 1)
showVariable("count")
cmd("/autowait false")
for i=1, 10000 do
    _G.count = count +1
end
say("it finished instantly with autowait false", 3)
cmd("/autowait true")
for i=1, 10000 do
    _G.count = count +1
end
```

控制

等待`1`秒: wait(1)

```
-- 例子1:
say("hi")
wait(1)
say("bye", 1)
-- 例子2:等待下一个时钟周期
while(true) do
    if(isKeyPressed("space")) then
        say("space is pressed", 1)
    end
    wait()
end
```

等待直到``: repeat wait(0.01) until()

```
-- 例子1:每帧检测一次
say("press space key to continue")
repeat wait(0.01) until(isKeyPressed("space"))
say("started")
-- 例子2:输入为某个变量或表达式
repeat wait(0.01) until(gamestate == "gameStarted")
repeat wait(0.01) until(current_level == 1)
```

重复`10`次``: for i=1, 10 do
end

```
-- 例子1:
for i=1, 10 do
    moveForward(0.1)
end
```

永远重复``: while(true) do
end

```
-- 例子1:
while(true) do
    moveForward(0.01)
end
```

循环:变量`i`从`1`到`10```: for i=1, 10 do
end

```
-- 例子1:
for i=1, 10, 1 do
    moveForward(i)
end
```

循环:变量`i`从`1`到`10`递增`1```: for i=1, 10, 1 do
end

```
-- 例子1:
for i=1, 10, 1 do
    moveForward(i + 1)
end
```

等待直到`status == "start"`为真: repeat wait() until(status == “start”)

```
-- 例子1:
status = "gameStarted"
repeat wait() until(status == "gameStarted")
say("game started")
```

如果`那么`: if() then
end

```
-- 例子1:
```

如果`那么否则``: if() then
else
end

```
-- 例子1:
while(true) do
    if(distanceTo("mouse-pointer")<3) then
        say("mouse-pointer")
    else
        say("")
    end
    wait(0.01)
end
```

每个`key`,`value`在`data```: for key, value in pairs(data) do
end

```
-- 例子1:
myData = {
    key1="value1",
    key2="value2",
    key2="value2",
}
for k, v in pairs(myData) do
    say(v, 1);
end
```

每个`index`,`item`在数组`data```: for index, item in ipairs(data) do
end

```
-- 例子1:
myData = {
    {x=1, y=0, z=0, duration=0.5},
    {x=0, y=0, z=1, duration=0.5},
    {x=-1, y=0, z=-1, duration=1},
}
```

```
for i, item in ipairs(myData) do
    move(item.x, item.y, item.z, item.duration)
end
```

并行执行``: run(function()
end)

```
-- 例子1:
run(function()
    say("follow mouse pointer!")
    while(true) do
        if(distanceTo("mouse-pointer") < 7) then
            turnTo("mouse-pointer");
        elseif(distanceTo("@p") > 14) then
            moveTo("@p")
        end
    end
end)
run(function()
    while(true) do
        moveForward(0.02)
    end
end)
```

执行角色`myself`代码``: runForActor(“myself”, function()
end)

```
-- 例子1:
runForActor("myself", function()
        say("hello", 1)
end)
say("world", 1)
-- 例子2:
local actor = getActor("myself")
local x, y, z = runForActor(actor, function()
    return getPos();
end)
say(x..y..z, 1)
```

结束程序: exit()

```
-- 例子1:
say("Press X key to exit")
registerKeyPressedEvent("x", function()
    exit()
end)
```

> 重新开始: restart()

```
-- 例子1:
say("Press X key to restart")
registerKeyPressedEvent("x", function()
    restart()
end)
```

> 成为 当前玩家 的化身: becomeAgent("@p")

```
-- 例子1:成为当前角色的化身
becomeAgent("@p")
```

> 设置方块输出 15 : setOutput(15)

```
-- 例子1:
setOutput(15)
wait(2)
setOutput(0)
```

声音

> 播放音符 7 持续 0.25 节拍: playNote(“7”, 0.25)

```
-- 例子1:
while (true) do
    playNote("1", 0.5)
    playNote("2", 0.5)
    playNote("3", 0.5)
end
```

> 播放背景音乐 1 : playMusic(“1”)

```
-- 例子1:播放音乐后停止
playMusic("2")
wait(5)
playMusic()
```

播放声音 击碎 : playSound(“break”)

```
-- 例子1:播放音乐后停止
playSound("levelup")
-- 例子2:播放声道
playSound("channel1", "levelup")
wait(0.5)
playSound("channel1", "breath")
-- 例子3:一个声音同时播放多次
for i=1, 80 do
    -- at most 5 at the same time
    playSound("breath"..(i % 5), "breath")
    wait(0.1)
end
-- 例子4:音调和音量不同
for pitch = 0, 1, 0.1 do
   playSound("click", "click", 0, 1, pitch)
   wait(0.5)
end
for volume = 0, 1, 0.1 do
   playSound("click", nil, 0, volume, 1)
   wait(0.5)
end
```

暂停播放声音 击碎 : stopSound(“break”)

```
-- 例子1:
playSound("levelup")
wait(0.4)
stopSound("levelup")
-- 例子2:
playSound("levelup1", "levelup")
wait(0.5)
playSound("levelup2", "levelup")
wait(0.3)
stopSound("levelup2")
```

感知

是否碰到 方块 : isTouching(“block”)

```
-- 例子1:是否和方块与人物有接触
turnTo(45)
while(true) do
    moveForward(0.1)
    if(isTouching(62)) then
        say("grass block!", 1)
```

```
    elseif(isTouching("block")) then
        bounce()
    elseif(isTouching("box")) then
        bounce()
    end
end
-- 例子2:
local boxActor = getActor("box")
if(isTouching(boxActor)) then
    say("touched")
end
```

设置名字为`frog`：setActorValue(“name”, “frog”)

```
-- 例子1:复制的对象也可有不同的名字
registerCloneEvent(function(name)
    setActorValue("name", name)
    moveForward(1);
end)
registerClickEvent(function()
    local myname = getActorValue("name")
    say("my name is "..myname)
end)
setActorValue("name", "Default")
clone("myself", "Cloned")
say("click us!")
```

设置物理半径`0.25`：setActorValue(“physicsRadius”, 0.25)

```
-- 例子1:
cmd("/show boundingbox")
setBlock(getX(), getY()+2, getZ(), 62)
setActorValue("physicsRadius", 0.5)
setActorValue("physicsHeight", 2)
move(0, 0.2, 0)
if(isTouching("block")) then
    say("touched!", 1)
end
setBlock(getX(), getY()+2, getZ(), 0)
wait(2)
move(0, -0.2, 0)
cmd("/hide boundingbox")
```

设置物理高度`1`：setActorValue(“physicsHeight”, 1)

```
-- 例子1:
cmd("/show boundingbox")
```

```
setBlock(getX(), getY()+2, getZ(), 62)
setActorValue("physicsRadius", 0.5)
setActorValue("physicsHeight", 2)
move(0, 0.2, 0)
if(isTouching("block")) then
    say("touched!", 1)
end
setBlock(getX(), getY()+2, getZ(), 0)
wait(2)
move(0, -0.2, 0)
cmd("/hide boundingbox")
```

当碰到`name`时``: registerCollisionEvent(“name”, function(actor) end)

```
-- 例子1:某个角色
broadcastCollision()
registerCollisionEvent("frog", function(actor)
    local data = actor:GetActorValue("some_data")
end)
-- 例子2:任意角色
broadcastCollision()
registerCollisionEvent("", function(actor)
    local data = actor:GetActorValue("some_data")
    if(data == 1) then
        say("collide with 1")
    end
end)
-- 例子3:某个组Id
broadcastCollision()
setActorValue("groupId", 3);
registerCollisionEvent(3, function(actor)
    say("collide with group 3")
end)
```

广播碰撞消息: broadcastCollision()

```
-- 例子1:
broadcastCollision()
registerCollisionEvent("frog", function()
end)
```

到`鼠标`的距离: distanceTo(“mouse-pointer”)

```
-- 例子1:
while(true) do
```

```
    if(distanceTo("mouse-pointer") < 3) then
        say("mouse-pointer")
    elseif(distanceTo("@p") < 10) then
        say("player")
    elseif(distanceTo("@p") > 10) then
        say("nothing")
    end
    wait(0.01)
end
-- 例子2:
if(distanceTo(getActor("box")) < 3) then
    say("box")
end
```

计算物理碰撞距离`0`,`0`,`0`: calculatePushOut(0, 0, 0)

```
-- 例子1:保证不与刚体重叠
while(true) do
   local dx, dy, dz = calculatePushOut()
   if(dx~=0 or dy~=0 or dz~=0) then
      move(dx, dy, dz, 0.1);
   end
   wait()
end
-- 例子2:尝试移动一段距离
for i=1, 100 do
   local dx, dy, dz = calculatePushOut(0.1, 0, 0)
   if(dx~=0 or dy~=0 or dz~=0) then
      move(dx, dy, dz, 0.1);
   end
   wait()
end
```

提问`你叫什么名字?`并等待回答: local result = ask(“你叫什么名字?”)

```
-- 例子1:
ask("what is your name")
say("hello "..tostring(answer), 2)

ask("select your choice", {"choice A", "choice B"})
if(answer == 1) then
    say("you choose A")
elseif(answer == 2) then
    say("you choose B")
end
-- 例子2:
local name = ask("what is your name?")
say("hello "..tostring(name), 2)
-- 例子3:关闭对话框
run(function()
```

```
    wait(3)
    ask()
end)
ask("Please answer in 3 seconds")
say("hello "..tostring(answer), 2)
```

> 提问的结果: get(“answer”)

```
-- 例子1:
say("<div style='color:#ff0000'>Like A or B?</div>html are supported")
ask("type A or B")
if(answer == "A") then
    say("A is good", 2)
elseif(answer == "B") then
    say("B is fine", 2)
else
    say("i do not understand you", 2)
end
```

> 空格键是否按下: isKeyPressed(“space”)

```
-- 例子1:
say("press left/right key to move me!")
while(true) do
    if(isKeyPressed("left")) then
        move(0, 0.1)
        say("")
    elseif(isKeyPressed("right")) then
        move(0, -0.1)
        say("")
    end
    wait()
end
-- 例子2:
say("press any key to continue!")
while(true) do
    if(isKeyPressed("any")) then
        say("you pressed a key!", 2)
    end
    wait()
end
-- 例子3:按键列表
-- numpad0,numpad1,...,numpad9
```

> 鼠标是否按下: isMouseDown()

```
-- 例子1:点击任意位置传送
say("click anywhere")
while(true) do
    if(isMouseDown()) then
        moveTo("mouse-pointer")
        wait(0.3)
    end
end
```

> 鼠标选取: local x, y, z, blockid = mousePickBlock()

```
-- 例子1:点击任意位置传送
while(true) do
    local x, y, z, blockid, side = mousePickBlock();
    if(x) then
        say(format("%s %s %s :%d", x, y, z, blockid))
    end
end
```

> 获取方块`19178` `5` `19209`: getBlock(19178, 5, 19209)

```
-- 例子1:
local x,y,z = getPos();
local id = getBlock(x,y-1,z)
say("block below is "..id, 2)
-- 例子2:获取方块的Data数据
local x,y,z = getPos();
local id, data = getBlock(x,y-1,z)
-- 例子3:获取方块的Entity数据
local x,y,z = getPos();
local entity = getBlockEntity(x,y,z)
if(entity) then
    say(entity.class_name, 1)
    if(entity.class_name == "EntityBlockModel") then
        say(entity:GetModelFile())
    end
end
```

> 放置方块`19178` `5` `19209` `62`: setBlock(19178, 5, 19209, 62)

```
-- 例子1:
local x,y,z = getPos()
local id = getBlock(x,y+2,z)
setBlock(x,y+2,z, 62)
wait(1)
-- 0 to delete block
```

```
setBlock(x,y+2,z, 0)
setBlock(x,y+2,z, id)
```

> 计时器: getTimer()

```
-- 例子1:
resetTimer()
while(getTimer()<5) do
    moveForward(0.02)
end
```

> 重置计时器: resetTimer()

```
-- 例子1:
resetTimer()
while(getTimer()<2) do
    wait(0.5);
end
say("hi", 2)
```

> 设置为游戏模式: cmd("/mode game")

```

```

> 设置为编辑模式: cmd("/mode edit")

```

```

运算

> `+`: () + ()

```
-- 例子1:数字的加减乘除
say("1+1=?")
wait(1)
say(1+1)
```

> `>`: () > ()

```
-- 例子1:
if(3>1) then
   say("3>1 == true")
end
```

`==` : () == ()

```
-- 例子1:
if("1" == "1") then
   say("equal")
end
```

随机选择从`1`到`10`: math.random(1,10)

```
-- 例子1:
while(true) do
    say(math.random(1,100))
    wait(0.5)
end
```

`并且` : () and ()

```
-- 例子1:同时满足条件
while(true) do
    a = math.random(0,10)
    if(3<a and a<=6) then
        say("3<"..a.."<=6")
    else
        say(a)
    end
    wait(2)
end
```

不满足`` : (not)

```
-- 例子1:是否不为真
while(true) do
    a = math.random(0,10)
    if((not (3<=a)) or (not (a>6))) then
        say("3<"..a.."<=6")
    else
        say(a)
    end
```

```
    wait(2)
end
```

连接字符串`hello`和`world`：（“hello”⋯“world”）

```
-- 例子1:
say("hello ".."world".."!!!")
```

字符串`hello`的长度：（#“hello”）

```
-- 例子1:
say("length of hello is "..(#"hello"));
```

`66`除以`10`的余数：（66%10）

```
-- 例子1:
say("66%10=="..(66%10))
```

四舍五入取整`5.5`：math.floor(5.5+0.5)

```
-- 例子1:
while(true) do
    a = math.random(0,10) / 10
    b = math.floor(a+0.5)
    say(a.."=>"..b)
    wait(2)
end
```

开根号``9`：math.sqrt(9)

```
-- 例子1:
say("math.sqrt(9)=="..math.sqrt(9), 1)
say("math.cos(1)=="..math.cos(1), 1)
say("math.abs(-1)=="..math.abs(1), 1)
```

数据

变量`score`：score

```
-- 例子1:
local key = "value"
say(key, 1)
```

新建本地变量`score`为`value`: local key = “value”

```
-- 例子1:
local key = "value"
say(key, 1)
```

`score`赋值为`1`: score = 1

```
-- 例子1:
text = "hello"
say(text, 1)
```

设置全局变量`score`为`1`: set(“score”, “1”)

```
-- 例子1:也可以用_G.a
_G.a = _G.a or 1
while(true) do
    _G.a = a + 1
    set("a", get("a") + 1)
    say(a)
end
```

当角色被复制时(`name`)``: registerCloneEvent(function(name)
end)

```
-- 例子1:
registerCloneEvent(function(msg)
    move(msg or 1, 0, 0, 0.5)
    wait(1)
    delete()
end)
clone()
clone("myself", 2)
clone("myself", 3)
```

复制`此角色`(``): clone(“myself”)

```
-- 例子1:
registerClickEvent(function()
    move(1,0,0, 0.5)
end)
clone()
clone()
say("click")
```

删除角色: delete()

```
-- 例子1:
move(1,0)
say("Default actor will be deleted!", 1)
delete()
registerCloneEvent(function()
    say("This clone will be deleted!", 1)
    delete()
end)
for i=1, 100 do
    clone()
    wait(2)
end
```

设置角色的 名字 为 actor1 : setActorValue(“name”, “actor1”)

```
-- 例子1:
registerCloneEvent(function(name)
    setActorValue("name", name)
    moveForward(1);
end)
registerClickEvent(function()
    local myname = getActorValue("name")
    say("my name is "..myname)
end)
setActorValue("name", "Default")
setActorValue("color", "#ff0000")
clone("myself", "Cloned")
say("click us!")
-- 例子2:改变角色的电影方块
local pos = getActorValue("movieblockpos")
pos[3] = pos[3] + 1
setActorValue("movieblockpos", pos)
-- 例子3:改变电影角色
setActorValue("movieactor", 1)
setActorValue("movieactor", "name1")
-- 例子4:电影方块广告牌效果
local yaw, roll, pitch = getActorValue("billboarded")
setActorValue("billboarded", {yaw = true, roll = true, pitch = pitch});
setActorValue("billboarded", {yaw = true});
```

获取角色的`名字`: getActorValue(“name”)

```
-- 例子1:
registerCloneEvent(function(msg)
    setActorValue("name", msg.name)
    moveForward(msg.dist);
end)
registerClickEvent(function()
    local myname = getActorValue("name")
    say("my name is "..myname)
end)
setActorValue("name", "Default")
clone("myself", {name = "clone1", dist=1})
clone(nil, {name = "clone2", dist=2})
say("click us!")
```

获取角色对象`myself`: getActor(“myself”)

```
-- 例子1:
local actor = getActor("myself")
runForActor(actor, function()
        say("hello", 1)
end)
-- 例子2:
local actor = getActor("name1")
local data = actor:GetActorValue("some_data")
```

“`string`”: “string”

```
-- 例子1:
```

`true`: true

```
-- 例子1:
```

`0`: 0

```
-- 例子1:
```

获取表`_G`中的`key`: _G[“key”]

```
-- 例子1:
local t = {}
t[1] = "hello"
t["age"] = 10;
log(t)
```

空的表{}: {}

```
-- 例子1:
local t = {}
t[1] = "hello"
t["age"] = 10;
log(t)
```

新函数(`param`)``: function(param)
end

```
-- 例子1:
local thinkText = function(text)
        say(text.."...")
end
thinkText("Let me think");
```

调用函数`log`(`param`): log(param)

```
-- 例子1:
local thinkText = function(text)
        say(text.."...")
end
thinkText("Let me think");
```

调用函数并返回`log`(`param`): log(param)

```
-- 例子1:
local getHello = function()
        return "hello world"
end
say(getHello())
```

显示全局变量`score`: showVariable(“score”)

```
-- 例子1:
_G.score = 1
_G.msg = "hello"
showVariable("score", "Your Score")
showVariable("msg", "", "#ff0000")
while(true) do
   _G.score = _G.score + 1
   wait(0.01)
end
```

隐藏全局变量`score`: hideVariable(“score”)

```
-- 例子1:
_G.score = 1
showVariable("score")
wait(1);
hideVariable("score")
```

输出日志`hello`: log(“hello”)

```
-- 例子1:查看log.txt或F11看日志
log(123)
log("hello")
log({any="object"})
log("hello %s %d", "world", 1)
```

输出到聊天框`hello`: echo(“hello”)

```
-- 例子1:
echo(123)
echo("hello")
something = {any="object"}
echo(something)
```

引用文件`hello.npl`: include(“hello.npl”)

```
-- 例子1:文件需要放到当前世界目录下
-- _G.hello = function say("hello") end
include("hello.npl")
hello()
```

获取全局表`scores`：gettable(“scores”)

```
-- 例子1:
some_data = gettable("some_data")
some_data.b = "b"
say(some_data.b)
```

继承表`baseTable`,新表`newTable`：inherit(“baseTable”, “newTable”)

```
-- 例子1:
MyClassA = inherit(nil, "MyClassA");
function MyClassA:ctor()
end
function MyClassA:print(text)
    say("ClassA", 2)
end

MyClassB = inherit("MyClassA", "MyClassB");
function MyClassB:ctor()
end
function MyClassB:print()
    say("ClassB", 2)
end

-- class B inherits class A
MyClassB = gettable("MyClassB")
local b = MyClassB:new()
b:print()
b._super.print(b)
-- 例子2:
MyClassA = inherit(nil, gettable("MyClassA"));
function MyClassA:ctor()
end
function MyClassA:print(text)
    say("ClassA", 2)
end
local a = MyClassA:new()
a:print()
```

保存用户数据`name`为`value`：saveUserData(“name”, “value”)

```
-- 例子1:存储本地世界的用户数据
saveUserData("score", 1)
saveUserData("user", {a=1})
```

```lua
local score = loadUserData("score", 0)
assert(score == 1)
```

加载用户数据`name`默认值``: loadUserData(“name”, “”)

```lua
-- 例子1:
saveUserData("score", 1)
local score = loadUserData("score", 0)
assert(score == 1)
```

保存世界数据`name`为`value`: saveWorldData(“name”, “value”)

```lua
-- 例子1:常用于开发关卡编辑器
-- only saved to disk when Ctrl+S, otherwise memory only
saveWorldData("maxLevel", 1)
local maxLevel = loadWorldData("maxLevel")
assert(maxLevel == 1)
-- 例子2:从指定的文件加载
saveWorldData("monsterCount", 1, "level1")
local monsterCount = loadWorldData("monsterCount", 0, "level1")
assert(monsterCount == 1)
```

加载世界数据`name`默认值``: loadWorldData(“name”, “”)

```lua
-- 例子1:常用于开发关卡编辑器
-- only saved to disk when Ctrl+S, otherwise memory only
saveWorldData("maxLevel", 1)
local maxLevel = loadWorldData("maxLevel")
assert(maxLevel == 1)
-- 例子2:从指定的文件加载
saveWorldData("monsterCount", 1, "level1")
local monsterCount = loadWorldData("monsterCount", 0, "level1")
assert(monsterCount == 1)
```

附录3： 术语表

> 格式为：章节号 术语名称：定义或简单描述

1 **Paracraft**：Paracraft(创意空间)是一款免费开源的3D动画与编程创作软件。
1 **keepwork**： https://keepwork.com 是我们为Paracraft开发的一个学习平台。KeepWork是保持(keep)有事可做(work)或保存(keep)作品(work)的意思。
1.1 **BMAX**： bmax是一种Paracraft专用的基于粒子的3D静态模型文件格式。

2.1 **编译**：所有的计算机语言都需要被转换成这种底层硬件指令才能被执行，这个过程就是编译。
2.4 **变量**：程序中的变量只是某个存储单元的名字，它会直接输出程序执行的瞬间它所代表的存储单元。变量也可以看成是计算机语言的词汇。写代码的过程其实就是在不断创造新的词汇，并用这些词汇去描述我们要解决的问题。
2.4 **作用域**：变量是有生命周期的，变量的生命周期在程序中叫做作用域(Scope)。
2.5 **字符串**：字符串是一定长度的二进制数据。
2.5 **UTF8编码**：NPL中默认的编码规则叫做UTF8，UTF8是全世界使用最广泛的编码规则，几乎互联网上所有的文字都是这种编码。这种编码将每个英文字母或数字映射到一个Byte，将中文或其它特殊字符映射到2个或多个Byte。
2.6 **表**：表（Table）是数据到数据的映射关系。
2.6 **原表 metatable**：原表metatable是也是一种表，通过Metatable，程序员可以自定义[]和.函数针对某个表的输入/输出映射规则。更多内容可以搜索：`Lua metatable`
2.7.1 **函数**：函数是代码的主要形态，甚至可以说是唯一形态。可以说代码就是由函数构成的。 如果说中文，英文是由文字和单词构成的， 那么函数就如同自然语言中的文字或单词。 只不过，自然语言中的词汇是单向串联起来的，并且文字和单词的总量是基本固定的。
2.7.1 **重构**：写代码的过程中，程序员会不断的将重复的逻辑封装到函数中，这个过程叫做代码的重构。或者说重构是在程序功能基本不变的前提下，逐步优化代码的过程。
2.8 **LISP语言**：LISP是一种面向语言的语言。它很古老，发明于1960年， 后来一个分支成为面向过程和对象的语言（C/C++/C#/Java等）， 另一个分支成为了更高级的类汇编语言。

3 **CG**：计算机图形学(Computer Graphics or CG)是最激动人心且快速增长的现代技术之一。在早期的研究中，计算机图形学要解决的是如何在计算机中表示三维几何图形,以及如何利用计算机进行图形的生成、处理和显示的相关原理与算法，产生令人

赏心悦目的真实感图像。
3 **CAD**: 计算机辅助设计Computer Aided Design (CAD)。

4 **CPU**: 中央处理器 （英语：Central Processing Unit ，缩写： CPU ），是计算机的主要设备之一，功能主要是解释计算机指令以及处理计算机软件中的数据。计算机的可编程性主要是指对中央处理器的编程
4 **RAM**: 随机存取存储器（英语：Random Access Memory，缩写：RAM），也叫 主存，是与 CPU 直接交换数据的内部存储器。
4 **ROM**: 只读存储器（ Read-Only Memory，ROM ）是一种半导体存储器，其特性是一旦存储数据就无法再将之改变或删除，且内容不会因为电源关闭而消失。
4 **BIOS**: BIOS（Basic Input/Output System 的缩写、中文： 基本输入输出系统），BIOS 是 个人计算机启动时加载的第一个软件。
4 **操作系统**: 操作系统（英语：operating system，缩写作 OS）是管理计算机硬件与软件资源的系统软件，同时也是计算机系统的内核与基石。操作系统需要处理如管理与配置内存、决定系统资源供需的优先次序、控制输入与输出设备、操作网络与管理文件系统等基本事务。操作系统也提供一个让用户与系统交互的操作界面。

5 **AI**: 用代码写出具有人类智能的程序是人类最前沿的科研领域，也叫做人工智能Artificial Intelligence (AI)。
5 **NPL语言**: 神经元并行计算机语言Neural Parallel Language（简称NPL）。 NPL语言希望设计成为一种接近人类思维本质的新的计算机语言系统。它是面向未来人工智能时代的计算机语言系统。

6 **教育**: 教育就是让人保持一直有事可做。

附录4：如何学习Paracraft编程

Paracraft的教学理念可以用一个词来概括：KeepWork。 这里又包含了两个含义：

一是”keep working”， 反映了人的一种精神丰富的状态，是基于兴趣在做事的，永远不会是无聊的状态。

一是”keep your works”，即人要有作品。Keepwork网站可以让人们去创建，完成并展示自己的作品。

第一个是**教育的输入**，即学习应该是基于兴趣的，第二个是**教育的输出**，即要能创造出作品。我们认为教育的目标应该是培养富有生活情趣有创造力可以创造出美的作品的人，两方面相辅相成有机融合才是拥有全面素质的人。

Paracraft的编程学习与教学也是基于这两层含义来进行的。

首先，我们的学习和教学是**玩中学，做中学**。

玩中学

即玩他人制作的作品，包括观看他人制作的动画电影，玩他人制作的游戏。在学生学习Paracraft编程的第一阶段，我们鼓励学生大量的接触他人的作品，通过玩来探索Paracraft 3D世界。

- 老师可以带领学生每节课玩一个项目，然后在课堂上跟着老师完成动画的制作或者打完相应的代码（低年龄的小朋友可以使用可拖拽的条块）。在这些过程中，老师不需要讲多少理论的知识，因为这里最重要的是要让孩子多动手。
- 课后鼓励学生在家里多玩Paracaft的游戏或者观看动画。在Paracraft的客户端的官方推荐作品里，有我们推荐的优秀作品。

做中学

做中学，就是在做项目的过程中学习。

在学生开始的第一阶段，我们主要是玩中学，玩很多其他人制作的项目，并跟着老师一起完成一些小项目。到了第二阶段，这里我们就要自己开始做项目了。当然我们也是从比较简单的小项目开始。

做项目就是要有自己的作品，并参加Paracraft的创意大赛。

在玩和做项目中，有经验的老师会懂得如何适时的引导学生去观察，去探索，让学生从各个侧面去对整个知识体系逐渐的建立一些认识。

因为Paracraft对孩子有极强的吸引力，可以制作多样的甚至复杂的3D作品，小孩可以通过玩他人的作品来进行丰富的探索，同时也特别渴望可以创作出自己的3D作品。教师一定要把这些因素有机的运用起来。课程可以教一定的知识，但更多的知识是需要学生自己去探索的。玩和做项目给了他们最丰富的第一手的经验，他们大多会自己去总结一些规律的。这时候老师也要善于引导学生去更好的总结他们的经验，帮助他们建立起自己的知识模型。

这就涉及到下面的看书和反思。

看书和反思

除了玩和做项目，平时的看书和反思对于知识体系的建立必不可少。这本书里有系统的理论知识，教师可以引导学生平时注意阅读相关的章节，善于引导学生基于自身的体验进行反思，逐步建立起自己的系统知识。

下面具体讲讲玩中学阶段的课程怎么上比较好。

玩中学的课程如何上

- 以小游戏或动画吸引人，每节课都有个小游戏或动画，小孩需要完成个任务。
- 因为有游戏或动画，小孩愿意吃些苦头去学习，包括打完代码。有些比较长的代码，小孩打起来是比较累的，但是大多愿意打完。
- 在每节课中穿插了一些我们对编程的重要体验。因为课程的趣味性，小孩愿意去听这些，学习程序员是怎么编程的。
- 积累了足够的感性经验后，开始逐步引入系统性的知识，但仍然引导学生从自己的感性经验去总结，实践中发现小孩是有一定的总结能力的。这可以说是相似性原理的运用。

升级：复杂项目的学习

本书中的50个项目一开始的都比较简单，属于引入性质的玩中学课程/项目，学生多玩多动手，充分增加感性经验。后面的项目则会比较复杂，比如BlockBot，人力资源游戏，MindMaster，坦克大战等等。在大家完成前面的课程/项目，可以自己动手设计实现小的项目后，可以通过学习这些比较复杂的项目学会设计大型的游戏。这些复杂项目里，我们已经把如何面对复杂项目进行抽象建模的方法针对每个项目都做了阐述，但没有像前面的小的游戏一样细节到每个指令。学生需要通过自己的自主学习，

去参看该游戏世界里的代码进行学习。老师也可以做一定的引导，包括让学生改动或增加游戏的某个部分。

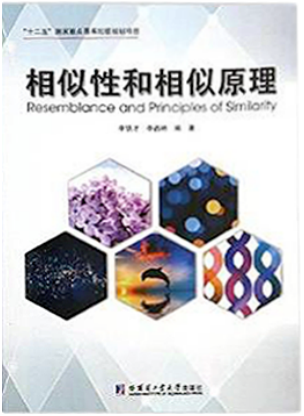
学习编程的前置条件

几岁的小孩适合开始学习编程？从我们的线下课实践发现，还是7岁以上会比较合适。这里有几个基本技能条件：

- 一定程度上掌握键盘和鼠标操作，比如鼠标左右键，键盘的ctrl,shift，上下左右键，回车键。Paracraft涉及大量鼠标和键盘上这些键的使用，哪怕只是做动画。其他的键，如回退，删除，以及26个字母和一些常用符号（+,-,=等），如果小孩熟悉这些键的位置会好些，不熟悉现找问题也不大。如果小孩还不会键盘和鼠标操作，比如来上课的一些7岁以下的小孩，那就先需要教一下鼠标和键盘的使用，再通过动画搭场景或者玩小游戏来熟练键盘和鼠标的操作，也不是大问题，小孩基本很快就能够掌握好。
- 认识简单的中文，因为使用条块的拖拽式编程需要能识别需要的条块。这些中文比较简单，大概二年级的都问题不大。我们也有一年级的小朋友，还不太会认字，但也在努力的找哪个条块是老师演示里用的（同时也算是在学习认字了）。
- 认识简单的英文。使用条块的拖拽式编程对英文的需求不大，偶尔会用到英文的是变量的名字是英文的，这个跟着老师取同样的名字就好。但我们对学过英文的小孩要求他们尽量直接打代码编程而不是使用拖拽式条块。
- 逻辑思维能力等。确实有些小孩似乎更喜欢编程更擅长编程，有些小孩更喜欢搭建场景制作动画。即使喜欢编程的小孩里也有不喜欢数字和计算的小孩。人的出发点是多元的。不应该太过黑白分明的认为那些擅长学习编程的小孩才应该学习编程。在一个paracraft项目里需要不同类型的人参与，paracraft本身的特点也让不同类型的人都能找到自己的兴趣点，同时在同一个环境同一个项目中通过做自己感兴趣的比较大的事情接触到其他方面的思维模式，这样一个开放的状态有助于他们将来在兴趣自然转移时（其实是很自然天然的过程）能够比较顺利的转入另一个领域的学习，成为全面素质的人才。

附录5：推荐书目

- 《相似性与相似原理》： 本书作者的另外一本书，是Paracraft的理论基础。



- 《Paracraft创意动画入门》：于平老师编写的专注3D动画创作的教科书。