



NetLogo 4.0.2 用户手册

(简体中文版)

翻译：张发

Richter2000@163.com

审校：王军

2008 年 10 月 16 日

目 录

译者说明.....	1
我为什么要翻译NetLogo用户手册？	1
读者的法律责任.....	1
联系方式.....	1
致谢.....	1
词汇对照表.....	2
NetLogo简介（What is NetLogo?）	3
产品特性.....	3
版权信息.....	5
Copyright Information	6
第三方许可证（Third party licenses）	6
更新历史（What's New?）	13
4.0.2 版 (2007 年 12 月).....	13
4.0 版 (2007 年 9 月).....	13
3.1.5 版 (2007 年 12 月).....	18
3.1 版 (2006 年 4 月).....	18
3.0 版 (2005 年 9 月).....	18
2.1 版 (2004 年 12 月).....	19
2.0.2 版(2004 年 8 月).....	19
2.0 版(2003 年 12 月).....	19
1.3 版 (2003 年 6 月).....	19
1.2 版 (2003 年 3 月).....	20
1.1 版 (2002 年 7 月).....	20
1.0 版(2002 年 4 月).....	20
系统需求（System Requirements）	21
系统需求： 应用程序.....	21
系统需求： 保存Applets	21
系统需求： 3 维视图.....	21
已知问题（Known Issues）	23
已知bug (所有系统).....	23
Windows上的 bug	23
Macintosh上的bug.....	24
Linux/UNIX上的 bug	24
计算机HubNet的已知问题	24
联系我们（Contacting Us）	25
网站.....	25
反馈、问题等.....	25
报告Bug.....	25
模型实例： 聚会（Sample Model:Party）	26
聚会.....	26
挑战.....	28
用模型思考.....	28

下一步?	28
教学# 1: 模型 (Tutorial #1 :Models)	30
模型实例: 狼吃羊 (Wolf Sheep Predation)	30
控制模型: 按钮.....	31
控制速度: 速度滑动条.....	31
调整设置: 滑动条和开关.....	32
收集信息: 绘图和监视器.....	34
控制视图.....	34
模型库.....	38
下一步?.....	39
教学# 2: 命令 (Tutorial #2:Commands)	40
模型实例: 基本交通模型(Traffic Basic)	40
命令中心 (The Command Center)	40
操纵颜色.....	43
主体监视器 (Agent Monitors) 和主体命令器 (Agent Commanders)	44
下一步?.....	47
教学 #3: 例程 (Tutorial #3:Procedures)	48
主体和例程.....	48
制作setup 按钮	48
制作go 按钮.....	50
试试命令.....	52
瓦片和变量.....	52
海龟变量.....	54
监视器 (Monitors)	56
开关和标签 (Switches and labels)	58
更多例程.....	60
画图 (Plotting)	62
时钟计数器 (Tick counter)	64
更多细节.....	65
下一步?.....	66
附录: 完整代码.....	67
界面指南 (Interface Guide)	70
菜单.....	70
标签页.....	72
界面页.....	73
信息页.....	81
例程页.....	83
Includes 菜单	84
编程指南 (Programming Guide)	86
主体(Agents).....	86
例程(Procedures)	87
变量(Variables)	89
颜色(Colors)	90
请求(Ask)	92

主体集合(Agentsets)	94
种类(Breeds).....	96
按钮(Button)	98
列表 (Lists)	99
数学 (Math)	103
随机数 (Random Numbers)	105
海龟图形 (Turtle shapes)	106
链图形 (Link Shapes)	106
时钟计数器 (Tick Counter)	106
视图更新 (View Updates)	107
绘图 (Plotting)	109
字符串 (Strings)	112
输出 (Output)	113
文件输入输出 (File I/O)	113
电影 (Movies)	115
视角 (Perspective)	116
绘画 (Drawing)	116
拓扑 (Topology)	117
链 (Links)	121
并发请求 (Ask-Concurrent)	123
捆绑 (Tie)	125
多个源文件 (Multiple source files)	125
语法 (Syntax)	125
迁移指南 (Transition Guide)	129
从NetLogo 3.1 迁移	129
从NetLogo 3.0 迁移	136
NetLogo词典 (NetLogo Dictionary)	138
分类 (Categories)	138
内建变量 (Built-In Variables)	141
关键词 (Keywords)	142
常量 (Constants)	142
A.....	143
B.....	148
C.....	152
D.....	160
E.....	164
F	168
G.....	177
H.....	177
I	183
J	191
L	191
M	201
N.....	212

O.....	216
P	220
R.....	229
S	239
T	251
U.....	258
V.....	262
W.....	262
X.....	267
Y.....	268
?.....	268
Java小程序 (Applets)	269
制作和显示Applets	269
Java需求	269
扩展.....	270
已知问题.....	270
图形编辑器指南 (Shapes Editor Guide)	271
开始.....	271
创建和编辑海龟图形.....	274
创建和编辑链图形.....	276
在模型里使用图形.....	277
行为空间指南 (BehaviorSpace Guide)	278
什么是BehaviorSpace?.....	278
怎样使用.....	279
高级用法.....	282
结论.....	284
系统动力学指南 (System Dynamics Guide)	285
NetLogo系统动力学建模工具是什么?.....	285
怎样使用.....	286
教学：狼吃羊(Wolf-Sheep Predation).....	289
结论.....	295
HubNet指南 (HubNet Guide)	296
理解HubNet.....	296
计算机HubNet.....	297
计算器HubNet.....	300
教师研讨会.....	300
HubNet 编程指南	300
获得帮助.....	300
HubNet编程指南 (HubNet Authoring Guide)	301
一般信息.....	301
HubNet活动编程	301
计算器HubNet信息	305
计算机HubNet 信息	305
日志 (Logging)	307

开始记录.....	307
使用日志工具.....	307
高级配置.....	311
控制指南 (Controlling Guide)	312
为NetLogo启动Java虚拟机	312
例子 (有GUI)	313
例子 (headless)	314
行为空间 (BehaviorSpace)	315
其他选项.....	315
结论.....	316
Mathematica 连接 (Mathematica Link)	317
它是什么?.....	317
我能用它做什么?.....	317
使用NetLogo-Mathematica 连接	317
安装.....	318
已知问题.....	319
致谢.....	319
扩展指南 (Extensions Guide)	320
使用扩展.....	320
编写扩展.....	321
数组与表扩展 (Array and Table Extensions)	327
何时使用.....	327
如何使用.....	327
数组例子.....	328
哈希表例子.....	328
已知问题.....	328
数组原语.....	329
哈希表原语.....	329
声音扩展 (Sound Extension)	332
使用Sound 扩展	332
MIDI支持	332
原语.....	332
声音名称.....	335
GoGo扩展 (GoGo Extension)	339
GoGo板 (GoGo Board) 是什么?.....	339
怎样得到 GoGo板?.....	339
安装GoGo 扩展.....	339
使用 GoGo 扩展.....	340
原语.....	340
性能剖析扩展 (Profiler Extension)	345
告诫.....	345
使用.....	345
原语.....	345
常见问题解答(Frequently Asked Questions).....	348

问题.....	348
一般问题.....	351
下载问题.....	354
Java 小程序 (Applet) 问题.....	356
运行问题.....	357
使用问题.....	359
编程问题.....	362
行为空间问题.....	367
扩展问题.....	368

译者说明

我为什么要翻译 NetLogo 用户手册？

这几年我对复杂系统很感兴趣，了解、使用过一些复杂系统仿真工具。平心而论，NetLogo 并不是特别强大，但与其他工具相比非常容易使用。对于许多从事复杂系统研究的人来说，用它作为一个工具搞点研究是比较省事的。

以前我并没有要翻译 NetLogo 学习资料的想法。我本来认为做学术研究的人读点软件文档不成问题，而不做学术研究的人也用不着学习 NetLogo。后来我发现情况并非如此，有的学生使用 NetLogo 做东西，向我抱怨说英文帮助看着费劲，因此影响了研究进展。我想也许这是事实，中国人看中文总比看英文容易点吧。

因此本项目就是让那些时间宝贵，看英文不是那么顺畅的人学习 NetLogo 使用的。当然如果是从事学术研究的人，我的忠告是：还是要多看英文！

读者的法律责任

任何人可以用任何方式阅读、打印、复制、传播本翻译作品，不需向译者支付任何有形或无形的报酬。

任何人不得以任何方式将本翻译作品用于商业目的。

联系方式

如果你发现译文有错误或不当之处，望不吝赐教。

我的电子邮件：Richter2000@163.com

致谢

这项不打粮食的工作之所以得以进行，需要衷心感谢以下人员：

- (1) 我的一个好朋友让我萌生了启动这项工作的想法（虽非直接，但确有关系）。
- (2) 感谢电视节目制作人员，他们那些充斥荧屏的不太吸引人的作品，让我能够放弃每天晚上 2-3 个小时的电视时间，用来从事这项工作，心里也不是那么痛苦。
- (3) 感谢我的父母，他们赐给我一个基本够用的脑袋，尤其是脑袋里那副质量过硬的牙齿。当我感到难以继续时，有牙可咬，还不至于咬坏！

词汇对照表

英 文	中 文
agent	主体
agentset	主体集合
breed	种类
diffuse	扩散
directed link	有向链
drawing	画图
extensions	扩展
follow	跟随
heading	方向
histogram	直方图
import	输入
interface	界面
item	项
label	标签
layout	布局
list	列表
link	链
movie	电影
neighbors	邻居
observer	观察者
report	返回
patch	瓦片
pen	画笔
plot	绘图
procedure	例程
seed	种子
shape	图形
tick	滴答
tie	捆绑
turtle	海龟
timer	计时器
undirected link	无向链
wrap	回绕

NetLogo 简介 (What is NetLogo?)

NetLogo 是一个用来对自然和社会现象进行仿真的可编程建模环境。它是由 Uri Wilensky 在 1999 年发起的，由连接学习和计算机建模中心 (CCL) 负责持续开发。

NetLogo 特别适合对随时间演化的复杂系统进行建模。建模人员能够向成百上千的独立运行的“主体”(agent)发出指令。这就使得探究微观层面上的个体行为与宏观模式之间的联系成为可能，这些宏观模式是由许多个体之间的交互涌现出来的。

NetLogo 可以让学生运行仿真并参与其中，探究不同条件下他们的行为。它也是一个编程环境，学生、教师和课程开发人员可以创建自己的模型。**NetLogo** 足够简单，学生和教师可以非常容易的进行仿真，或者创建自己的模型。并且它也足够先进，在许多领域都可以做为一个强大的研究工具。

NetLogo 有详尽的文档和教学材料。它还带着一个模型库，库中包含许多已经写好的仿真模型，可以直接使用也可修改。这些仿真模型覆盖自然和社会科学的许多领域，包括生物和医学，物理和化学，数学和计算机科学，以及经济学和社会心理学等。几个用 **NetLogo** 实现的基于模型的探究性课程正在开发。

NetLogo 提供了一个课堂参与式仿真工具，称为HubNet。通过联网计算机或者一些如TI图形计算器这样的手持设备，每个学生可以控制仿真模型中的一个主体。详情见[链接](#)。

NetLogo 是一系列源自 StarLogo 的多主体建模语言的下一代。它基于我们的产品 StarLogoT，增加了许多显著的新特征，重新设计了语言和用户界面。**NetLogo** 是用 Java 实现的，因此可以在所有主流平台上运行 (Mac,Windows,Linux 等)。它作为一个独立应用程序运行。模型也可以作为 Java Applets 在浏览器中运行。

产品特性

你可以通过下面列表了解 **NetLogo** 的特点和所提供的功能。

系统

跨平台:

可以在 Mac,Windows,Linux 等平台运行

语言:

完全可编程

简单语言结构

对 Logo 语言进行扩展支持主体

移动主体（海龟）在由静态主体（瓦片）组成的网格上移动
主体之间可以创建链接，形成聚集、网络和图
内置大量原语
双精度浮点数（IEEE 754）
运行过程在不同平台上完全可复现

环境:

用 2 维或 3 维模式查看模型
可伸缩、可旋转矢量图形
海龟和瓦片标签
可以进行运行中（on-the-fly）交互的命令中心
界面构建，包括按钮、滑动条、开关、选择器、监视器、文本框、注解、输出区
快进滑动条使你可以对模型进行快进和慢放
强大灵活的绘图系统
信息页用来解释模型
HubNet: 使用联网设备进行参与式仿真
主体监视器用来监视和控制主体
输出输入功能（输出数据，保存、恢复模型状态，制作电影）
行为空间（BehaviorSpace）工具用来从多次运行中收集数据。
系统动力学建模

Web

模型可以存为 applet 嵌入 web 页（注释：有些功能 applets 不能使用，例如有些扩展和 3 维视图）

版权信息

Copyright 1999–2007 by Uri Wilensky。保留所有权利。

NetLogo 软件、模型和文档免费提供给公众，用于探究、构建模型。允许对 NetLogo 软件、模型、文档进行复制、修改用于教育科研目的，不需支付任何费用，但前提是所有拷贝和支持文档中必须附带此版权信息和原作者姓名。对本软件其他形式的使用，不管是原始还是修改版本，包括但不限于整体或部分分发，必须首先获得 Uri Wilensky 的特别允许。在没有得到 Uri Wilensky 相应的授权时，不得使用、重写、修改本软件用做商业软件或硬件产品的基础。我们不对软件的适用性做任何承诺，也没有任何明确的或暗含的保证。

在学术出版物上引用本软件的方式是：Wilensky, U. (1999). NetLogo.
<http://ccl.northwestern.edu/netlogo/>. Center for Connected Learning and
Computer-Based Modeling, Northwestern University, Evanston, IL.

本项目衷心感谢美国国家科学基金的支持（(REPP 和 ROLE 项目) - 基金编号 REC #9814682 和 REC #0126227。

* 声明 *

本页为 NetLogo 4.0.2 User Manual 中 Copyright Information 的简体中文翻译，仅供读者粗略了解相关法律要求。准确的版权声明见本文档所附 Copyright Information。译者不对此译文的准确性承担法律责任。

Copyright Information

Copyright 1999-2007 by Uri Wilensky. All rights reserved.

The NetLogo software, models and documentation are distributed free of charge for use by the public to explore and construct models. Permission to copy or modify the NetLogo software, models and documentation for educational and research purposes only and without fee is hereby granted, provided that this copyright notice and the original author's name appears on all copies and supporting documentation. For any other uses of this software, in original or modified form, including but not limited to distribution in whole or in part, specific prior permission must be obtained from Uri Wilensky. The software, models and documentation shall not be used, rewritten, or adapted as the basis of a commercial software or hardware product without first obtaining appropriate licenses from Uri Wilensky. We make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

To reference this software in academic publications, please use: Wilensky, U. (1999). NetLogo. <http://ccl.northwestern.edu/netlogo/>. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

The project gratefully acknowledges the support of the National Science Foundation (REPP and ROLE Programs) -- grant numbers REC #9814682 and REC #0126227.

第三方许可证 (Third party licenses)

MersenneTwisterFast

NetLogo 使用 Sean Luke 的 MersenneTwisterFast 类作为随机数发生器。这些代码的版权信息如下：

Copyright (c) 2003 by Sean Luke.

Portions copyright (c) 1993 by Michael Lecuyer.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of the copyright owners, their employers, nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Colt

NetLogo 部 分 (特 别 是 random-gamma 原 语) 基 于 Colt 库 (<http://hoschek.home.cern.ch/hoschek/colt/>) 的代码。这些代码的版权信息如下：
Copyright 1999 CERN – European Organization for Nuclear Research. Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. CERN makes no representations about the suitability of this software for any purpose. It is provided "as is" without expressed or implied warranty.

MRJ Adapter

NetLogo 使用 MRJ Adapter 库， 版权是 Copyright (c) 2003–2005 Steve Roy sroy@roydesign.net 。 该 库 受 Artistic License 保 护 ， <http://homepage.mac.com/sroy/artisticlicense.html> 。 MRJ Adapter 可 从 <http://homepage.mac.com/sroy/mrjadapter/> 得到。

Quaqua

NetLogo 使用 Quaqua 观感库， 版权是 Copyright (c) 2003–2005 Werner Randelshofer, <http://www.randelshofer.ch/>, werner.randelshofer@bluewin.ch, All Rights Reserved. 该库遵从 GNU LGPL (Lesser General Public License)。该许可文件在“docs”文件夹，与 NetLogo 一起下载，也可从 <http://www.gnu.org/copyleft/lesser.html> 得到。

JHotDraw

系统动力学建模工具使用JHotDraw库，版权是Copyright (c) 1996, 1997 by IFA Informatik and Erich Gamma。该库遵从GNU LGPL (Lesser General Public License)。该许可文件在“docs”文件夹，与NetLogo一起下载，也可从<http://www.gnu.org/copyleft/lesser.html>得到。

MovieEncoder

NetLogo 使用节选自 Sean Luke 的 sim.util.media.MovieEncoder.java 的代码制作电影。该代码遵从 MASON Open Source License，版权信息如下：

This software is Copyright 2003 by Sean Luke. Portions Copyright 2003 by Gabriel Catalin Balan, Liviu Panait, Sean Paus, and Dan Kuebrich. All Rights Reserved.

Developed in Conjunction with the George Mason University Center for Social Complexity

By using the source code, binary code files, or related data included in this distribution, you agree to the following terms of usage for this software distribution. All but a few source code files in this distribution fall under this license; the exceptions contain open source licenses embedded in the source code files themselves. In this license the Authors means the Copyright Holders listed above, and the license itself is Copyright 2003 by Sean Luke.

The Authors hereby grant you a world-wide, royalty-free, non-exclusive license, subject to third party intellectual property claims:

to use, reproduce, modify, display, perform, sublicense and distribute all or any portion of the source code or binary form of this software or related data with or without modifications, or as part of a larger work; and under patents now or hereafter owned or controlled by the Authors, to make, have made, use and sell (“Utilize”) all or any portion of the source code or binary form of this software or related data, but solely to the extent that any such patent is reasonably necessary to enable you to Utilize all or any portion of the source code or binary form of this software or related data, and not to any greater extent that may be necessary to Utilize further modifications or combinations.

In return you agree to the following conditions:

If you redistribute all or any portion of the source code of this software or related data, it must retain the above copyright notice and this license and disclaimer. If you redistribute all or any portion of this code in binary form, you must include the above copyright notice and this license and disclaimer in the documentation and/or other materials provided with the distribution, and must indicate the use of this software in a prominent, publically accessible location

of the larger work. You must not use the Authors's names to endorse or promote products derived from this software without the specific prior written permission of the Authors.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS, NOR THEIR EMPLOYERS, NOR GEORGE MASON UNIVERSITY, BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

JpegImagesToMovie

NetLogo 使用节选自 Sun 公司的 JpegImagesToMovie.java 制作电影。这些代码的版权信息如下：

Copyright (c) 1999–2001 Sun Microsystems, Inc. All Rights Reserved.

Sun grants you ("Licensee") a non-exclusive, royalty free, license to use, modify and redistribute this software in source and binary code form, provided that i) this copyright notice and license appear on all copies of the software; and ii) Licensee does not utilize the software in a manner which is disparaging to Sun.

This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This software is not designed or intended for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. Licensee represents and warrants that it will not use or redistribute the Software for such purposes.

JOGL

NetLogo 使用 JOGL, Java API for OpenGL, 做图形渲染。关于 JOGL 的更多信息, 见 <http://jogl.dev.java.net/>。该库遵从 BSD license:

Copyright (c) 2003–2006 Sun Microsystems, Inc. All Rights Reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

– Redistribution of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

– Redistribution in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of Sun Microsystems, Inc. or the names of contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN MICROSYSTEMS, INC. ("SUN") AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You acknowledge that this software is not designed or intended for use in the design, construction, operation or maintenance of any nuclear facility.

Matrix3D

NetLogo 使用 Matrix3D 类进行 3D 矩阵操作。它的许可证如下：

Copyright (c) 1994-1996 Sun Microsystems, Inc. All Rights Reserved.

Sun grants you ("Licensee") a non-exclusive, royalty free, license to use, modify and redistribute this software in source and binary code form, provided that i) this copyright notice and license appear on all copies of the software; and ii) Licensee does not utilize the software in a manner which is disparaging to Sun.

This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED

OF THE POSSIBILITY OF SUCH DAMAGES.

This software is not designed or intended for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. Licensee represents and warrants that it will not use or redistribute the Software for such purposes.

ASM

NetLogo 使用 ASM 库产生 Java 字节码。它的许可证如下:

Copyright (c) 2000–2005 INRIA, France Telecom. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Log4j

NetLogo 使用 Log4j 库实现日志功能。该库的版权和许可证如下:

Copyright 1999–2005 The Apache Software Foundation

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed

under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

更新历史 (What's New?)

用户反馈对我们设计和改进NetLogo非常有价值。我们希望听取你的意见。请把评论、建议和问题发送到 feedback@ccl.northwestern.edu，Bug报告发送到 bugs@ccl.northwestern.edu

4.0.2 版 (2007 年 12 月)

- 文档:
 - 用户手册进行了许多小的更正和改进
- 模型:
 - 新的演化模型: Bug Hunt Coevolution
 - 改进模型: Climate Change (现在已校核), GasLab Atmosphere (更正), Red Queen, Bug Hunt Camouflage
- 引擎更正:
 - 更正链死亡 bug (只影响有多个链种类的模型)
 - 主体集合的sort-by，相等的主体采用随机排序
 - 当端点重合时，link-heading 报错
 - 更正了画笔颜色与import-world 不兼容的bug
 - 更正了性能剖析扩展中当海龟死亡时有时出现荒谬的结果
- 用户界面更正:
 - 在 Linux 上 3D 视图现在又可以用了
 - 现在当打开或保存 3.1 之前的模型时，给出警告。因为 4.0 不后向兼容，也不总会前向兼容
 - 更正了一个 bug，当最小最大值改变时滑动条不能总是保持其值在范围之内。
 - 更正了滑动条相关的几个 bug，他们引起 Java 例外或不期望的结果
 - 更正了那些有长代码和许多滑动条的模型编译很慢的 bug。
 - 更正了在 HubNet 客户编辑器不能删除界面元素或编辑开关的 bug
 - 改进了 3D 视图中点划线的外观
 - 改进了 applets 与一些浏览器和操作系统的兼容性

4.0 版 (2007 年 9 月)

- 模型:
 - 新的地球科学模型: Continental Divide, Climate Change
 - 新的化学模型 1: Diprotic Acid
 - 新的材料科学模型 1: Solid Diffusion
 - 新的数学模型: PANDA BEAR Solo, Surface Walking 2D
 - 新的网络模型: Team Assembly

- 新的计算机科学模型: Hex Cell Aggregation, Particle System Basic, Particle System Fountain, Particle System Waterfall, Particle System Flame
- 新游戏: Planarity
- 新的社会科学模型: Language Change, El Farol (老的 El Farol 模型现在叫 El Farol Network Congestion)
- 新的 NIELS 电磁模型: Ohm's Law, Series Circuit, Parallel Circuit
- 一套新的 Urban Suite 课程模型
- 一套新的 Connected Chemistry 课程模型
- 一套新的 BEAGLE Evolution 课程模型
- 改进和校核的模型: Dice Stalagmite, Autumn, Conic Sections 2, Echo, Rebellion, Daisyworld, Sound Machines, Birthdays, Bug Hunt Speeds, Electrostatics
- 其他改进模型: Small Worlds (更正), 多数 CA 1D models (更正), Star Fractal (检查代码), Genetic Drift T Interact (增加了可选墙), Flocking (平滑动画), Planarity (代码简化), Mimicry (代码简化), GasLab Circular Particles (代码清晰, 增加图), acid/base models ((代码简化))
- 新的代码实例: Random Grid Walk Example, Link Lattice Example, Lattice-Walking Turtles Example, Link-Walking Turtles Example, Intersecting Links Example, State Machine Example, Breed Procedures Example, Link Breeds Example, Mouse Drag Multiple Example, Hill Climbing Example, Rolling Plot Example, Ask-Concurrent Example, Ask Ordering Example, Random Network Example, Fully Connected Network Example, Mobile Aggregation Example, Wall Following Example, Circular Path Example, Profiler Example
- 改进代码实例: Halo Example (简化 tie 的使用), Intersecting Lines Example (更正), RGB and HSB Example (现在演示 RGB 列表), File Output Example
- 新的 HubNet 活动: PANDA BEAR
- 新的 HubNet 代码实例: Template
- 改进 HubNet 活动: Dice Stalagmite HubNet (校核), Bug Hunters Camouflage (校核), Root Beer Game (校核), Disease Doctors (校核), Minority Game
- 文档:
 - 在用户手册新增迁移指南, 帮助早期模型在 4.0 中能运行
 - 在用户手册的编程指南新增语法部分
 - 在用户手册新增 Applets 部分
- 功能:
 - 大多数模型运行的更快:
 - 改进程度随模型不同而不同, 一般快了约 1.5 倍
 - 加速原因是改进了编译器, 现在部分 NetLogo 代码编译为 Java 字节码
 - 链成为除海龟、瓦片之外的另一个主体类型, 这对网络模型、几何模型等有用
 - 新增链图形编辑器, 用来控制链的形状
 - 日志帮助研究者记录学生的行为, 供以后分析
 - 内置滴答计数器保持模型时间 (见下面的语言变化)

- 新的视图更新系统:
 - 现在有两种视图更新模式，基于滴答或连续
 - 启动 NetLogo 后默认是连续模式；模型库中的多数模型是基于滴答模式；连续模式对不基于滴答的模型，例如 Termites，有用，但也对调试有帮助
 - 基于滴答更新的模型多数运行较快，因为可以避免显示中间状态
 - 按钮不再有"force view update"勾选项；多数模型应使用 tick 和或 display
- 改进速度滑动条:
 - 能使用滑动条“快进”模型（通过更少的更新视图）
 - 在基于滴答的更新模式中，两个滴答之间有个暂停而不显示中间状态
 - 在连续更新模式，慢进时显示中间状态，甚至可以看到每个海龟的运动
- 在界面页的输入框允许输入文本、数值、颜色或 NetLogo 代码作为模型参数
- 通过扩展支持数组和哈希表（见用户手册数组与哈希表部分）
- 新的性能剖析扩展使你可以计量例程的运行时间
- 新的实验性 __includes 关键词允许将模型代码分到多个源文件
- 颜色变量可以包含 NetLogo color 或 RGB color（作为三个数的列表）
- 可编程滑动条范围（可使用 NetLogo 报告器作为滑动条的 min, max, 或 increment）
- 输出世界时包括所有的绘图数据，当输入世界时绘图内容被恢复
- 声音扩展现除 MIDI 声音外，还可播放音频文件
- 现在注释、监视器和输出区域字体大小可设置
- 注释中的字体颜色可设置，背景颜色是否透明可选
- 界面页的"Snap to Grid"
- 增加菜单项（和 F1 快捷键）快速访问 NetLogo 词典
- 滑动条可垂直或水平
- 模型文件自动保存到临时目录，免遭死机的损失
- HubNet 客户现在更容易编辑（客户不再是独立模型）
- Mathematica-NetLogo 连接在 Mathematica 和 NetLogo 之间提供实时连接，控制模型运行和可视化，进行后分析或实时分析
- 语言变化:
 - ask 命令总是以“无中断”方式运行；如果需要老式的并发行为，使用 ask-concurrent
 - 使用扩展的语法更简单（无下划线、双引号、.jar 后缀）
 - 新的滴答计数原语: tick, ticks, tick-advance, reset-ticks
 - 改变了数值的工作方式:
 - 所有数值都是双精度浮点数
 - 没有小数部分的数值显示为整数（没有小数点）
 - 可以表示更大的整数（最大约 9x10^15）
 - 新原语 of 代替了 VARIABLE-of, value-from, values-from
 - 新报告器 all? 测试主体集合中的所有原语是否满足条件
 - 海龟的 who numbers 不再重用，除非使用了 clear-turtles 或 clear-all

- 新原语 other 返回除调用主体之外的主体集合(等价于with [self != myself])
- 去掉了other-turtles-here 和 other-<breeds>-here, 使用other
- 新的move-to 命令移动海龟到指定的海龟或瓦片处
- 创建海龟或链的命令后面的命令是可选的
- 不再有create-custom-turtles 和 create-custom-<breeds> 命令, 相反使用create-turtles 和 create-<breeds>, 加上可选命令块
- 现在create-turtles以随机方向和颜色创建海龟; 要得到均匀分布、顺序颜色和id的海龟, 使用create-ordered-turtles (简写cro)
- 新的turtle-set, patch-set, link-set报告器, 用来以多种方式创建主体集合
- 去掉turtles-from 和patches-from(相反使用 turtle-set 和 patch-set)
- 新的uphill, uphill4, downhill, downhill4 命令执行爬山
 - 这些命令代替了老的同名命令
 - 新命令的语义略有不同
 - 使用老命令的模型需手工修改
- 新的报告器 no-turtles, no-patches, no-links 报告空的主体集合
- 两个主体集合可以做相等测试
- tie 和 untie 命令不再是实验性的, 它们没有输入参数, 只能由链使用; 有两个捆绑模式 "fixed" 和"free"
- 操作符+ 只能对数值进行; 不能用于字符串或列表;模型必须手工修改, 对字符串使用word ,对数值使用sentence
- 新的histogram 命令替代了histogram-list, histogram-from
- 去掉了 random-int-or-float; 有些模型需手工修改, 使用random或random-float
- 去掉nsum 和 nsum4; 使用sum [reporter] of neighbors/neighbors4
- 新的主体集合原语min-n-of 和max-n-of
- 新的原语 with-local-randomness 运行代码时不改变随机数发生器的状态
- 新的文件 I/O原语 file-flush 强制输出到磁盘
- 新的颜色原语 base-colors 返回 14 个基本色调的列表
- turtle 原语不再接收非整数输入
- patch 原语现在接受非整数并取整, 允许时也回绕
- 观察者不再使用patch-at, turtles-at, BREED-at; 相反使用 patch, turtles-on patch, BREED-on patch
- 比较操作符可用于海龟、瓦片和链
- 新的报告器原语plot-pen-exists?
- 老的 rgb 和 hsb原语更名为 approximate-rgb 和approximate-hsb; 现在它们的输入参数范围是 0-255 而不是 0-1.0
- hsb 和 rgb报告器现在返回RGB 列表而不是NetLogo color
- 新的原语 import-pcolors-rgb 将图像以RGB color导入瓦片
- 新的原语netlogo-applet? 测试模型是否以applet方式运行
- 代码中可以为滑动条设置一个最小、最大或增量所不允许的值
- 不再有locals; 相反使用 let
- 扩展原语必须使用扩展名(默认), 例如 sound:drums 而不是 drums

- file-read 原语跳过注释
 - 去掉了许多文档中未提及的原语的别名
- 用户界面改变:
 - "text box"更名为"note"
 - 滑动条可以用鼠标滚轮移动
 - 内置变量和原始报告器一样使用语法加亮为紫色
 - 与user-* 原语关联的对话框外观和功能更一致
 - 文本区增加了上下文菜单(cut/copy/paste 和 dictionary lookup)
 - 绘图上的 Pens 按钮去掉了 (现在编辑绘图显示或隐藏画笔图例)
- 引擎修正:
 - 现在监视器使用辅助随机数发生器, 因此监视器中的代码不影响模型的可重现
 - 现在对相同字符串重复调用run 和 runresult原语时, 速度大大提高
 - 即使由海龟\瓦片或链使用, display 命令也可以工作
 - 在内部, 列表由链表实现, 而不是数组。这不会影响模型行为, 但可能影响性能 (影响是好是坏取决于具体操作, 见编程指南)
 - 当海龟移动不指明方向时 (例如setxy), 总是采用最短路 (即使回绕)
 - 修正一个 bug, 即某些情况下模型运行过程中输出然后再输入世界时, 结果会改变(通过影响 who numbers 的重用)
 - 对列表的sort-by现在稳定了 (即不打乱相等项的顺序)
 - 读取很长的列表时, file-read 原语更快了
 - 更正了在某些拓扑in-cone 和 distance 不正常的bug
 - 更正了某些布局命令不受随机数种子控制的 bug
 - 与滑动条、开关、选择器、输入框关联的全局命令在无头和 GUI 方式下运行一致, 即拒绝类型不符或超出范围的值
- 其他更正:
 - Tools 菜单的 Halt 项在更多的情境下有效 (而不是挂起 NetLogo)
 - 如果出现无限递归, Netlogo 报错, 而不是盲目运行
 - 当 startup 命令运行时, 用户不能与模型交互
 - 无头方式下支持绘图(使用export-plot 或 export-world 保存图形数据, 以后查看)
 - 在 3D 视图, 如果世界回绕, 在边界上的海龟图形也会回绕
 - 升级了 JOGL, 修正了某些用户的 3D 视图 bug
 - Applets 现在可以找到与模型相关的文件, 即使模型文件与 HTML 文件不在同一目录
 - 在系统动力学建模工具中, 速率连接器可以重定位
 - 系统动力学建模工具现在显示语法错误的位置
 - 在系统动力学建模工具中, 现在可以使用set-current-plot 控制使用哪个绘图, 使用plot-pen-exists?控制绘制哪个存量
 - 默认最大 Java 堆大小为 1G
 - 无头模式可以在 IBM 的 Java 下工作
 - GoGo 扩展现在更易用(不需独立的安装)
 - 新的勾选项允许在 3D 视图中关闭世界线框
- 扩展 API 的变化:

- 现在扩展 API 有版本号，因此只要扩展 API 的版本号没变，就可以在不同版本的 NetLogo 中使用
- 扩展不仅是个 jar 文件，而是一个包含 jar 文件的目录，因此可以包括其他支持文件
- 扩展 API 为定义新的数据类型提供了基本支持(数组和表扩展就是一个例子)
- 现在扩展可以访问随机数发生器
- 扩展实例以及 Java 源代码已安装(以前要单独下载)

3.1.5 版 (2007 年 12 月)

- 安装包支持 Windows Vista
- 重写并扩展了教学#3
- 新的声音扩展命令play-note-later 用于播放乐句
- 不需要附加 jar 的扩展能在 applet 工作
- sort-by 主体集合时相等主体随机排序
- 修正了 bug

3.1 版 (2006 年 4 月)

- 拓扑 (在世界边界上回绕是可选的)
- 对主体集合自动随机排序
- 为种类指定单数形式和复数形式名称
- 现在sort 和sort-by 可用于主体集合
- 对网络和几何增加了 link 原语 (实验性的)
- _tie 和 _untie 原语 (实验性的)

3.0 版 (2005 年 9 月)

- 3D 视图(用于 2D 模型)
- 系统动力学建模工具
- 跟踪特定主体的follow, ride 和 watch 命令
- "drawing layer" 用于主体做出的标记
- 更多的颜色
- 更好的信息页
- GoGo 扩展用于连接 NetLogo 与物理设备
- Color Swatches 对话框用来选择颜色
- 导入图像文件
- 按钮有次序(而不是代码相互交叉)

2.1 版 (2004 年 12 月)

- 无头模式用于命令行操作
- 编辑器加亮括号和方括号的配对
- "action keys" 让按钮可以通过按键激活
- 制作模型的 QuickTime 电影
- 模型增加"output area"
- 改进图形编辑器和内置图形
- 新的原语let 和carefully
- 计算机 HubNet:
 - 现在更可靠
 - 客户自动寻找服务器
 - 改进客户界面和控制中心

2.0.2 版(2004 年 8 月)

- 扩展 API , 使用 Java 编写命令和报告器
- 控制 API, 从外部 Java 代码控制 NetLogo
- 声音扩展制作声音和音乐

2.0 版(2003 年 12 月)

- 全部支持 Mac OS X 和 Linux
- 不再支持 Windows 95, MacOS 8/9
- 改进观感
- 更快、无闪烁, 非基于网格的图形
- 读写外部文件的原语
- 在不同的平台上数学结果完全相同
- 将视图或界面页以图像文件输出
- 改进行为空间
- 计算机 HubNet (不再是 alpha)

1.3 版 (2003 年 6 月)

- 视图控制条
- 选择器
- 新的原语run, runresult, map, foreach, filter, reduce
- 有些原语接受可变数量输入

1.2 版 (2003 年 3 月)

- 速度更快
- 计算机 HubNet (alpha)

1.1 版 (2002 年 7 月)

- "Save as Applet" 将模型嵌入网页
- 支持打印机
- 例程菜单
- 界面页可滚转

1.0 版(2002 年 4 月)

- 第一次发行 (经过一系列 beta 版)

系统需求 (System Requirements)

NetLogo 可以运行在目前几乎所有计算机上。

如果你的 NetLogo 不能正常运行，发送错误报告到 bugs@ccl.northwestern.edu

系统需求：应用程序

Windows

NetLogo 可以运行在 Windows Vista, XP, 2000, NT, ME 和 98 上

NetLogo 安装程序安装 Java 1.5.0，由 NetLogo 独占使用，不影响计算机上的其他程序。

Mac OS X

强烈推荐 Mac OS X 10.4(或以上)，10.3 或 10.2 也支持。

请运行软件更新以确保有最新的 Java。

其他平台

NetLogo 可以运行在安装了 Java 虚拟机 1.4.1 以上的任何平台上。1.5.0_13 以上更好。

通过运行提供的脚本程序 `netlogo.sh` 启动 NetLogo

系统需求：保存 Applets

NetLogo 模型存为 Java Applet 后，可以运行在任何安装了 Java 1.4.1 以上的浏览器中。

系统需求：3 维视图

少数情况下一些老的、性能差的系统不能成功使用 3 维视图。试试看。

一些系统能使用 3 维视图但不能切换到全屏模式，这与图形卡有关。例如 ATI Radeon IGP 345 和 Intel 82845 可能不能工作。

Windows 用户关于 Java 的技术细节

多数 Windows 用户应选择捆绑了 Java 的 NetLogo 下载包。

有两个可能的原因使用没有捆绑 Java 的其他下载包：

1. 希望下载包较小，少占用硬盘空间
2. 因为某些特别的技术原因，你需要使用其他 Java 版本

如果你认为其他下载包适合你，请阅读下面的详细技术信息。

即使你已经安装了 Java，它也可能不能与 NetLogo 一起工作。

为了获得最佳性能, NetLogo 使用了 Java 虚拟机的一个特别选项 “server” 。JRE 默认安装时没有这个选项, 只有 JDK 有这个选项。

如果你不是 Java 开发人员, 你可能使用的是 JRE, 而非 JDK。

因此, 如果你要用自己的 Java 虚拟机运行 NetLogo, 你有两种选择:

1. 确保你有完全的 JDK 而非 JRE。

2. 或者你能编辑一个配置文件, 让 NetLogo 与 JRE 一起工作。

我们不推荐选项 2, 因为没有 “server” 选项使 NetLogo 运行特别慢。

如果你非要用选项 2, 你就应这样做。你必须告诉 NetLogo 不要使用 “server” 虚拟机选项。首先, 使用本页的下载包安装 NetLogo, 然后使用文本编辑器如 NotePad 打开 NetLogo. 4. 0. 2. lax, 这个文件在 NetLogon 安装目录里。在附加 java 选项里去掉-server 选项。将这一部分:

```
# LAX.NL.JAVA.OPTION.ADDITIONAL  
# -----  
# don't load native libs from user dirs, only ours, also run server not client VM  
lax.nl.java.option.additional=-Djava.ext.dirs= -server -Dsun.java2d.noddraw=true
```

改为:

```
# LAX.NL.JAVA.OPTION.ADDITIONAL  
# -----  
# don't load native libs from user dirs, only ours, also run server not client VM  
lax.nl.java.option.additional=-Djava.ext.dirs= -Dsun.java2d.noddraw=true
```

再说一次, 使用这种方法, NetLogo 性能会变差。

已知问题 (Known Issues)

如果NetLogo有问题, 请将bug报告发给我们。具体方法见[“联系我们”](#) 部分。

已知 bug (所有系统)

语言/引擎 bug

- 数组和表扩展与输入世界功能仅部分兼容。当输出世界时（使用[export-world](#)命令或Export World菜单项），数组和表“按值”输出。这意味着如果相同的表或数组存储在多个位置，当输出然后再次输入时，在原始数组或表的每个原始位置上有不同的数组或表。这些复制品初始时包括相同的值，但当一个复制品变化时，其他的不变。
- "Export World"不保存用文件原语打开的文件状态。如果有文件打开，输出世界，然后将世界输入到NetLogo，需要在恢复运行前重新打开文件。
- 如果原始图形文件有灰度调色板，Java的一个bug使得用[import-pcolors](#)输入的瓦片颜色变得浅一些。要解决这个问题，将图像文件转换为RGB调色板。

其他 bug

- 内存溢出没有得到完美处理
- 当缩放时，“Snap to Grid”功能不可用
- 在画图层画出然后擦除一条线，不会精确的擦掉每个像素
- 需要附加外部 jar 的扩展在保存为 applet 的模型里不能工作(我们已经着手解决这个问题)
- 在某些图形配置 ([some graphics configurations](#)) 下，3D视图不能工作；其他配置下3D视图能工作，但全屏模式不行
- 当以无头模式(从命令行)运行模型时，如果模型是使用早期 NetLogo 版本创建的，则可能不能正常工作。在以无头方式运行前，使用当前版本的 NetLogo 以 GUI 模式打开然后重新保存模型。

Windows 上的 bug

- 不是在每台计算机上 Help 的"User Manual" 项都能用 (Windows 98 和 ME 最可能受影响，Windows 版本越新越少受影响)
- 在某些笔记本上，当滚转时例程页和信息页可能混乱。要避免这个bug，减小NetLogo 窗口，和/或减少监视器的颜色深度。(例如从 32-bit 减为 16 或 8-bit)。这是Java本身的一个bug，不是NetLogo的问题。关于这个bug的技术细节，见

<http://developer.java.sun.com/developer/bugParade/bugs/4763448.html> (需要免费注册)。
鼓励NetLogo用户访问此站点投票，让Sun修正此bug。

Macintosh 上的 bug

- 当在 Finder 打开一个模型时 (在上面双击，或把它拖到 NetLogo 图标)，如果 NetLogo 还没有运行，这个模型可能或不能打开。这个 bug 是偶发的。(如果 NetLogo 已运行，模型总能打开)
- 仅在 Mac OS X 10.4 上，“Copy View” 和 “Copy Interface” 项不工作：结果图像失真。解决方法是使用“Export View”和 “Export Interface” 项。如果使用软件更新从 Apple 得到最新的 Java，这个问题消失。
- 在 10.4 以前的 Mac OS X，NetLogo 的菜单可能出问题，“Quit”项不工作。如果出现这个问题，可以按下 NetLogo 标题栏左侧的红色关闭按钮退出 NetLogo。

Linux/UNIX 上的 bug

- 用户手册总是在 Mozilla 里打开，而不是默认浏览器。一个可能的解决方法是在你喜欢的浏览器里收藏 docs/index.html 做为书签。另一个方法是制作一个符号连接，叫做“mozilla”(这是 NetLogo 尝试运行的命令名)，但实际运行另一个浏览器。
- 在 Linux 上我们发现了一个问题，“exp”对相同的值有时返回略有不同的结果（最后一位不同）。根据 Sun 的一个工程师的说法，该问题仅发生在 Linux 内核 2.4.19 上，但我们发现在很多内核新版本上也有。我们假设这个问题是 Linux 特有的，不会发生在其他 Unix 上。我们不确定这个问题是否会在实际运行 NetLogo 时发生，还是只发生在测试情形下。这个 bug 是 Sun 的 Java VM 的，不是 NetLogo 的。我们希望只是“exp”报告器受影响，但我们不能完全确认。鼓励 NetLogo 用户访问此站点
<http://developer.java.sun.com/developer/bugParade/bugs/5023712.html> (需要免费注册)，投票让 Sun 修正此 bug。
- 如果 NetLogo 找不到 Lucida 字体，菜单会模糊不清。这已知发生在 Fedora Core 3 升级后。重启 X Font Server (xfs) 解决了这个问题。
- Sun's 1.5.0 Java runtime 对 GTK 2.0 和 NetLogo 有显示问题。问题可能包括窗口不能正常更新，界面元素的大小不对劲，菜单底部截去，视图上出现奇怪字符等。要避免这个问题，升级到 Java 1.6。

计算机 HubNet 的已知问题

参见 [HubNet 指南](#)，在那儿列出了已知问题。

联系我们 (Contacting Us)

用户反馈对我们设计和改进 NetLogo 非常有价值。我们很高兴听到你的反馈。

网站

我们的网站 ccl.northwestern.edu 包括我们的邮件地址和电话号码。也有我们的人员信息和研究活动信息。

反馈、问题等

如果 你 的 模 型 需 要 帮 助 , 考 虑 在 NetLogo 用 户 组 发 帖
<http://groups.yahoo.com/group/netlogo-users/> 。

还 有 一 个 专 门 针 对 教 育 人 员 的 组
<http://groups.yahoo.com/group/netlogo-educators/> 。

如果有反馈、建议、问题，发邮件到feedback@ccl.northwestern.edu 。

报告 Bug

如果发现NetLogo有bug, 想告诉我们, 请写信到bugs@ccl.northwestern.edu 。当报告 bug时, 尽量包括下面的信息:

- 问题的完整描述以及它怎样发生
- 有问题的 NetLogo 模型或代码。可能的话, 附上完整的模型。
- 系统信息: NetLogo 版本, 操作系统版本, Java 版本等。 (这些信息可以在 NetLogo 的"About NetLogo"菜单项, 然后单击 System 标签。在保存的 applet 里, ctrl+单击 (Mac) 或在 applet 白色背景右击, 可以得到该信息。)
- 所显示的任何错误信息。

模型实例：聚会（Sample Model:Party）

这一部分让你思考什么是计算机建模以及如何使用它，也让你对 NetLogo 软件有所了解。我们推荐初学者从这里开始。

聚会

你是否参加过聚会，注意过人们是怎样聚集成小组的吗？你也可能注意到人们并非一直呆在一个小组里，而是走来走去。当个人走来走去时，小组就发生变化。如果你长期观察这种变化，你应该注意到模式的形成。

例如，在社交场合人们倾向于展示出与工作或家庭中不同的行为。那些在工作中信心满满的人可能在社交场合变得羞怯，而那些在工作中安静保守的人却可能与朋友发起聚会。

聚集模式也取决于聚会的性质。在某些场合，人们接受训练组织成混合小组，例如聚会游戏或校园活动。但在非结构化的气氛里，人们以更加随机的方式形成小组。

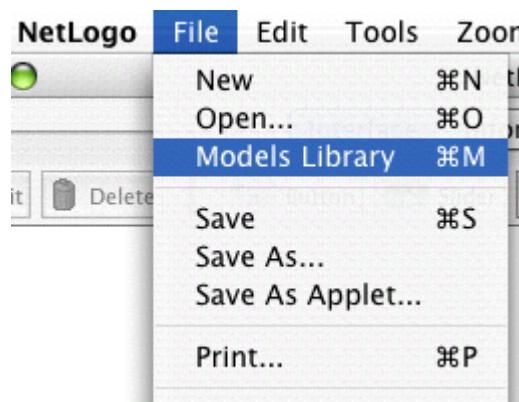
这种分组行为有没有什么模式呢？

让我们使用计算机对聚会中人们的行为建模，更详细的考察这个问题。NetLogo 的“Party”模型从性别这个特殊角度考察这个问题：为什么这些小组多数是男性，或多数是女性？

我们使用 NetLogo 研究这个问题。

操作步骤：

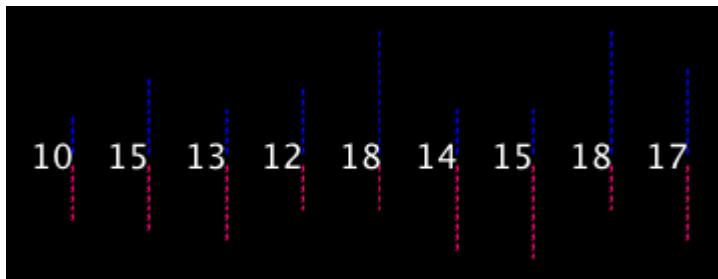
1. 启动 NetLogo
2. 在 File 菜单中选择 "Models Library"



3. 打开文件夹 "Social Science"
4. 点击模型 "Party".
5. 按下"open" 按钮
6. 等待模型加载
7. (可选)放大 NetLogo 窗口，这样能看多更多内容

8. 按下”Setup“ 按钮

在视图中你可以看到粉线和蓝线，还有数字。



这些线表示聚会上男女混合的小组。男性用蓝色表示，女性用粉色。数字是每个小组的人数。

所有小组的人数相同吗？

所有小组的每种性别人数相同吗？

例如你邀请了 150 人参加聚会，你想知道人们怎样扎堆。假设人们分成了 10 组。

你怎么思考分组情况？

这里我们使用计算机仿真，而不是去问你那 150 个亲密朋友。

操作步骤

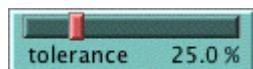
1. 按下 "go" 按钮 (再次按下 "go" 会停止模型运行)
2. 观察人们的移动直到模型停止
3. 看图形输出了解发生了什么

现在每组有多少人？

开始时你可能认为将 150 人分成 10 组的结果是每组大约 15 人。从模型运行得知，人们并没有均等的分成 10 组 — 相反，有些组人数特别少，而有些组人数却特别多。另外，随着时间发展，从所有小组男女都有转变为所有小组均由同性组成。

这怎么解释？

对这个问题有很多可能的回答。本模型的设计者认为聚会上的小组不是完全按随机方式形成的，小组如何形成取决于个体的行为。模型设计者关注一个特殊变量“tolerance”（容忍度）：



这里将容忍度定义为个体感到舒服的异性的比例。如果小组中异性比例超过容忍度，他们就觉得不舒服，因此离开这一组去寻找别的小组。

例如，如果容忍度水平设为 25%，那么一个男性只有在女性比例少于 25% 的小组里才感到舒服。同样女性只有在男性少于 25% 的小组里才感到舒服。

当个体变得不舒服时选择离开，移动到别的小组，这可能又让这个组中的某些人不舒服。这种链式反应不断进行，直到聚会上的所有人都感到舒服。

注意这个模型中，容忍度不是固定的。用户可以用滑动条改变容忍度，重新运行模型，看看结果如何。

怎样重新启动模型：

1. 如果 "go" 按钮已按下 (黑色)，说明模型还在运行。再次按下该按钮停止运行。
2. 通过拖动红色手柄调整 "tolerance" 滑动条，设置新值
3. 按下"setup" 按钮重设模型
4. 按下"go" 按钮再次启动模型

挑战

作为聚会的主人，你希望看到各组里都是男女混合。调整容忍度滑动条，让每组都男女混合。

为保证 10 个小组都是男女混合，容忍度水平要设成多少？

看看你的预测结果吧。

你能想到可能影响每组中男女比例的其他因素或变量吗？

进行预测，用模型检验你的想法。放开手脚同时操作多个变量。

当你检验假设的时候，你会从数据中注意到模式的涌现。例如，如果保持聚会人数不变，但逐渐增加容忍度水平，更多的组会成为男女混合组。

容忍度水平必须是多少才能得到混合组？

容忍度水平与混合组的比例有什么关系？

用模型思考

用 NetLogo 对聚会这样的情景建模使你可以对系统进行快速、灵活的试验，而在现实情况下这是很困难的。建模也给了你少受偏见的影响去观察各种情景的机会，因为你可以检查系统内部的动态。你会发现随着你建模越来越多，对许多现象的原有的想法会遇到挑战。例如 Party 模型一个令人惊讶的结果是：即使容忍度水平相对较高，大量性别分离仍然会发生。

这是关于“涌现”现象的一个经典例子，这里小组模式是许多个体交互的结果。“涌现”思想可以应用在几乎任何领域。

你能想到别的涌现现象吗？

要想获得更多的例子，对这个概念有更深的理解，你可以探索 NetLogo 模型库，里面有许多模型，演示了各种系统中的这类思想。

要更详细了解关于涌现的讨论以及 NetLogo 如何帮助学习者进行探索，参见 "[Modeling Nature's Emergent Patterns with Multi-agent Languages](#)" (Wilensky, 2001).

下一步？

用户手册的 [教学 1：运行模型](#) 讲述模型库的更多细节。

如果你想学习怎样在更深的层次上探索模型, [教学 2: 命令](#) 将引导你了解NetLogo建模语言。

最后, 你将学习 [教学 3: 过程](#), 你将学习怎样替换、扩展模型, 增加新行为, 以及如何建造自己的模型。

教学# 1：模型 (Tutorial #1 :Models)

如果你已经读过 [模型实例：聚会](#) 部分，你应该简单了解了怎么与 NetLogo 模型交互。这一部分更深入了解一些功能，这些功能在你探索模型库时会用到。

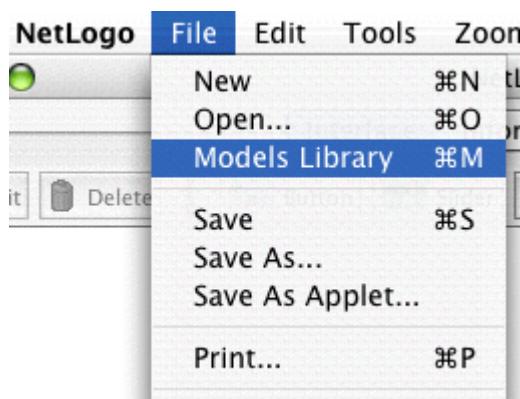
在整个教学过程中我们会请你做一些预测，预测修改模型后会出现什么后果。记住，后果往往令你惊讶。我们认为这种惊讶非常令人激动，提供了特棒的学习机会。

有些人发现把这些教学材料打印出来放在手边，对学习很用帮助。打印出来后，你的计算机屏幕上更大的空间显示你要查看的 NetLogo 模型。

模型实例：狼吃羊（Wolf Sheep Predation）

我们要打开一个模型实例详细探索。我们试试一个生物模型：狼吃羊，这是一个捕食者-猎物种群模型。

- 从文件菜单打开模型库



- 从 Biology 部分选择"Wolf Sheep Predation" 按下"Open"

界面标签页填满了许多按钮、开关、滑动条和监视器。这些界面元素使你可以与模型交互。按钮是蓝色的，用它们设置、启动、停止模型。滑动条和开关是绿色的，它们用来修改模型配置。监视器和绘图是浅褐色的，它们用来显示数据。

如果你想让窗口大一些，让所有元素都能容易的看到，你可以使用窗口顶部的 zoom 菜单。

当你第一次打开模型时，你会看到视图是空的（全黑）。要让模型开始，你需要先进行初始设置。

- 按下 "setup" 按钮

视图中出现什么？

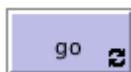
- 按下 "go" 按钮开始仿真

模型运行时，狼群和羊群发生什么？

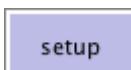
- 按下"go" 按钮停止运行

控制模型：按钮

按钮按下后模型就会通过执行一个动作做出响应。按钮分为“一次性”(once) 和“永久性”(forever)两种，可以通过按钮上的一个符号区分二者。永久性按钮的右下角有两个箭头，就像这样：



一次性按钮没有箭头，就像这样：



一次性按钮执行动作一次然后停止。当动作完成后，按钮弹起。

永久性按钮不断的执行一个动作。当你想让动作停止时，再次按下按钮，它会完成当前动作，然后弹起。

大多数模型，包括狼吃羊模型，有一个一次性按钮称为“setup”和一个永久性按钮称为“go”。许多模型还有一个一次性按钮称作“go once”或“step once”，它们很像 go 按钮，但区别在于它们只执行一步（时间步长）。使用这样的一次性按钮能让你更仔细的查看模型的运行过程。

停掉永久按钮是终止模型的正常方式。通过停止永久性按钮暂停模型运行，然后再次按下按钮让模型继续，这非常安全。你也可以使用 Tools 菜单的“Halt”停止模型运行，但是只有当模型因某种原因卡住时才应该这样做。使用“Halt”可能会让模型在某次行动的中间停住，这可能导致模型乱套。

- 如果你愿意的话，试试狼吃羊模型的 "setup" 和 "go" 按钮

如果使用同样的设置多次运行模型，结果会有不同吗？

控制速度：速度滑动条

速度滑动条控制模型运行速度，就是海龟的移动速度、瓦片颜色改变的速度，等等。



滑块左移使模型速度变慢，每个时间步之间的暂停时间更长，这样更容易观察发生了什么。你甚至可以让模型运行的极慢，能看到每个海龟做什么。

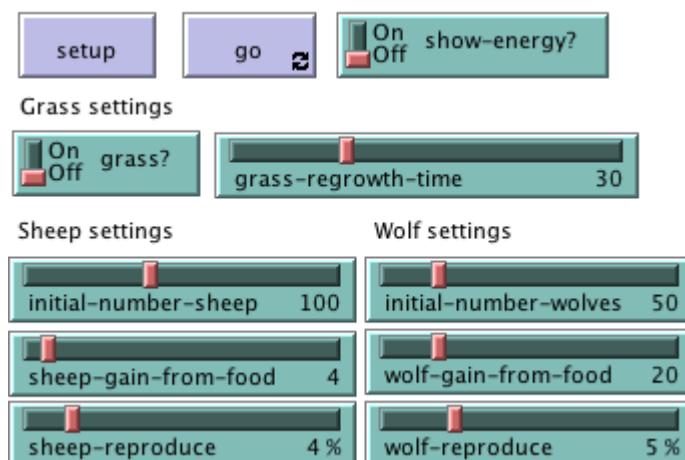
滑块右移使模型速度变快。NetLogo 可能会跳帧，意思是说不再是每个时间步都进行视图刷新。显示世界状态要耗费时间，因此少显示这些一般意味着运行更快。

注意如果滑块太靠右的话，视图更新太频繁，看起来却好像变慢了。实际并没有变慢，你可以根据时钟显示确认这一点，只是视图更新频率降低了。

调整设置：滑动条和开关

模型的配置给了你尝试不同场景或假设的机会。修改配置，运行模型，观察这些改变所引起的反应，使你能更深入的了解所模拟的现象。开关和滑动条用来修改模型配置。

下面是狼吃羊模型中的开关和滑动条：



我们来试试这些行为的效果。

- 如果狼吃羊模型还没打开，现在打开它
- 按下"setup" 和 "go"，运行大约 100 时间步 (注意：图的右上有时钟读数)
- 按下"go" 停止

羊群怎么变化？

我们看看如果改变下面的设置的话，羊群怎么变化。

- 打开"grass?" 开关
- 按下"setup" 和 "go"，运行与上次差不多相同的时间

这个开关对模型有什么作用？结果和上次一样吗？

像按钮一样，开关也有与它相连的信息，这些信息采用开/关格式。开关发出特别的指令，这些指令对模型并非必要，但为模型增加了附加的维度。打开“grass?”影响模型结果。本次运行之前，草的增长率为常数。在掠食-食饵关系中这是不真实的，因此通过设置和打开草的增长率，我们能够对三个因素建模：羊、狼和草。

另一种配置类型是滑动条。

滑动条是不同于开关的一种配置类型。开关有两个值：开或关。滑动条是一个可调的数值范围。例如“initial-number-sheep”滑动条最小值为 0，最大值为 250。模型运行时可以有 0 只羊，也可以有 250 只羊，或者中间的任何一个数值。试试看。当你从左到右移动滑块时，滑动条右侧的数字变化，这就是当前值。

我们研究一下狼吃羊模型的滑动条。

- 阅读信息标签页中的说明，了解每个滑动条表示什么。

信息标签页提供了模型的指导和洞察。在这一页你会找到模型的解释，尝试建议以及其他信息。你可以在运行模型前读它，也可以先试试模型，再返回来读它。

如果仿真开始时羊更多而狼更少，会怎么样？

- 关掉“grass?”开关
- 设置“initial-number-sheep”滑动条为 100.
- 设置“initial-number-wolves”滑动条为 20.
- 按下“setup”和“go”。
- 模型运行约 100 时间步

尝试重复运行模型几次。

羊群数量发生了什么变化？

结果令你惊讶吗？调整哪些其他开关、滑动条能帮助羊群？

- 设置“initial-number-sheep”为 80，“initial-number-wolves”为 50。(这与你第一次打开模型时接近)
- 设置“sheep-reproduce”为 10.0%.
- 按下“setup”和“go”。
- 模型运行约 100 时间步

本次运行狼群怎么样？

当你打开模型时，所有的滑动条和开关采用默认配置。如果你打开一个新模型或退出程序，你的更改不会保存，除非你选择保存它们。

(注意：除了滑动条和开关，一些模型还有第三类配置元素，选择器（Chooser）。但本模型没有。)

收集信息：绘图和监视器

建模的一个目的是对那些难以在实验室中进行研究的问题收集数据。NetLogo 主要有两个显示数据的方式：绘图和监视器。

绘图 (Plots)

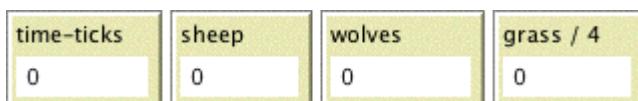
狼吃羊中的图有三条线：羊、狼和草/4（草除以 4 的原因是为了别使图形太高）。这些线显示了随着时间推进模型中发生了什么。要想知道每条线代表什么，在图形窗口的右上角单击”Pens”，打开画笔图例。一个关键字说明了每条线是什么。在本例中就是种群数量。

当图快被充满时，水平轴增加，以前的数据被压缩只占一部分空间，更多的空间用来绘制将来的图形。

如果你想保存图上数据以备查看，或在另一个程序里进行分析，使用 File 菜单的”Export Plot”。这些数据就被保存，数据格式可以被电子表格，如 Excel，或数据库程序识别。也可以通过 Ctr+单击 (Mac) 或右击(Windows) 弹出快捷菜单，然后选择”Export...”。

监视器 (Monitors)

监视器是模型显示信息的另一种方法。下面是狼吃羊模型中的监视器：



监视器”time-ticks”告诉我们仿真时间。其他的监视器告诉我们狼、羊、草的数量。（记住，草的数量除以 4，为了别使图形太高）

当模型运行时监视器中的数值不断更新，而图形能显示模型整个运行过程中的数据。

注意 NetLogo 还有另一种监视器，叫做”agent monitors”，在教学 2 里介绍。

控制视图

如果观察界面标签页，会看到沿工具条上边缘有一条控件。这些控件改变视图的不同方面。

试试这些控件的效果。

- 按下 “setup” 和 “go” 启动模型
- 模型运行时，将速度滑动条向左移动

发生什么？

如果模型运行的太快，你可以使用它看清细节。

- 移动速度滑动条到中间
- 右移速度滑动条
- 现在试试勾选或不勾选“view updates”选择框

发生什么？

如果你不耐烦，想让模型运行的更快，可以快进也可以关闭视图更新。快进（速度滑动条右移）关闭视图更新，因此模型运行的更快，这是因为更新视图需要时间。

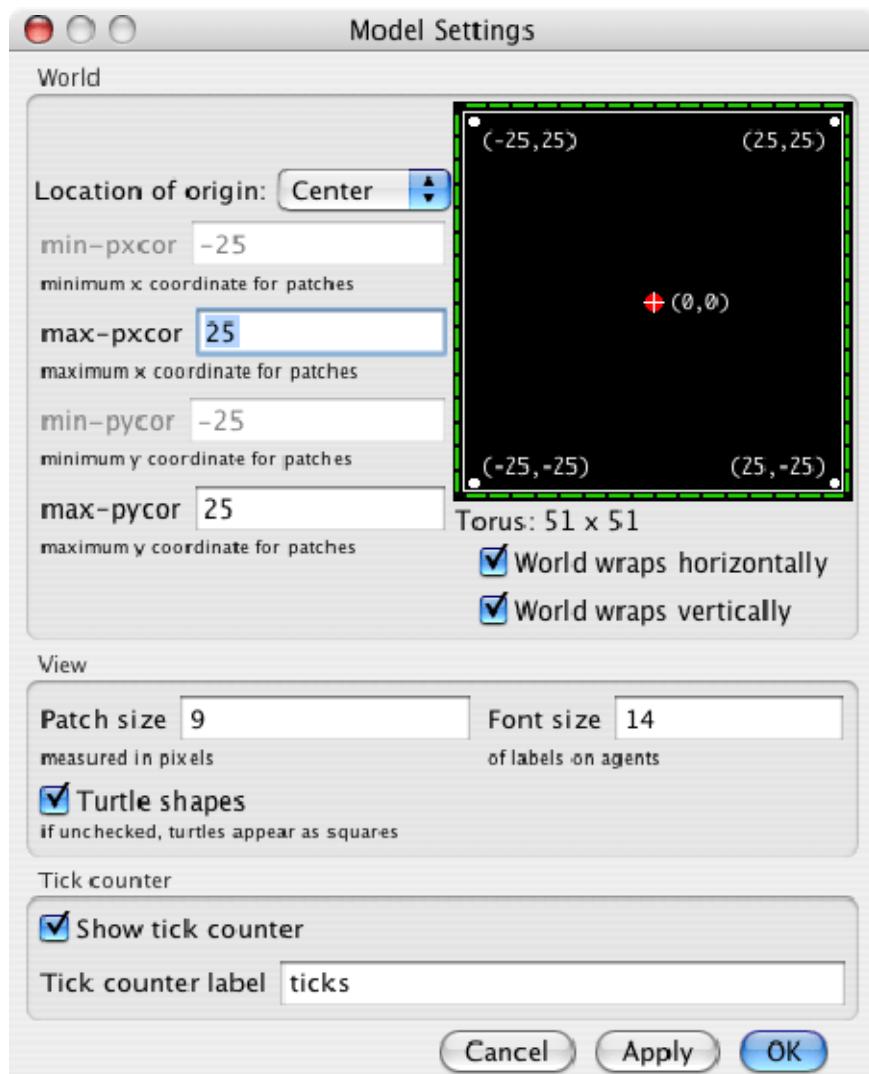
当视图更新完全关闭后，模型继续在后台运行，绘图和监视器也一直在更新。如果你想看看在发生什么，你需要重新勾选视图更新选项。当视图更新关闭后，多数模型运行速度更快。

视图的尺寸由 5 个设置共同决定：最小 X、最大 X，最小 Y、最大 Y 和瓦片尺寸。现在让我们看看当我们改变狼吃羊模型视图的尺寸时发生什么。

有更多的关于世界和视图的设置，因为工具条面积有限，没有放上。“Settings...”按钮可以让你获得其他设置。

- 按下工具条上的“Settings...”按钮

会打开一个对话框，其中包括所有视图设置：



当前的 max-pxcor, min-pxcor, max-pycor, min-pycor 和 Patch size 是多少?

- 按下 "cancel" 按钮取消所做的改变。
- 将鼠标指针靠近视图，但不要进入视图窗口

注意到鼠标指针变成了十字型

- 按着鼠标按钮在视图上拖动

现在视图被选中，看到视图被灰框环绕，你可以确知这一点。

- 拖动黑色方块型“手柄”。手柄在视图的边上和角上。
- 在界面标签页的白色背景的任何地方单击，反选视图
- 再次按下"Settings..." 按钮，看看设置

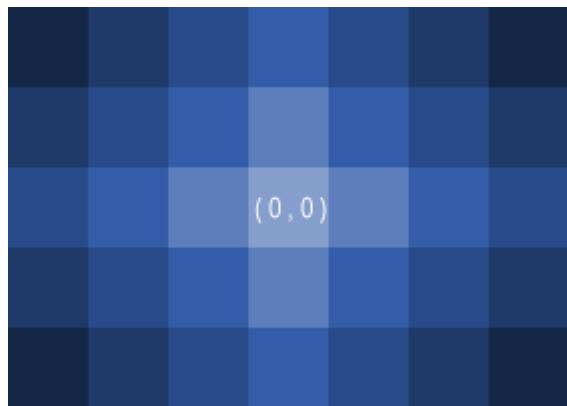
哪些数字改变了？

哪些数字没有改变？

NetLogo 世界是由“瓦片”构成的二维网格，瓦片是网格中的一个方格。

在狼吃羊模型中，当“grass?”开关打开时瓦片很容易看到，因为一些瓦片是绿色的，一些是褐色的。

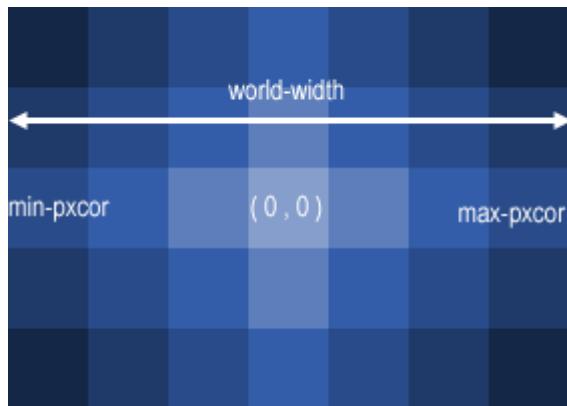
可以把瓦片想象成地板上铺的方形瓷砖。默认情况下房子正中的一片瓷砖标记为(0,0)，意味着如果我们在水平和垂直方向画等分线，交叉点这在此处。这样我们就有了一个在房间中定位对象的坐标系统：

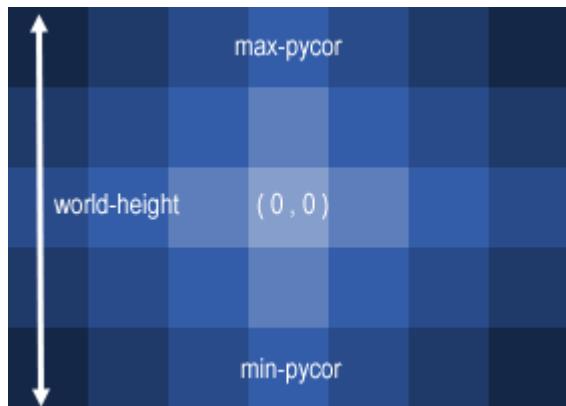


瓷砖(0,0)与房间最右边之间有多少瓷砖？

瓷砖(0,0)与房间最左边之间有多少瓷砖？

在 NetLogo 中，从右到左的瓷砖数称为世界宽度 (world-width)。从顶到底的瓷砖数称为世界高度 (world-height)。这些数字由顶、低、左、右边界 (top, bottom, left and right) 来定义。





在这些图中, max-pxcor 是 3 , min-pxcor 是 -3, max-pycor 是 2 , min-pycor 是 -2. 当你改变瓦片大小时, 瓦片(瓷砖)的数量不变, 只是屏幕上瓦片的大小变化了。 让我们看看改变世界的最小、最大坐标的效果。

- 使用仍在打开的 Settings 对话框, 改变 max-pxcor 为 30, max-pycor 为 10。注意 min-pxcor 和 min-pycor 也变了, 这是因为默认原点 (0,0) 在世界的中心。

视图的形状发生了什么变化?

- 按下 "setup" 按钮

现在可以看到你创建的新瓦片

- 再次按下"Settings..." 按钮
- 将瓦片大小设为 20, 按下"OK".

视图的大小发生了什么变化? 它的形状变了吗?

编辑视图也可以让你改变其他设置, 包括标签字体大小\视图是否使用形状 (shape) 。 随便试试这些设置吧。

当你探索完狼吃羊模型后, 你可能想花点时间探索一下模型库中的其他模型。

模型库

模型库包括五部分: Sample Models, Perspective Demos, Curricular Models, Code Examples, HubNet Computer Activities.

模型样例 (Sample Models)

Sample Models 部分是分科目组织的, 目前有 210 多个模型。我们一直在增加模型, 因此过段时间后能看到新加的模型。

有些文件夹下包含"(unverified)"子文件夹。这些模型是完整、可用的, 但模型的内容、

精度、代码质量等仍在评审之中。

视角演示 (Perspective Demos)

这些模型在 Sample Models 中也有。但是略作修改，用来演示 NetLogo 的视角功能。

课程模型 (Curricular Models)

这些模型是西北大学 CCL 开发的在学校使用的课程。有些模型在 Sample Models 中也有，有些没有。看看信息标签页，了解更多的信息。

代码例子 (Code Examples)

这是 NetLogo 特别功能的一些简单演示。当你以后扩展现存模型或新建模型时很有用。例如，你想在模型中增加直方图，可以看看“Histogram Example”，看看怎么做。

HubNet 计算机活动(HubNet Computer Activities)

这一部分包括教室中使用的参与式仿真。要了解 HubNet 的更多信息，参见 [HubNet Guide](#)。

下一步？

如果想在更深的层次上探索模型，[教学 2：命令](#) 将引导你了解 NetLogo 建模语言。

在 [教学 3：例程](#) 中，你学习怎样替换现有模型，以及如何构建你自己的模型。

教学# 2: 命令 (Tutorial #2:Commands)

在教学#1, 你有机会查看了一些 NetLogo 模型, 我们引导你打开、运行模型、按下按钮、改变滑动条和开关值, 以及使用绘图和监视器从模型收集信息。在这一部分, 焦点从观察模型转换到操纵模型。你将开始看模型的内部运转, 能够改变它们的样子。

模型实例: 基本交通模型(Traffic Basic)

- 到模型库去 (File 菜单).
- 在"Social Science"部分, 找到并打开 Traffic Basic
- 运行模型两分钟, 感受一下
- 如果有什么问题, 去信息标签页查查

在这个模型里, 你会注意到一系列蓝车里有一辆红车, 车流同向移动。这些车时不时的会挤成一堆, 无法移动。这是关于幽灵式阻塞的模型, 即有时交通流会出现阻塞, 但却找不到任何明显的原因, 例如事故、断桥、侧翻的卡车等。要形成交通阻塞, 没有“明显原因”(centralized cause) 是需要的。

你可以改变配置运行几次, 对模型有个全面理解。

当你使用这个 Traffic Basic 模型时, 有没有注意到需要给模型增加什么?

看看这个模型, 你会注意到环境太简单了, 就是黑色背景、白色街道、一些蓝车和一辆红车。需要对模型做点改变: 改变车的形状和颜色、加上房子或路灯、新建信号灯、或者再创建一条车道。这些建议有些是装饰性的, 只是改善模型的观感, 另外一些是行为性的。在本教程里我们主要关注较简单的、装饰性的改变。 (教学#3 深入介绍行为性改变, 那需要在 Procedures 页面中进行修改)

我们使用命令中心做这些简单改变。

命令中心 (The Command Center)

命令中心位于Interface Tab页, 在这里你可以向模型发出命令或指令。命令就是你可以发给NetLogo主体 (海龟、瓦片、链、观察者) 的指示。 (参见界面指南[Interface Guide](#), 了解命令中心的各个部分)

在 Traffic Basic 中:

- 按下"setup" 按钮
- 找到命令中心
- 在命令中心底部的白框里按一下鼠标
- 输入下面所示的文本



- 按回车键

视图发生什么变化?

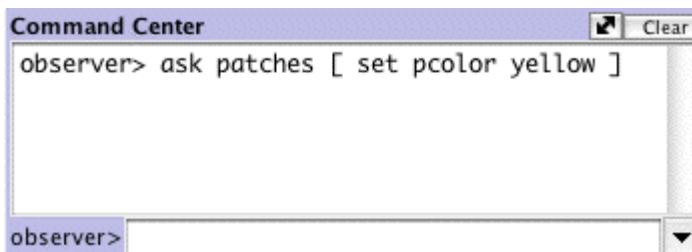
你注意到视图背景变成了黄色, 街道消失了。

为什么车没有变成黄色?

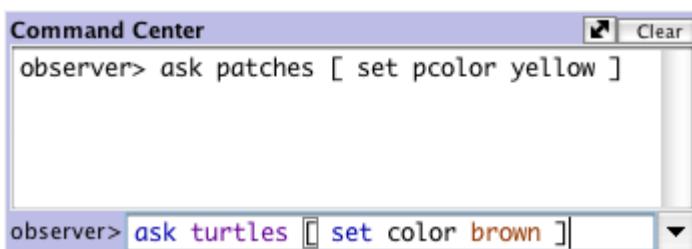
回头看看刚才输入的命令, 我们只是请求瓦片改变颜色。在这个模型里, 车辆用另外一种称为”海龟”的主体表示。因此车辆并没有接收指令, 也就没有改变。

命令中心发生了什么?

你可能没注意到你刚才输入的命令现在显示在命令中心的中间部分白框里, 就像下面这样:



- 在命令中心底部的白框中输入下面的文本:



结果和你想的一样吗?

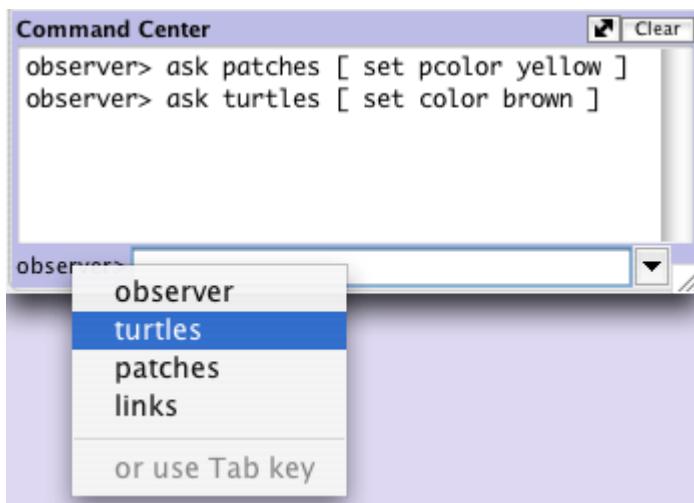
视图里是黄色背景, 中间是一串灰色的车:



NetLogo是由海龟、瓦片和观察者组成的二维世界。瓦片构成背景，海龟在背景上移动，观察者（observer）是观察着所有事情的一个主体。（关于世界的细节，参考 NetLogo编程指南[NetLogo Programming Guide](#)）

在命令中心，我们可以给海龟、瓦片和观察者发出命令。我们通过命令中心左下角的弹出式菜单进行选择，也可以用 Tab 键在选项之间循环。

- 在命令中心,单击左下角的"observer>":



- 在弹出菜单中选 "turtles"
- 输入 set color pink , 回车
- 按下 tab 键直到在左下角看到"patches>"
- 输入 set pcolor white, 回车.

现在视图看起来怎么样？

你注意到这两条命令和前面的 observer 命令的区别了吗？

观察者（observer）俯视着世界，因此使用ask向瓦片或海龟发出命令。正如第一个例子那样(observer> ask patches [set pcolor yellow])，observer必须请求（ask）瓦片把它的颜色pcolor设为黄色。但在第二个例子中，命令直接发给了一组主体(patches> set pcolor white)，你只需直接给出命令。

- 按下"setup".

发生什么？

为什么视图变回了原样，还是黑背景白路？因为按下"setup"后，模型重新按例程页中的内容配置模型。命令中心一般不用来对模型做永久性修改，而是用来对当前模型进行定制，让你能操纵模型，回答探究模型时冒出来的“what if”问题。（例程页（Procedures tab）在下个教学里解释，也可参考编程指南[Programming Guide](#).）

我们已熟悉了命令中心，再看看 NetLogo 中关于颜色的一些细节。

操纵颜色

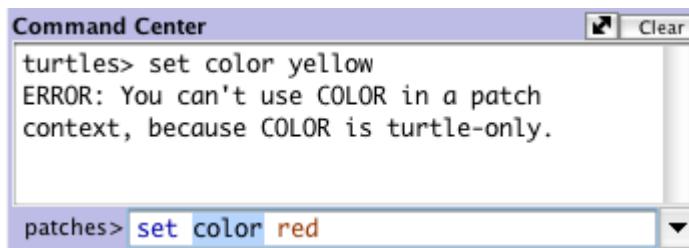
你可能注意到在上面我们使用了两个不同的词来改变颜色: [color](#)与 [pcolor](#).
 color 与 pcolor 有何区别?

- 在命令中心的弹出菜单中选 "turtles" (或使用 tab 键).
- 输入set color blue, 回车

车辆发生什么变化?

思考一下你做了什么让车变成了蓝色, 试试把瓦片变成红色。

如果想让瓦片变成红色, 出现一条错误信息:



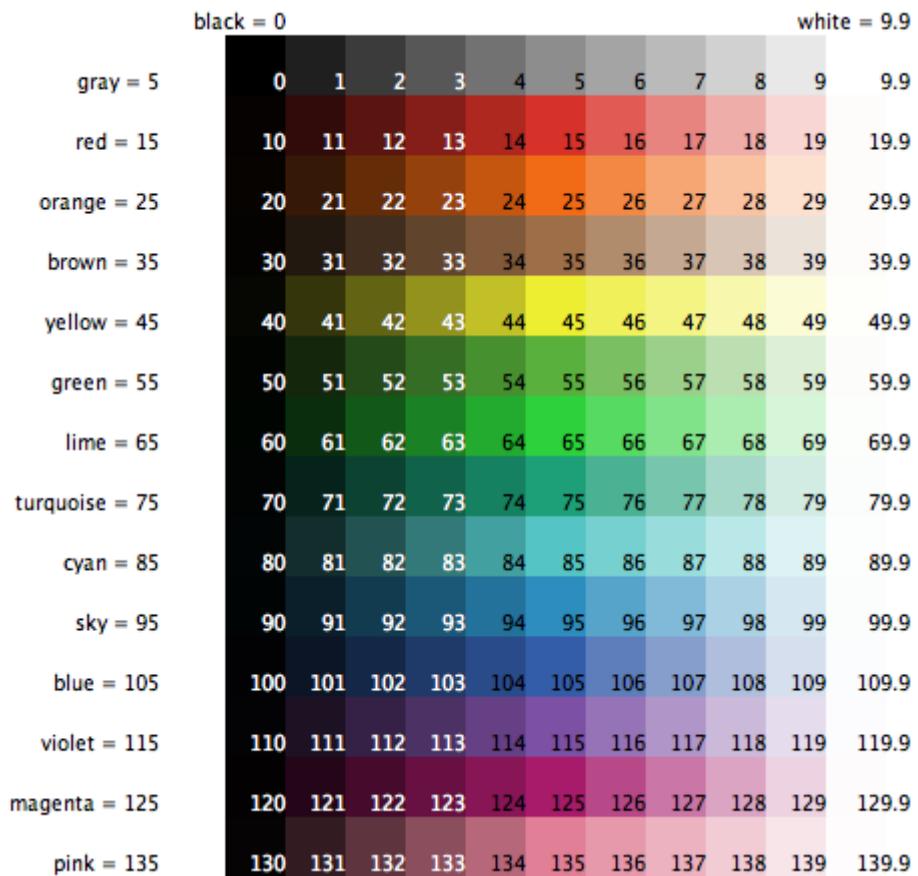
- 输入set pcolor red, 回车.

[color](#) 和[pcolor](#)是变量 (variables)。有些命令和变量是海龟专用的, 有些是瓦片专用的。例如, [color](#)是一个海龟变量, 而[pcolor](#)是一个瓦片变量。

继续尝试, 使用[set](#)命令和这两个变量改变海龟和瓦片颜色。

为了能对海龟和瓦片做更多的颜色改变, 也就是车辆和背景, 我们需要了解 NetLogo 如何处理颜色。

在 NetLogo 所有颜色对应一个数值。在这些练习里我们使用了颜色名, 只是因为 NetLogo 认识 16 个不同的颜色名。这并不意味 NetLogo 只能分辨 16 种颜色, 这些颜色之间的中间色也可使用。下面是 NetLogo 颜色空间的一张图:



为得到一个没有名字的颜色，你需要使用一个数值，或者在颜色名上加上或减去一个数。例如，输入 `set color red` 与输入 `set color 15` 效果完全一样。要得到一个更浅或更深的颜色，只需使用一个比该颜色更小或更大的一个数。如下所示：

- 在命令中心的弹出菜单选 "patches" (或使用 tab 键).
- 输入 `set pcolor red - 2` ("-" 两侧的空格很重要)

通过在 `red` 上减去一个数，得到更深的颜色。

- 输入 `set pcolor red + 2`

通过在 `red` 上加上一个数，得到更浅的颜色。

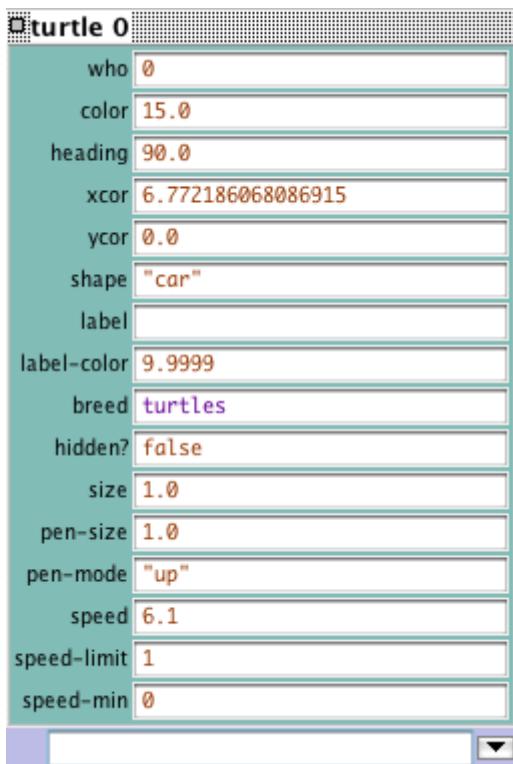
图上任何颜色均可采用这种方法。

主体监视器（Agent Monitors）和主体命令器（Agent Commanders）

上面我们使用 `set` 命令改变所有车辆的颜色。你是否记得，最初的模型中一群蓝车里有一辆红车。现在看看怎样只改变一辆车的颜色。

- 按下 "setup" 让红车再次出现
- 如果使用 Macintosh, 按下 Ctrl 在红车上单击。其他操作系统的话, 在红车上右击
- 如果有别的海龟与红车太近, 你可能看到在菜单底部列出多个海龟。将鼠标移动到海龟菜单项上, 注意到菜单和视图中海龟都加亮了。在红色海龟项的子菜单中选 "inspect turtle"。

关于那辆车的一个海龟监视器出现了:



仔细看看海龟监视器, 可以看到属于红车的所有变量。变量是存储数值的, 可以改变。还记得我们说过颜色在计算机里都是用数字表示的吗? 对主体也一样。例如, 每个主体都有一个 ID 号, 叫做"who number"。

再看看海龟监视器。

海龟的 who number 是多少?

海龟颜色是什么?

海龟形状是什么?

这个监视器显示该海龟的 who number 是 0, 颜色 15 (红色), 形状是“ car”。

除了右击 (mac 上是Ctrl+单击) 外, 还有两个方法打开海龟监视器。方法 1 是从Tools 菜单选择“Turtle Monitor”, 然后在 “who” 域中输入要查看的海龟的ID, 回车。另一种方法是在命令中心输入inspect turtle 0 (或其他ID)。

要关闭海龟监视器, 只需在窗口左上角关闭标志 (Macintosh) 或右上角关闭标志 (其他操作系统) 上单击。

现在了解了主体监视器, 有三种方式改变一个海龟的颜色。

第一种是使用主体监视器底部的主体命令器 (Agent Commander)。在这输入命令, 就

像在命令中心一样，只是在这输入的命令只由这个海龟执行。

- 在海龟 0 主体监视器中的主体命令器 中输入 set color pink.

视图发生什么变化?

海龟监视器有什么变化吗?

第二种是直接改变海龟监视器中的 color 变量

- 在海龟监视器中选择 "color" 右侧的文本.
- 输入新颜色，如green + 2.

发生什么?

第三种是使用观察者(observer)改变海龟或瓦片的颜色。因为 observer 俯视着 NetLogo 世界，它可以发出命令，影响单个或一组海龟。

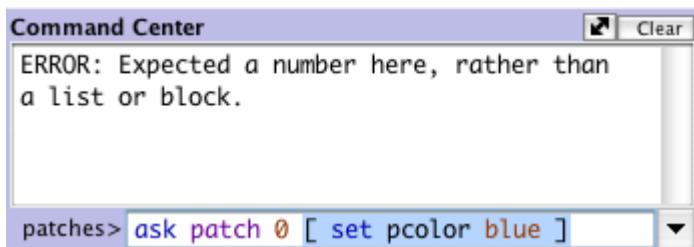
- 在命令中心的弹出菜单，选择 "observer" (或使用 tab 键).
- 输入 ask turtle 0 [set color blue]，回车

发生什么?

除了海龟监视器 (Turtle Monitors)，也有瓦片监视器 (Patch Monitors)。瓦片监视器与海龟监视器很相似。

你能使用瓦片监视器改变单个瓦片的颜色吗?

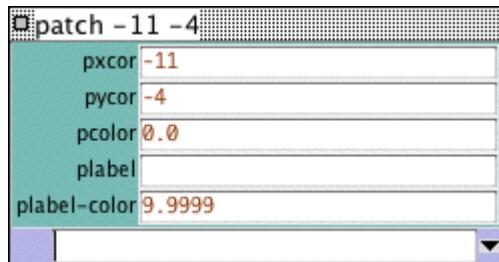
如果你让观察者 (observer) 请求瓦片 0 改变颜色 (ask patch 0 [set pcolor blue])，会出现一条错误信息。



要让某个海龟做什么，我们使用 who number。但瓦片没有 who number，需要其他方法。

记住，瓦片存在于一个坐标系统中。要在图上画个点需要两个数：x 坐标和 y 坐标。瓦片的定位方式与此相同。

- 随便选一个瓦片，打开瓦片监视器



监视器表明这个瓦片的 `pxcor` 变量是 -11，`pycor` 是 -4。在坐标平面上，这个点处于左下象限。

使用坐标让这个特定的瓦片改变颜色。

- 在瓦片监视器的底部，输入 `set pcolor blue`, 回车。

在海龟或瓦片的监视器中输入命令只对这个海龟或瓦片管用。

在命令中心也可操作单个瓦片：

- 在命令中心输入 `ask patch -11 -4 [set pcolor green]`，回车

下一步？

此时也许你想打开模型库的其他模型，试试刚学的这些技术。

在 [教学#3:例程](#) 你将学习怎样替换、修改已有模型，或构建自己的模型。

教学 #3: 例程 (Tutorial #3:Procedures)

本教程带你一步步建立一个完整的模型，对每一步做出解释。

主体和例程

在教学#2 你学习了怎样使用命令中心和主体监视器查看和修改主体，让他们执行动作。现在准备进入 NetLogo 模型的真正核心：例程页。

你已经使用了 NetLogo 中可以执行命令的主体类型：瓦片、海龟、链和观察者。瓦片是静止的，组成网格。海龟在网格上移动，链链接两个海龟。观察者俯视在进行的所有事情，做那些海龟、瓦片和链自己不能做的事情。

所有这四种主体都能执行 NetLogo 命令。前三种主体还能运行例程 (procedures)。一个例程包括一系列 NetLogo 命令，你将它们定义为一个单一的新命令。

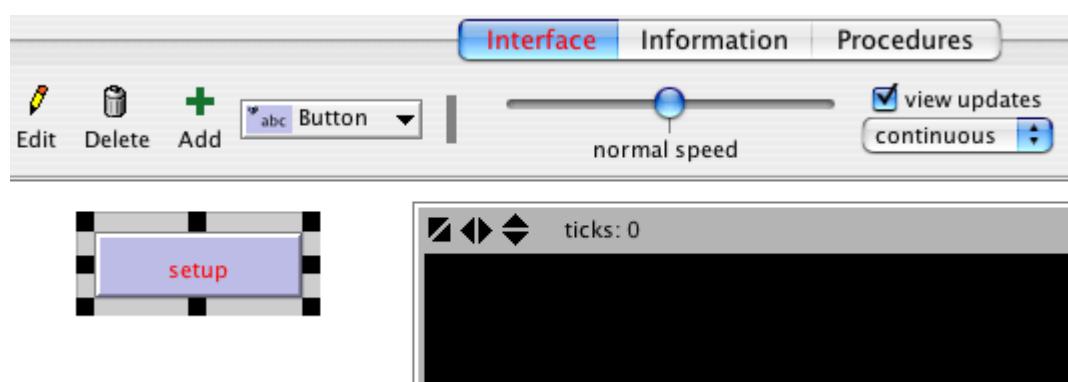
你将学习编写例程，让海龟移动、进食、繁殖和死亡。还将学习如何制作监视器、滑动条和绘图。我们要建立一个简单的生态系统模型，与教学#1 的狼吃羊模型部分相似。

制作 setup 按钮

要开始一个新模型，在 File 菜单中选择 New。然后从创建一个 setup 按钮开始：

- 在界面页上部的工具条上单击 "Button" 图标
- 在界面页的空白区域，定位到你想放置按钮的地方单击
- 编辑按钮的对话框出现了。在标签为"Commands"的文本域中输入setup
- 按下 OK 按钮，对话框关闭

现在有了一个 setup 按钮。按下按钮就执行一个名为"setup"的例程。例程就是一系列的 NetLogo 指令，我们给定一个新名字。现在还没有定义例程（一会就做）。由于按钮指向的例程现在还不存在，按钮变成了红色：



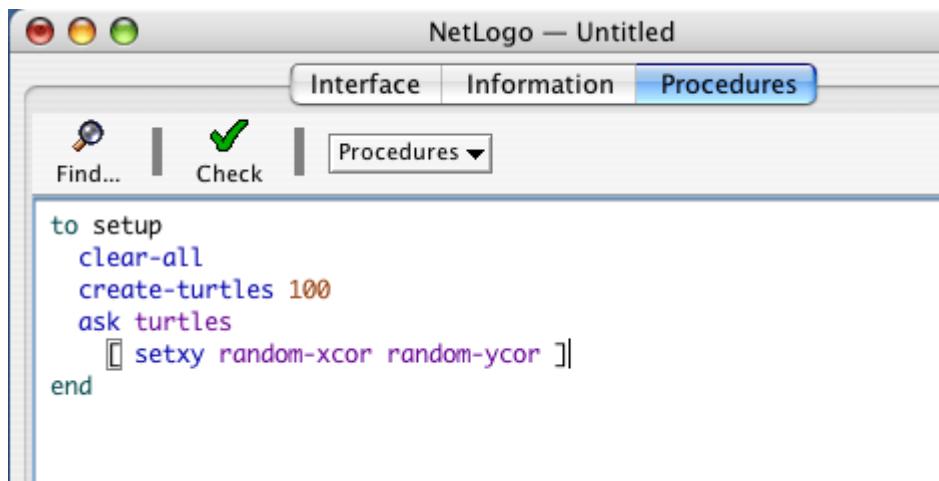
要想看实际的错误信息，单击该按钮。

现在创建“setup”例程，这样错误信息就会消失。

- 切换到 Procedures 页
- 输入下面的代码：

```
to setup
  clear-all
  create-turtles 100
  ask turtles [ setxy random-xcor random-ycor ]
end
```

做完后，Procedures 页就像下面的样子：



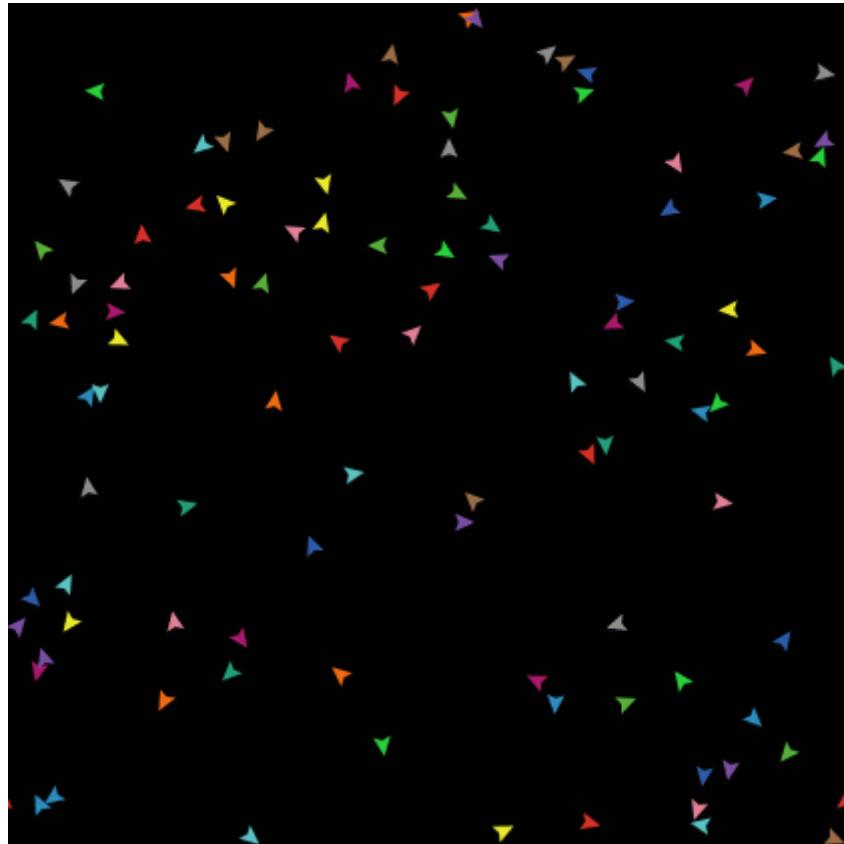
注意每行缩进量不同。多数人觉得代码这样缩进很有用，但这不是强制的，只是使得代码易读易改而已。例程以 [to](#) 开始，以 [end](#) 结束。所有的例程都要用这两个词开始和结束。

看看你输入了什么，每行是干什么的？

- [to setup](#) 开始定义一个名为 "setup" 的例程
- [clear-all](#) 将世界重设为初始、全空状态。所有瓦片变黑，你已经创建的海龟消失。基本上是将过去一笔勾销，为新模型运行做好准备。
- [create-turtles 100](#) 创建 100 个海龟。这些海龟都在原点，即瓦片 0,0 的中心。
- [ask turtles \[... \]](#) 告诉每个海龟独立地去运行方括号中的命令 (NetLogo 中每条命令都是由某些主体执行的。[ask](#) 也是一条命令。在这里是 observer 运行这条 [ask](#) 命令，这条命令又引起海龟运行命令)
- [setxy random-xcor random-ycor](#) 是一条使用 "reporters" 的命令。reporter 与命令不同，它只报告一个结果。首先每个 turtle 运行 reporter [random-xcor](#)，它返回 X 坐标范围内的一个随机数，然后每个 turtle 运行 reporter [random-ycor](#)，返回 Y 坐标范围的一个随机数。最后每个 turtle 使用前面的两个数做输入参数运行 [setxy](#) 命令，这使得 turtle 移动到相应的坐标处。

- `end` 结束"setup" 例程的定义。

输入完成后，切换到界面页，按下前面制作的 setup 按钮，你将看到海龟分散在世界内：



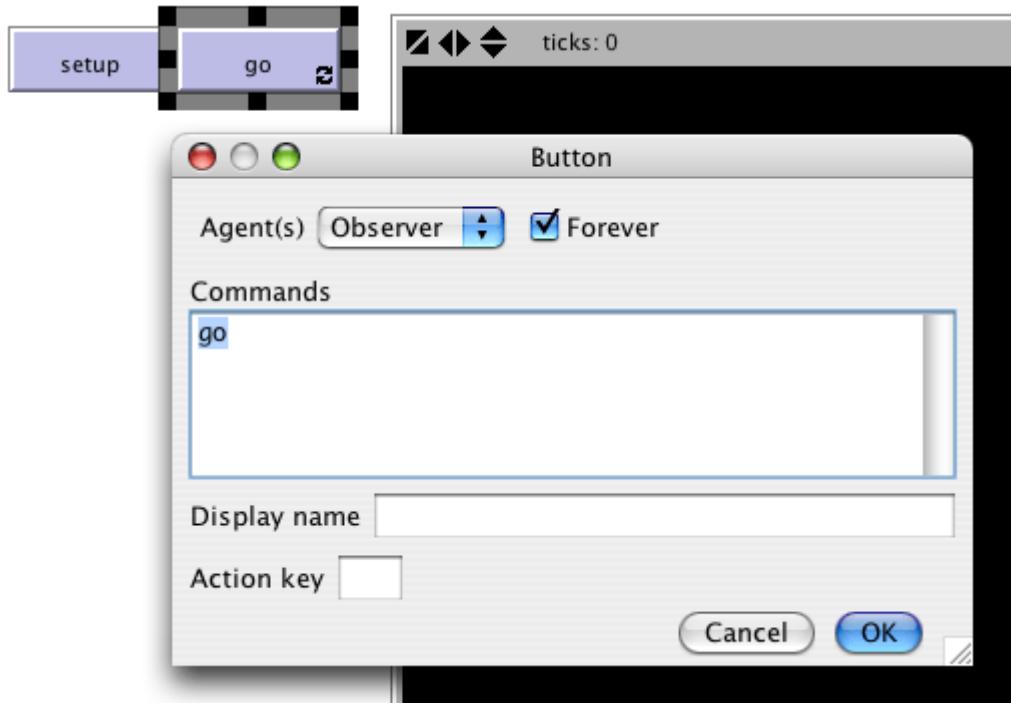
多按 setup 几次，看看海龟的散布有何不同。 注意有些海龟会叠压在一起。

稍微想想要达到这样的结果需要做哪些事情。你要在界面页里做一个按钮，还要创建该按钮要使用的一个例程。只有这两步都做了，按钮才能工作。在本教程的剩余部分，经常需要做类似两步或更多步，为模型增添新功能。在增加新功能时，当你觉得该做的步骤都完成了，但就是不能正常工作时，那就继续向前读，看看是否还要其他步骤。读过几节后，返回来看看是否遗漏了某个步骤。

制作 go 按钮

现在做个 "go" 按钮。步骤与 setup 按钮的一样，除了下面几点：

- 在命令部分输入 `go`，而非`setup`.
- 在编辑对话框里勾选 "forever"



“forever” 勾选项使得按钮按下后保持按下状态，因此命令会不断重复执行，而不是只执行一次。

- 在 Procedures 也增加 go 例程:

```
to go
  move-turtles
end
```

move-turtles是什么？它是一个象[clear-all](#)那样的原语（NetLogo内嵌的）吗？非也，它是一个需要你添加的例程。至今你已经有两个自己添加的例程了：setup 和go

- 在go例程后面增加move-turtles 例程:

```
to go
  move-turtles
end

to move-turtles
  ask turtles [
    right random 360
    forward 1
  ]
end
```

注意move-turtles中间的连字符两侧没空格。教学#2 中用过的red - 2 有空格，是为了做减法操作。此处我们要的是move-turtles，没空格。“-”将“move”和“turtles”组成一个词。

例程move-turtles中的命令：

- `ask turtles [...]` 每个turtle运行[]中的命令
- `right random 360` 是使用reporter的命令。首先每个turtle 在 0 和 359 之间随机选一个整数 (`random` 不会返回你给它的数)，然后turtle 右转这个度数。
- `forward 1` 让turtle前进1步。

为什么我们不把所有这些命令都写在 go 例程里，而是分为几个例程？确实可以这样做，但是在创建你的项目的过程中，你很可能会增加更多的东西。最好保持 go 例程尽量简单，这样更容易理解。最后还会包括许多其他的东西，比如计算，画图等。这些事情由相应的例程完成，每个例程有各自的名字。

界面页中的 go 按钮是永久性的，意味着将不断执行命令，直到你关掉它（重新单击它）。按下 setup 按钮，创建海龟，然后按下 go 按钮。观察模型。关闭它，你会看到所有海龟停住了。

注意当海龟越过世界边缘时，它要回绕（wrap），即出现在另外一边。（这是默认行为，可以改变，详情参见编程指南的[Topology](#)部分。）

试试命令

我们建议你试试其他海龟命令。

在命令中心输入命令（如 `turtles > set color red`），或在setup, go, move-turtles 中添加命令。

注意在命令中心输入命令时，你必须使用弹出菜单选择 `turtles >`, `patches >`, or `observer >`，具体选择取决于哪个主体将执行命令。这就像 `ask turtles` 或 `ask patches` 一样，只是不用打字而已。你可以使用 tab 键在主体类型之间切换，这比用菜单更方便。

可以试试在命令中心输入 `turtles > pen-down`，然后按下 go 按钮。

在move-turtles例程中，试试将 `right random 360` 改为 `right random 45`。

玩吧。很容易，并且结果立现——这是 NetLogo 许多优点之一。

试验够了吧，准备继续改进模型。

瓦片和变量

现在我们有 100 个海龟，它们漫无目的的移动，对周围的事物毫无知觉。下面我们给海龟一个好点的背景，让模型稍微有趣一些。

- 回到 setup 例程，修改例程如下：

`to setup`

```
clear-all  
setup-patches  
setup-turtles  
end
```

- 新的 setup 引用了两个新例程，定义 setup-patches：

```
to setup-patches  
  ask patches [ set pcolor green ]  
end
```

例程 setup-patches 将开始时所有瓦片颜色定义为绿色。（海龟的颜色变量是 [color](#), 瓦片的是 [pcolor](#)）

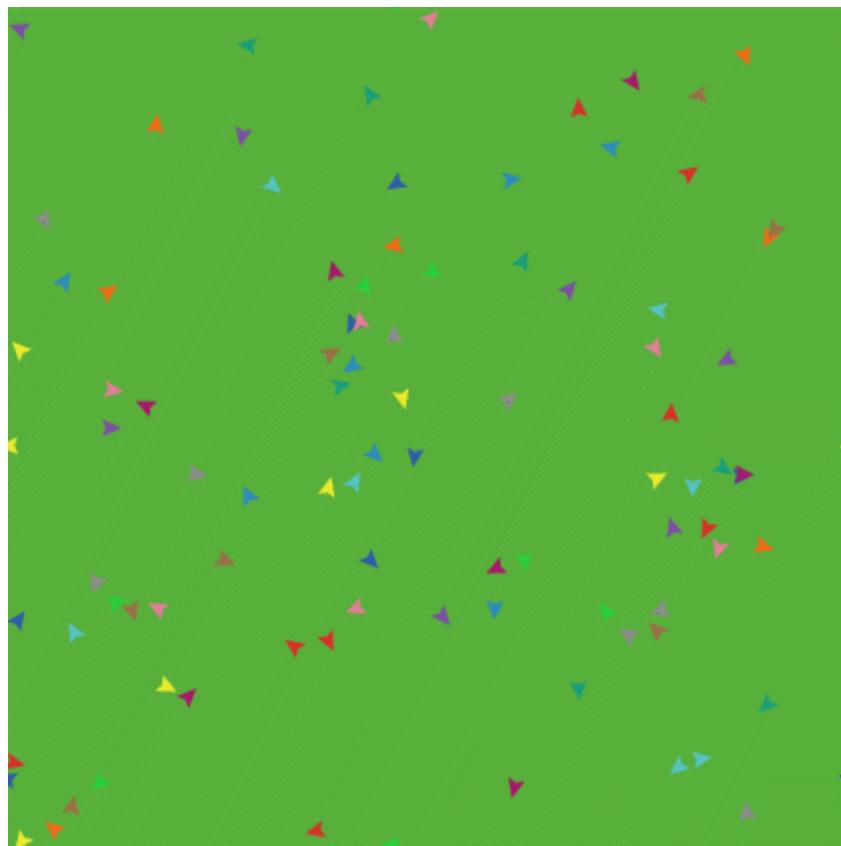
- 加上这个例程

```
to setup-turtles  
  create-turtles 100  
  ask turtles [ setxy random-xcor random-ycor ]  
end
```

你是否注意到新的 setup-turtles 与老的 setup 包含许多相同的命令？

- 切换回 Interface 页
- 按下 setup 按钮

瞧！由海龟和绿色瓦片构成的 NetLogo 风景：



看过新setup例程的效果后，你会发现重新读读这个例程很有帮助。

海龟变量

目前海龟可以在地表上移动，但什么都不做。现在在海龟和瓦片之间加上一些交互。

我们让海龟吃“草”（绿色瓦片），繁殖、死亡。草被吃掉后要逐渐恢复。

我们需要一种控制海龟繁殖和死亡的方式。我们通过跟踪海龟有多大能量 (energy) 来决定。要这样的话需要增加一个新的海龟变量。

你已经见过一些内置的海龟变量，如[color](#)。要添加新的海龟变量，需要在例程页的顶部加上[turtles-own](#) 声明，这个声明必须在所有例程之前，变量名为energy：

```
turtles-own [energy]
```

```
to go
  move-turtles
  eat-grass
end
```

使用这个新定义的变量(energy)，允许海龟吃草。

- 切换到 Procedures 页
- 重写 go 例程如下：

```
to go
  move-turtles
  eat-grass
end
```

- 加上新例程eat-grass：

```
to eat-grass
  ask turtles [
    if pcolor = green [
      set pcolor black
      set energy (energy + 10)
    ]
  ]
end
```

我们第一次使用`if` 命令，仔细看看代码。当每个海龟执行这些命令时，比较它所处的瓦片的颜色(`pcolor`)与绿色是否相同。（海龟能直接访问所处瓦片的变量），如果瓦片是绿色则返回true，这时才执行[]中的命令（否则跳过）。这些命令让海龟将瓦片改为黑色，海龟能量值增加 10。瓦片变黑表明该处的草被吃掉，因为吃了草，海龟能量增加。

下面，让海龟移动时消耗一些能量：

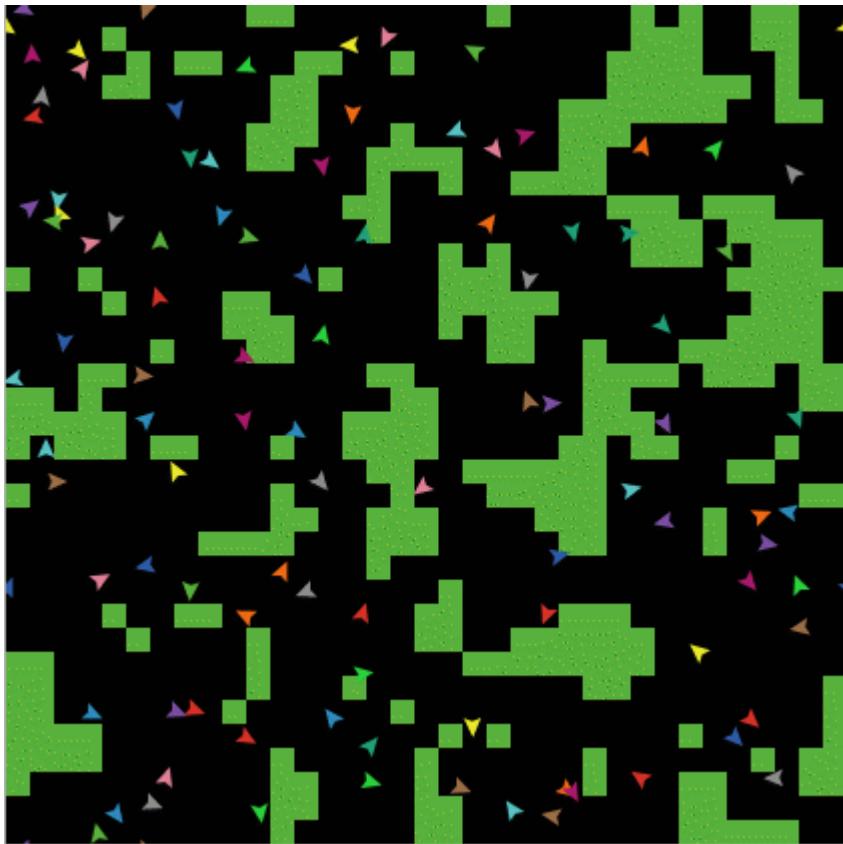
- 重写move-turtles如下：

```
to move-turtles
  ask turtles [
    right random 360
    forward 1
    set energy energy - 1
  ]
end
```

当海龟移动时，每步减少 1 个单位的能量。

- 切换到 Interface 页，按下 setup 和 go 按钮

你将看到当海龟走到瓦片上时，瓦片变为黑色。



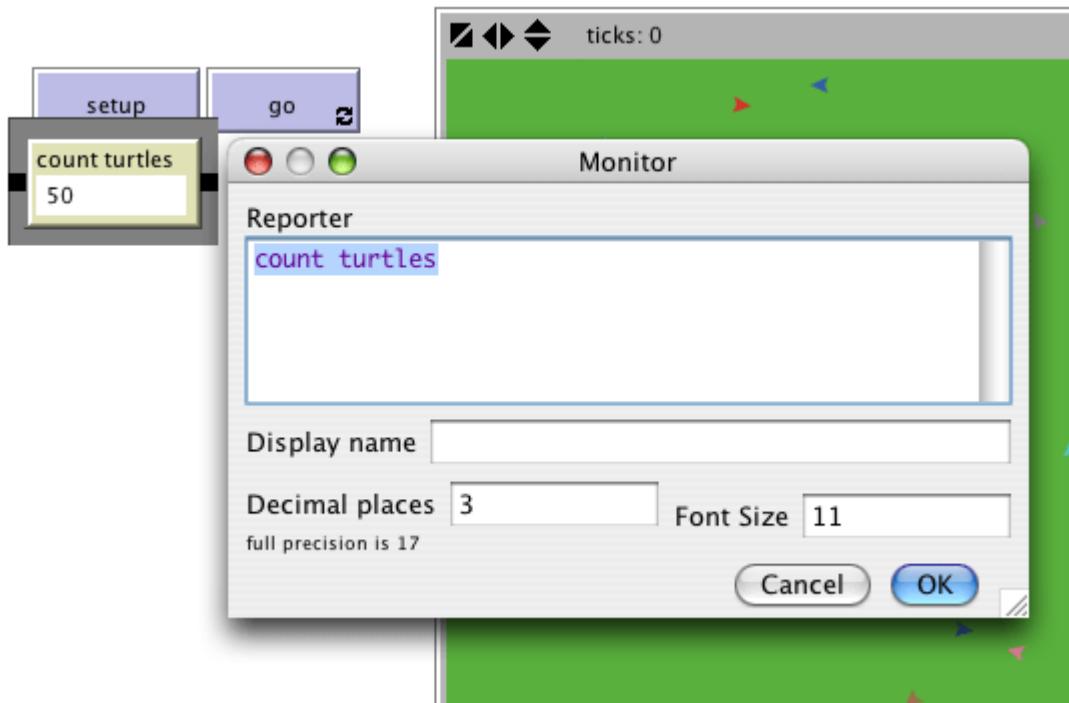
监视器 (Monitors)

下面使用工具条在 Interface 页中创建 2 个监视器。（像使用按钮、滑动条一样，使用工具条上的监视器图标）。先做第一个监视器。

- 使用工具条上的监视器图标，在界面页空白处创建一个监视器。

出现对话框

- 在对话框中输入: count turtles (见下图).
- 按 OK 关闭地对话框



turtles 是一个“agentset”，即所有海龟的集合。count 告诉我们这个集合中有多少主体。

制作第二个监视器：

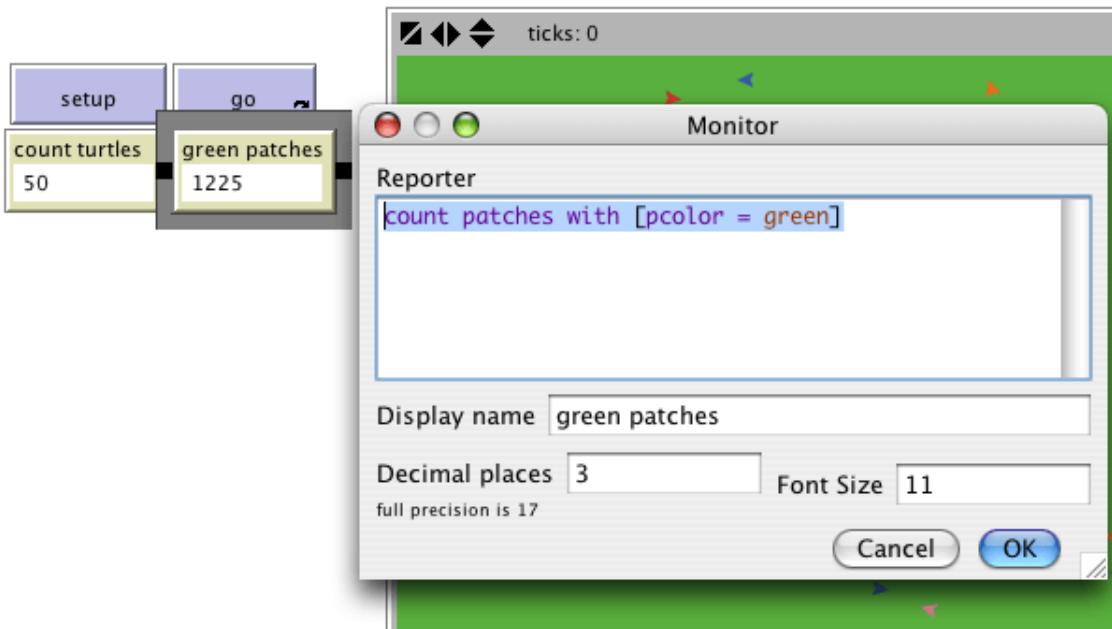
- 使用工具条上的监视器图标，在界面页空白处创建一个监视器。

出现对话框。

在对话框的Reporter部分输入：count patches with [pcolor = green] (见下图)。

在 Display name 部分输入：green patches

按 OK 关闭对话框



此处我们再次使用[count](#)查看一个agentset中有多少主体。[patches](#)是所有瓦片的集合，但我们并不想知道总共有多少瓦片，而是想知道有多少绿色的。这就是[with](#)要做的，它创建一个较小的agentset，只有满足[]中的条件的主体才会包含进来，条件是pcolor = green，因此得到的是绿色瓦片。

现在有两个监视器报告有多少海龟，有多少绿色瓦片，帮助我们跟踪模型运行。模型运行时，监视器中的数字自动变化。

使用 `setup` 和 `go` 按钮，观看监视器数值的变化。

开关和标签（Switches and labels）

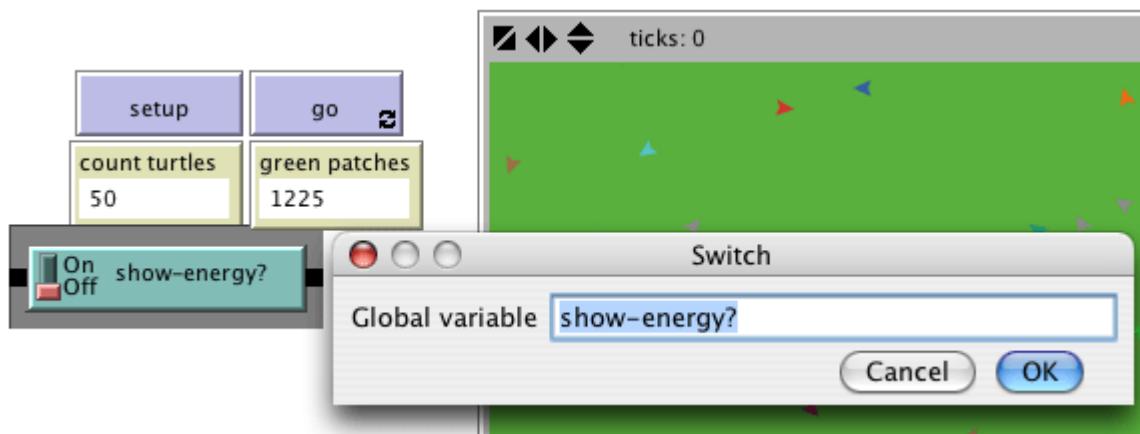
海龟不仅是将瓦片变黑，它们还获得、损失能量。模型运行时试试使用海龟监视器查看一个海龟的能量变化。

如果能在任何时候看到所有海龟的能量就更好了。现在就这样做，并且增加一个开关能控制这些额外信息显示与否。

- 在界面页 的工具条上选择开关图标，在空白处单击，创建一个开关。

出现对话框。

- 在对话框的Global variable 部分输入 `show-energy?` 别忘了包括问号(见下图)



- 返回 'go' 例程
- 重写 eat-grass 例程如下：

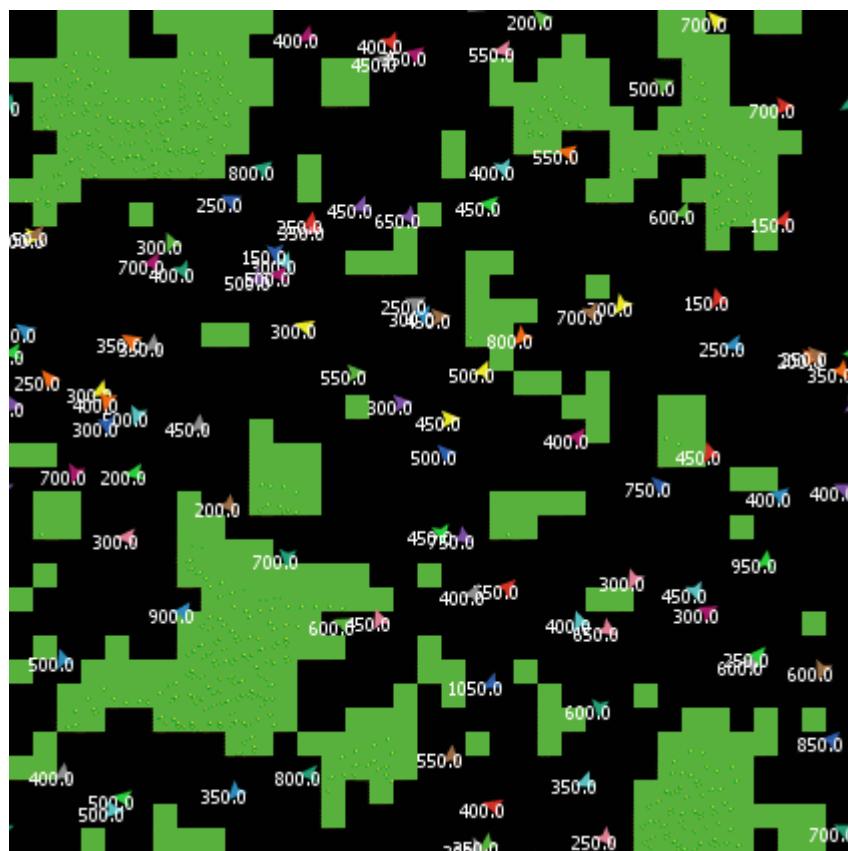
```
to eat-grass
  ask turtles [
    if pcolor = green [
      set pcolor black
      set energy (energy + 10)
    ]
    ifelse show-energy?
      [ set label energy ]
      [ set label "" ]
  ]
end
```

例程eat-grass用了`ifelse`命令，仔细看代码。每个海龟当运行这些新命令时检查`show-energy?`的值（由开关决定）。如果开关打开，比较结果为true，海龟执行第一个[]中的命令。这时将能量值赋给海龟标签。如果比较结果为false（开关关闭），海龟执行第二个[]中的命令，这时移去文本标签（通过将海龟标签设为空）。

（在NetLogo中几个字符称为字符串(string)。字符串是在双引号之间的一串字母和字符。此处两个双引号之间什么也没有，这是一个空串。如果海龟的标签是空串，就表示没有任何文本。）

- 测试一下。在界面页中运行模型（使用`setup`和`go`），来回拨动`show-energy?`开关。

当开关打开时，能看到海龟的能量因吃草而增加，移动时减小



更多例程

现在海龟在吃草，再让它们繁殖和死亡，也让草能恢复。现在增加三个例程，分别负责这三种行为。

- 切到 Procedures 页
- 重现 go 例程如下：

```
to go
  move-turtles
  eat-grass
  reproduce
  check-death
  regrow-grass
end
```

- 增加例程reproduce, check-death 和regrow-grass 如下：

```
to reproduce
  ask turtles [
    if energy > 50 [
      
```

```

    set energy energy - 50
    hatch 1 [ set energy 50 ]
]
]

end

to check-death
ask turtles [
  if energy <= 0 [ die ]
]
end

to regrow-grass
ask patches [
  if random 100 < 3 [ set pcolor green ]
]
end

```

这些例程都使用了[if](#)命令。当繁殖时，每个海龟检查它的energy值，如果大于 50 执行第一个[]中的命令。在这里energy减少 50，然后孵出（hatches）一个energy为 50 的新海龟。[hatch](#)命令是NetLogo的一个原语，形如hatch *number* [*commands*]，海龟创建*number*个新海龟，每个都与母体相同，并且请求这些新海龟执行*commands*，你可以使用*commands*让这些新海龟有不同的颜色、方向等。这里运行一条命令，将新海龟的energy设为 50。

当每个海龟运行check-death时，它检查energy是否小于等于 0。如果是真，则海龟被告知去死[die](#)（这是NetLogo的一个原语）。

当每个海龟运行regrow-grass 时，它检查随机产生的 0-99 之间的整数是否小于 3。如果是，瓦片颜色设为绿。对每个瓦片来说，发生的次数（平均）是 3%，因为在 100 个可能的数中，有三个数（0, 1, 2）小于 3。

- 切换到 Interface 页，按下 setup 和 go

现在能看到模型的一些有趣行为。一些海龟死掉，一些新海龟出现（孵出），一些草恢复。这正是我们要做的。

如果继续看模型的监视器，会发现 **count turtles** 和 **green patches** 监视器都有振荡。振荡模式能预测吗？这些变量之间有关系吗？

如果有更容易的跟踪模型行为的方式就更好了。NetLogo 可以为我们画图，这是下面要讲的。

画图 (Plotting)

要想画图的话，需要在界面页创建一个 plot，做一些设置。然后在例程页增加一些例程，这些例程为我们更新绘图。

先做例程页中的工作。

- 修改 setup，调用将要增加的新例程do-plots

```
to setup
  clear-all
  setup-patches
  setup-turtles
  do-plots
end
```

- 还要修改 go，调用do-plots

```
to go
  move-turtles
  eat-grass
  reproduce
  check-death
  regrow-grass
  do-plots
end
```

- 增加新例程。我们要画的是海龟的数量和绿瓦片的数量。在每个时间步 (go 例程的一次运行)把这些值加到图中。

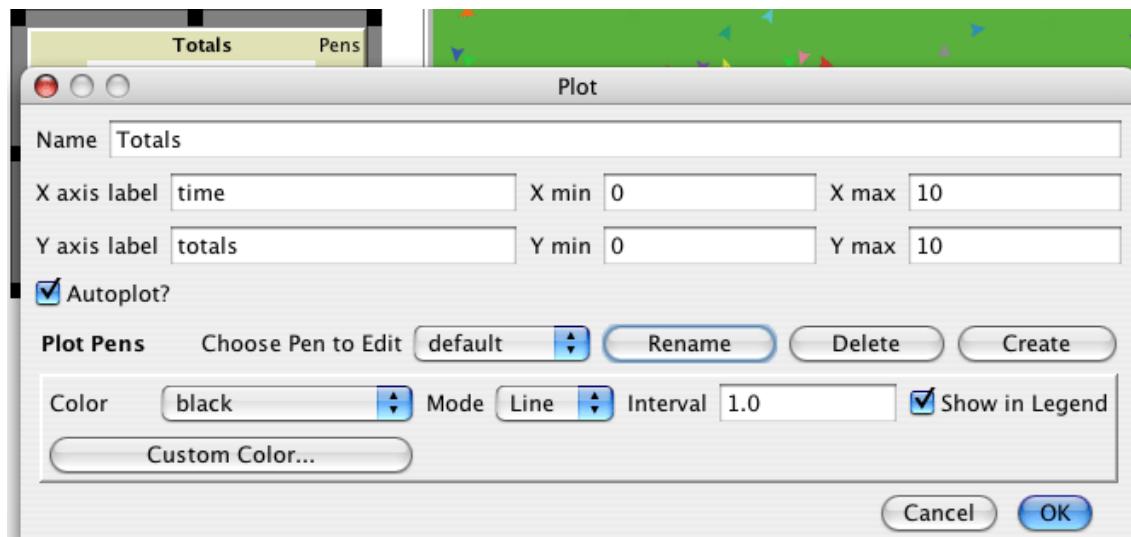
```
to do-plots
  set-current-plot "Totals"
  set-current-plot-pen "turtles"
  plot count turtles
  set-current-plot-pen "grass"
  plot count patches with [pcolor = green]
end
```

注意使用[plot](#)命令在图上增加新点。然而在这样做之前，需要告诉NetLogo两件事。第一，需要指明要使用哪个图（因为后面的模型有多个图）。第二，要指定使用哪支笔（pen）画图（本图使用两支笔）。

[plot](#)命令告诉画笔移动到新点，新点的X坐标是先前的X坐标加1，Y坐标就是plot命令中给定的值（第一种情况是海龟数量，第二种情况是绿瓦片数量）。当画笔移动时就画出线。

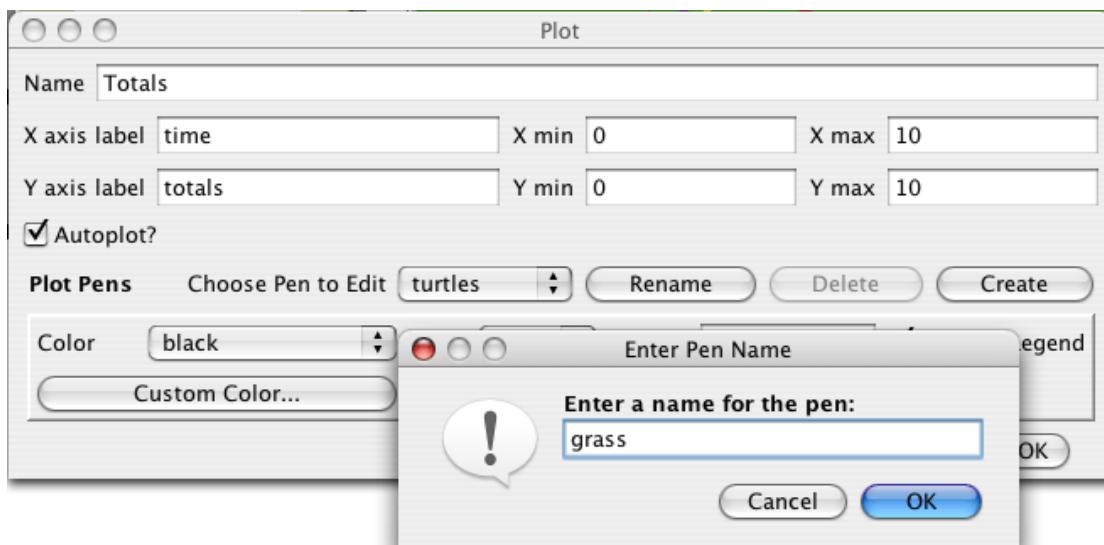
为了让 set-current-plot "Totals" 工作，必须在界面页中为你的模型增加一个图（plot），然后进行编辑，让它的名字与例程中用到的相同。即使名字中多个空格也会出错---两处必须完全相同。

- 在界面页上使用 plot 图标创建一个 plot
- 设置图名为"Totals" (见下图)
- 设置 X 轴标签为"time"
- 设置 Y 轴标签为"total"



下一步创建两支笔（pen）。

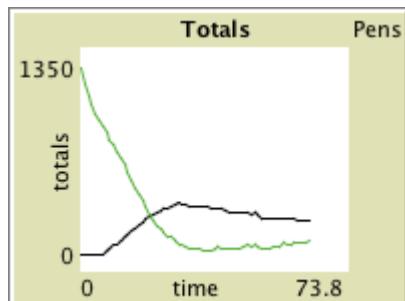
- 在 Plot 对话框中，按下'Create' 按钮创建一支新笔
- 输入笔的名字"turtles"，在"Enter Pen Name"对话框中按下 OK (见下图)
- 在 Plot 对话框中，按下'Create' 按钮创建第二支新笔
- 输入笔的名字"grass" ，在"Enter Pen Name"对话框中按下 OK (见下图)
- 选择笔的颜色，变为 green.
- 在 Plot 对话框中选择 OK



注意在创建图的时候，也可设置 X, Y 轴的最大最小值。让“Autoplot?”勾选着，如果图超过坐标轴的设定范围，坐标轴会自动增长，使你能看到所有数据。

- 再次按 Setup 和 go，运行模型

你会看到模型运行时就能绘图。图的样子与下图相似，尽管可能不完全一样。
记住我们让“Autoplot?”打开。这使得没有空间时，图能自动调整。



如果你忘了哪支笔是干什么的，单击图的右上角处 Pens 标签。你试试运行模型几次，看看这些图哪些部分相同哪些不同。

时钟计数器（Tick counter）

在比较同一模型多次运行得到的图时，如果每次运行长度相同会很有用。学会怎样让模型在特定的时刻停止或启动很有帮助，因为我们可以让模型在同一时刻停止。跟踪 go 例程运行了多少次是实现这一技术的关键。

要跟踪这一点，使用 NetLogo 内建的时钟计数器（tick counter）

- 修改 go 例程

```

to go
  if ticks >= 500 [ stop ]
  move-turtles
  eat-grass
  reproduce
  check-death
  regrow-grass
  tick
  do-plots
end

```

- 按下 `setup`，再运行模型

图形和模型不会一直运行下去，当界面页工具条上的时钟计数器达到 500 时，模型自动停止。

`tick`命令将时钟计数器加 1，`ticks`是一个报告器，返回时钟计数器当前值。每次重新运行时`clear-all`将时钟计数器清 0.

注意`tick`在`do-plots`之前。如果绘图代码使用时钟计数器就要这样，它会查看新值，而非旧值。（本教程实际没写这样的代码，但是一般说来将`tick`放在主体动作之后、绘图之前是个好做法）

现在模型使用了 `ticks`，你可能想使用界面页顶部的菜单从连续更新（continuous updates）变为基于时钟更新（“tick-based” updates）。这意味着 NetLogo 仅在时钟点上更新（重画）视图（主体显示区），`tick` 中间不更新。这样模型运行的更快一些，并保证稳定观感（因为更新的时间间隔固定）。关于视图更新的全部讨论见编程指南。

更多细节

首先，可以有可变数量的海龟，而非总是 100 个。

- 增加一个滑动条 'number'，改变最大最小值
- 在例程 `setup-turtles`中，不使用`create-turtles 100`，而是：

```

to setup-turtles
  create-turtles number
  ask turtles [ setxy random-xcor random-ycor ]
end

```

测试一下。比较初始时刻海龟更多或更少时图的变化。

其次，如果能调整海龟吃草获得的能量和移动时消耗的能量不是更好吗？

- 增加滑动条energy-from-grass。
- 增加滑动条birth-energy
- 修改eat-grass例程：

```
to eat-grass
  ask turtles [
    if pcolor = green [
      set pcolor black
      set energy (energy + energy-from-grass)
    ]
    ifelse show-energy?
      [ set label energy ]
      [ set label "" ]
  ]
end
```

- 修改 reproduce：

```
to reproduce
  ask turtles [
    if energy > birth-energy [
      set energy energy - birth-energy
      hatch 1 [ set energy birth-energy ]
    ]
  ]
end
```

最后，如果要改变草的恢复率应该增加什么滑动条？能增加什么海龟移动规则，或让孵化只发生在特定时刻，试着写出来。

下一步？

现在有了一个简单生态系统模型。瓦片长草，海龟移动、吃草、繁殖、死亡。你创建了界面，包括按钮、滑动条、开关、监视器和绘图。甚至写了一些例程让海龟做事。

教学到此结束。

如果要看更多的NetLogo文档，界面指南[Interface Guide](#)让你了解NetLogo所有界面元素的功能。要写例程，看编程指南[Programming Guide](#)。所有原语在NetLogo词典[NetLogo Dictionary](#)。

如果愿意，你还可以继续试验、扩展本模型，试验主体不同的变量和行为。

另外，还可以再看教学#1 的狼吃羊模型。你看到羊走来走去，消耗资源，资源随机补充，一定条件下繁殖，没有资源时死亡。该模型还有另一类生物—狼。增加狼需要另外的例

程及新的原语。狼和羊是两种不同的物种（breeds），研究该模型，了解怎样使用 breeds。

另外，还可以查看其他模型（包括模型库中的 Code Examples），甚至继续建立自己的模型。你甚至不必建模，只是看着瓦片和海龟形成模式就很有趣，或试着创建个游戏玩玩，等等。

希望你学到了一些东西，包括 NetLogo 语言及如何建模。上面创建的所有例程列在下面。

附录：完整代码

这些完整代码在 NetLogo 模型库里也有，在 Code Examples 部分，名为 “Tutorial 3”。

注意代码有注释，注释由分号开始。使用注释帮助你理解模型。

在例程页，注释是灰的，容易区别。

```
turtles-own [energy] ;; for keeping track of when the turtle is ready
                    ;; to reproduce and when it will die

to setup
  clear-all
  setup-patches
  setup-turtles
  do-plots
end

to setup-patches
  ask patches [ set pcolor green ]
end

to setup-turtles
  create-turtles number    ;; uses the value of the number slider to create turtles
  ask turtles [ setxy random-xcor random-ycor ]
end

to go
  if ticks >= 500 [ stop ]  ;; stop after 500 ticks
  move-turtles
  eat-grass
  reproduce
  check-death
  regrow-grass
  tick           ;; increase the tick counter by 1 each time through
  do-plots
end
```

```

to move-turtles
  ask turtles [
    right random 360
    forward 1
    set energy energy - 1 ;; when the turtle moves it loses one unit of energy
  ]
end

to eat-grass
  ask turtles [
    if pcolor = green [
      set pcolor black
      ;; the value of energy-from-grass slider is added to energy
      set energy (energy + energy-from-grass)
    ]
    ifelse show-energy?
      [ set label energy ] ;; the label is set to be the value of the energy
      [ set label "" ]     ;; the label is set to an empty text value
    ]
  ]
end

to reproduce
  ask turtles [
    if energy > birth-energy [
      set energy energy - birth-energy ;; take away birth-energy to give birth
      hatch 1 [ set energy birth-energy ] ;; give this birth-energy to the offspring
    ]
  ]
end

to check-death
  ask turtles [
    if energy <= 0 [ die ] ;; removes the turtle if it has no energy left
  ]
end

to regrow-grass
  ask patches [ ;; 3 out of 100 times, the patch color is set to green
    if random 100 < 3 [ set pcolor green ]
  ]

```

```
]  
end  
  
to do-plots  
  set-current-plot "Totals" ;; which plot we want to use next  
  set-current-plot-pen "turtles" ;; which pen we want to use next  
  plot count turtles ;; what will be plotted by the current pen  
  set-current-plot-pen "grass" ;; which pen we want to use next  
  plot count patches with [pcolor = green] ;; what will be plotted by the current pen  
end
```

界面指南 (Interface Guide)

手册的本部分带你依次了解 NetLogo 界面的所有要素，并解释其功能。

在 NetLogo 里你可以查看模型库中的模型，增加东西，或创建你自己的模型。NetLogo 界面设计用来满足所有需求。

界面分成两个主要部分：菜单和主窗口。主窗口分成标签页面。

- 菜单
- 标签页
- Interface 标签页
 - 使用界面元素
 - 2D 和 3D 视图
 - 命令中心
 - 绘图
- Infomation 标签页
- Procedures 标签页
- Includes 菜单

菜单

在 Macs，如果运行 NetLogo 应用程序，菜单条在屏幕顶部。在其他平台，菜单条在 NetLogo 窗口的顶部。



菜单条上的各菜单的功能如下表所列。

表：NetLogo 菜单

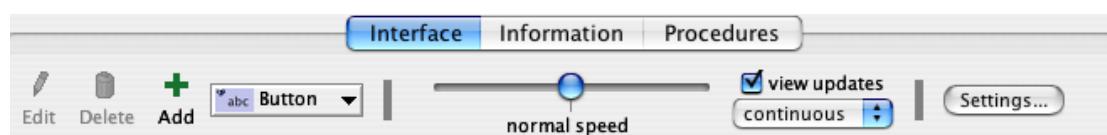
File		
	New	创建一个新模型
	Open	在计算机上打开任何一个模型
	Models Library	演示模型的集合
	Save	保存当前模型
	Save As	使用一个其他的名字保存当前模型
	Save As Applet	用来保存一个 HTML 格式的网页，其中以 Java applet 嵌入你的模型
	Print	将当前显示的页面内容发送到打印机

	Export World	保存所有变量、海龟和瓦片的当前状态、画图、绘图、输出区域和随机状态信息到一个文件
	Export Plot	将绘图中的数据保存到文件
	Export All Plots	将所有绘图中的数据保存到文件
	Export View	将当前视图(2D 或 3D)作为图片保存到文件 (PNG 格式)
	Export Interface	将当前界面页作为图片保存 (PNG 格式)
	Export Output	将输出区域的内容或命令中心的输出部分保存到文件
	Import World	加载用 Export World 保存的文件
	Import Patch Colors	将一个图像加载到瓦片, 见 import-pcolors 命令
	Import Patch Colors RGB	使用 RGB 颜色 将一个图像加载到瓦片 ; 见 import-pcolors-rgb 命令
	Import Drawing	将一个图像加载到画图层, 见 import-drawing 命令
	Import HubNet Client Interface	将其他模型的界面加载到 HubNet Client Editor.
	Quit	退出 NetLogo。 (在 Mac, 这一项在 NetLogo 菜单)
Edit		
	Cut	剪切或移除选中的文本, 临时存到剪贴板
	Copy	复制选中的文本
	Paste	将剪贴板上的文本放到光标处
	Delete	删除选择的文本
	Undo	撤销上一个文本编辑操作
	Redo	重做上一个 undo 操作
	Select All	选择活动窗口中的所有文本
	Find	在信息页或例程页查找一个词或字符序列
	Find Next	查找下一处
	Shift Left / Shift Right	在例程页用于改变代码缩进层次
	Comment / Uncomment	在例程页使用, 在代码中增加或移去分号(分号作为注释标志)
	Snap To Grid	当激活时, 新部件停在 5 个像素宽的网格上, 这样容易对齐。 (注意: 当缩放时本功能失效)
Tools		
	Halt	停止所有代码的运行, 包括按钮和命令中心。(警告: 因为代码强制中断, 如果继续运行时没有用"setup"重新启动模型, 可能得到不期望的结果)
	Globals Monitor	显示所有全局变量的值
	Turtle Monitor	显示特定海龟的所有变量值。也可编辑海龟变量的值, 或向海龟发出命令。(也可以通过视图打开海龟监视器, 见下面的视图部分)
	Patch Monitor	显示特定瓦片的所有变量值。也可编辑瓦片变量的值, 或向瓦片发出命令。(也可以通过视图打开瓦片监视器, 见下面的视图部分)
	Link Monitor	显示特定链的所有变量值。也可编辑链变量的值, 或向链

		发出命令。(也可以通过视图打开链监视器, 见下面的视图部分)
	Hide/Show Command Center	使命令中心可见或不可见。(注意命令中心也可以用鼠标实现显示、隐藏、改变大小)
	3D View	打开 3D 视图。见下面的视图部分。
	Color Swatches	打开 Color Swatches。见编程指南的颜色部分
	Turtle Shapes Editor	画海龟图形。更多信息见图形编辑器指南。
	Link Shapes Editor	画链图形。更多信息见图形编辑器指南。
	BehaviorSpace	使用不同的设置重复运行模型。更多信息见行为空间指南
	System Dynamics Modeler	打开系统动力学建模工具。更多信息见系统动力学建模工具指南
	HubNet Client Editor	打开 HubNet Client Editor。更多信息见 HubNet 编程指南
	HubNet Control Center	如果没有 HubNet 活动则不能用。更多信息见 HubNet 指南
Zoom		
	Larger	增加模型的屏幕大小。在大的监视器或投影上有用。
	Normal Size	将模型的屏幕大小重设为正常大小
	Smaller	减小模型的屏幕大小
Tabs 本菜单提供了每个页的快捷键 (Mac 上是 Command 1 到 Command 3。在 Windows, 是 Control 1 到 Control 3。)		
Help		
	About NetLogo	显示关于当前 NetLogo 的版本信息(在 Mac, 该项在 NetLogo 菜单)
	Look Up In Dictionary	对命令或报告器在浏览器中打开相应词典条目
	NetLogo User Manual	在浏览器中打开本手册
	NetLogo Dictionary	在浏览器中打开词典

标签页

在 NetLogo 主窗口的顶部是三个标签页：Interface(界面)、Information(信息)和 Procedures(例程)，任一时刻只有其中之一可见，但可以通过单击窗口顶部的标签进行切换。



在这些标签下方是一个工具条，上面有一排按钮，当切换标签时会显示不同的按钮。

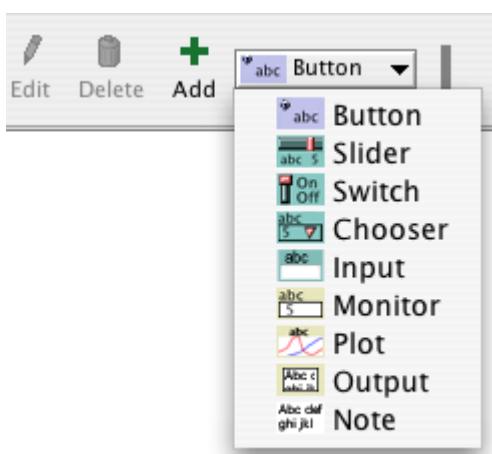
界面页

在界面页查看模型的运行，其中有工具用来监视和更改模型内部的运行情况。

当首次打开 NetLogo 时，界面页只有主视图和命令中心，主视图显示海龟和瓦片，命令中心用来发出 NetLogo 命令。

使用界面元素

界面页的工具条包括按钮，按钮用来编辑、删除、创建界面项，还有一个菜单用来选择不同的界面项（例如按钮和滑动条）。



工具条上的按钮如下所述。

添加: 要添加界面元素，首先在下拉菜单中选择所需元素，注意 Add 按钮呈按下状态，然后在工具条下方的空白区单击。（如果菜单项已经显示所需的类型，只需按下 Add 按钮，不用使用菜单）

选择: 要选择一个界面元素，用鼠标拖出一个矩形包围它。该元素会出现灰色边框，表明被选中了。

选择多项: 通过用拖出的矩形同时包围多个界面元素，可以选中多项。如果选择了多项，其中一项是“key”项，含义是如果使用界面页工具条上的“Edit”或“Delete”，则只影响“key”项。“key”项上是一个深灰色边框，以示区别。

取消选择: 要取消所选的所有元素，在界面页的空白处单击。要取消选择某个元素，Ctrl-单击（Macintosh）或右击（其他系统）该元素并在弹出菜单中选择“Unselect”。

编辑: 要改变一个界面元素的特性，选择该元素，按下界面页工具条的“Edit”按钮。也可以选择该元素后双击。

移动: 选中界面元素，用鼠标将它拖到新位置。如果拖动时按下 Shift 键，则只能做水平会或垂直移动。

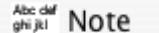
改变大小: 选中界面元素，用鼠标拖动选择边框的黑色“手柄”。

删除: 选中要删除的一个或多个界面元素，然后按下界面工具条的“Delete”按钮。也可

以通过 Ctrl+单击 (Macintosh) 或右击 (其他系统)，然后在弹出菜单中选择“Delete”。如果使用后面这种方法，不必先选中元素。

要对各种界面元素有更多了解，参考下表。

表：界面工具条

图标 & 名字	描述
 Button	按钮可以是一次性的或永久性的。在一次性按钮上单击，将执行命令一次。永久性按钮则不断重复执行命令，直到再次按下按钮。如果为按钮分配了快捷键，则当按钮有焦点时，按下相应的键就等同于按下按钮。如果按钮有快捷键则在右上角显示快捷键字符。如果输入光标在另外的界面元素上，如命令中心，则按下快捷键不会触发按钮，这种情况下按钮右上角的字符会变暗。要激活快捷键，在界面页的空白背景上单击。
 Slider	滑动条是全局变量，可以被所有主体访问。在模型中使用他们作为快速改变变量的方式，而不需重新编程的。相反，用户移动滑动条到一个值，观察模型发生的行为。
 Switch	开关是 true/false 变量的可视化表示。通过拨动开关，用户设置变量为 on (true) 或 off (false)。
 Chooser	用户使用选择器在选择列表中为一个全局变量选定值，选择列表显示为下拉菜单。
 Input	输入框是包含字符串或数值的全局变量。编程人员选择用户可以输入的变量类型。可以设置输入框对输入的命令或报告器字符串进行语法检查。数值型输入框可以读取任何形式的常值表达式，这比滑动条灵活的多。颜色输入框为用户提供了 NetLogo 颜色选择器。
 Monitor	监视器显示任何表达式的值。表达式可以是变量、复杂表达式，或对报告器的调用。监视器每秒自动更新几次。
 Plot	绘图实时显示模型数据图形化。
 Output	输出区是一个文本卷滚区，用来记录模型活动。一个模型只能有一个输出区。
 Note	注释用来为界面页添加信息型文本标签。模型运行过程中注释内容不变。

界面页工具条上的其他控件用来控制视图更新和其他模型属性。



- 滑动条用来控制模型运行快慢 – 这很有用，因为有些模型运行的太快，很难看出发 生的事情。也可以通过左移滑块快进模型以及减慢视图更新频率。
- view updates 勾选框控制是否进行视图更新
- update mode 菜单用来在 continuous 和 tick-based 更新模式之间切换
- Settings 按钮用来编辑不同的模型属性

“Continuous”更新是指 NetLogo 每秒更新（即重绘）视图很多次，不管模型运行的是什 么。“Tick-based”更新是指只有滴答计算器推进时才更新视图。（关于视图更新的完整讨论， 见编程指南）

2D 和 3D 视图

界面页中的大块黑色区域是 2D 视图，它是 NetLogo 海龟和瓦片世界的可视化表示。初 始时它是全黑的，因为瓦片是黑色的，还没有海龟。通过在视图控制条上单击“3D”按钮打开 3D 视图，这是世界的另外一个可视化表示。



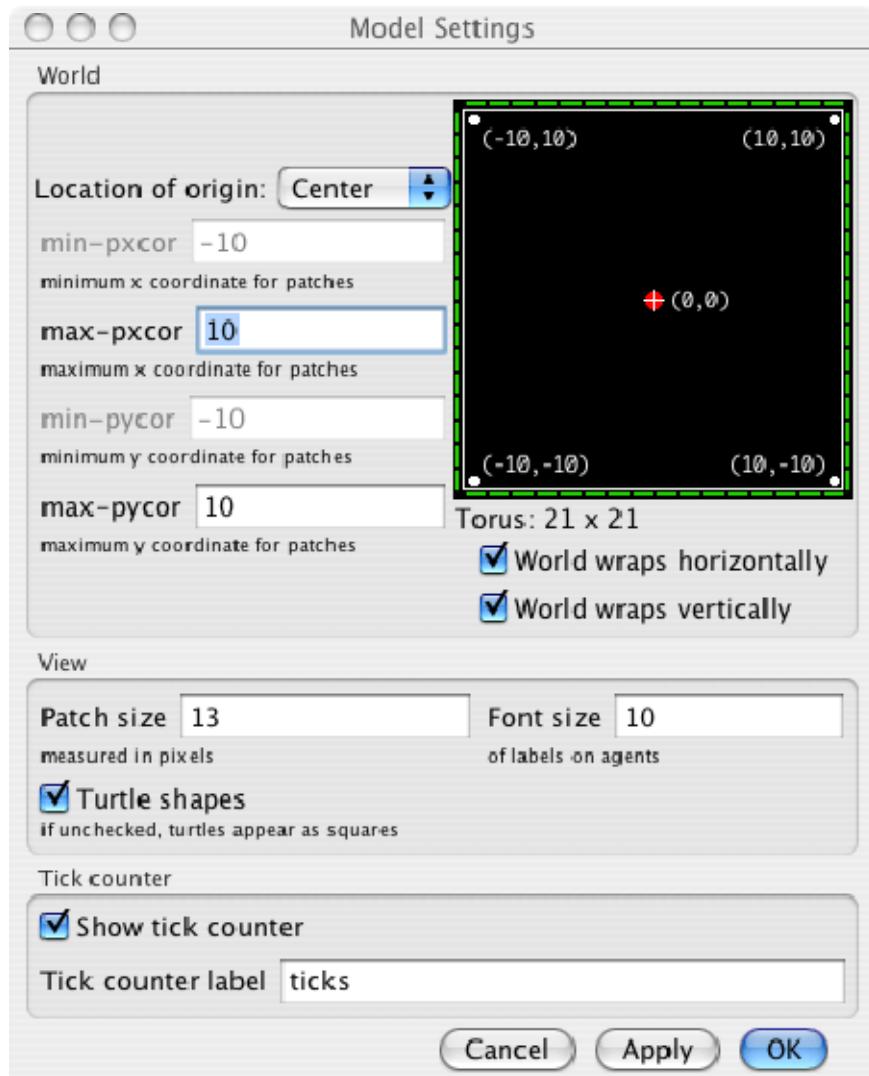
左上角三组黑色箭头用来改变世界大小。当原点在世界中点时世界变化的最小增量是 2，为最大值加 1，为最小值减 1。如果某条边为 0，则世界的增量为 1，在对面增加 1，以 保持原点在边上。如果原点在自定义的位置，则黑色箭头失效。

有很多与视图相关的设置。有几种改变设置的方式：使用视图顶端的控制条，或编辑 2D 视图，如上面的“使用界面元素”部分所讲，或按下工具条的“Settings...”按钮。

注意 3D 视图中的控制条组合了来自 2D 视图控制条的滴答计数器和界面工具条右部的控 件。



下面是视图的设置（通过编辑视图，或按下界面工具条的“Settings...”按钮）。



注意设置分为三组，即 world, view, ticks counter。World 部分的设置影响海龟生存的世界的特性（改变他们后要对世界重设）。View 和 tick counter 部分仅影响表现，不会影响模型的输出。

World 部分的设置用来定义世界的边界和拓扑。World 面板左部的顶端用来选择世界原点的位置，有“Center”，“Corner”，“Edge”，“Custom”四种。默认世界原点是 center 型的，即 (0,0) 在中心位置，用户定义从中心到左右边界和上下边界的瓦片数。例如：如果设置 Max-Pxcor = 10，则 Min-Pxcor 自动设为-10，则在瓦片 patch 0 0 的左侧有 10 个瓦片，在右侧有 10 个瓦片。

Corner 型配置允许用户将原点定义到世界的一角，左上、右上、左下或右下。然后定义 x 和 y 方向的远端边界。例如将原点放在左下角，定义右和上（正）边界。

Edge 型允许用户将原点放在一条边上 (x 或 y)，然后定义该方向的远端边界，及另一方向的两个边界。例如沿世界底部选择了 edge 模式，则必须定义顶边界和左右边界。

最后，Custom 模式允许用户将原点放在世界的任何位置，但瓦片 patch 0 0 必须存在。

当改变设置时，面板右部的预览区反映出你的选择，显示原点和边界。世界的宽度和高度显示在预览区下方。

在预览区下方还有两个勾选框，是 world wrap 设置，用于控制世界的拓扑。注意当点

击勾选框时，预览区显示哪个方向是回绕的，拓扑的名字显示在世界尺寸的旁边。更多信息参见编程指南的拓扑部分。

View 部分的设置用来定制视图观感而不改变世界。改变 view 设置不会强迫世界重设。要改变 2D 视图的大小，调整“Patch Size”，单位是像素。这不会改变瓦片的数量，只改变 2D 视图瓦片显示的大小。（注意 patch size 不影响 3D 视图，因为只需让窗口变大就可以使 3D 视图变大）

“Turtle Shapes”勾选框用来打开或关闭海龟“图形”，如果关闭则海龟显示为彩色方块，没有特别的形状。方块容易绘制，因此模型运行的会快一点。

“Smooth edges”仅出现在 3D 视图中，用来控制 3D 视图的反锯齿设置。使得直线看起来不太参差，但模型运行会慢一些。

Tick counter 部分的设置控制滴答计数器的显示，在视图控制条中出现与否。

在视图中容易调出 turtle, patch, link 监视器，只需在要查看的海龟或瓦片上 Ctrl-单击(Macintosh)或右击(其他系统)，并在弹出菜单中选择 inspect turtle...”或“inspect patch ...”。通过在 turtle 子菜单中选择合适的选项，实现对海龟的观察、跟随和乘骑。

(turtle, patch, link 监视器也可在 Tools 菜单中打开，或使用 [inspect](#) 命令)

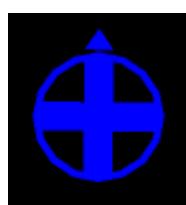
有些模型让你在视图中通过鼠标点击或拖动实现与海龟或瓦片的交互。

操纵 3D 视图

在窗口的底部有一些按钮用来移动观察者或改变观察世界的视角。



当你调整这些设置时，在当前焦点出现一个蓝色十字。小蓝三角形总是指向正 y 轴方向，如果迷失方向时用它帮你确定方向。很容易！



要从不同的角度看世界，按下“rotate”按钮，单击后上、下、左、右拖动鼠标。观察者持续面向与以前一样的点（蓝十字所在处），但与 xy-平面的关系变了。

要靠近或远离世界或正在观看、跟随或乘骑的主体，按下“zoom”按钮并上下拖动鼠标。（注意当在跟随或乘骑模式时，缩放将导致乘骑和跟随之间的切换，因为乘骑是跟随的一个特例，只是跟随距离为 0）

如果不改变观察者的方向而改变它的位置，选择“move”按钮并保持鼠标按下，在 3D 视图中上下左右拖动鼠标。

要允许鼠标位置和状态传给模型，选择“interact”按钮，则它的功能就像鼠标在 2D 视图中一样。

要将观察者和焦点返回默认位置，按下“Reset Perspective”按钮（或使用[reset-perspective](#)命令）

全屏模式

要进入全屏模式，按下“Full Screen”按钮，要退出全屏模式，按下 Esc 键。

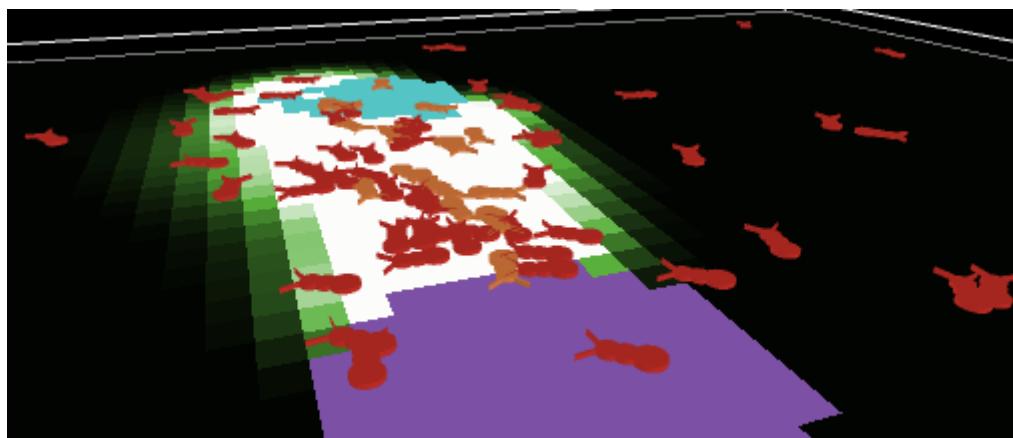
注意：全屏模式在某些计算机上不能用，取决于所有的图形卡，见系统需求部分。

3D 图形

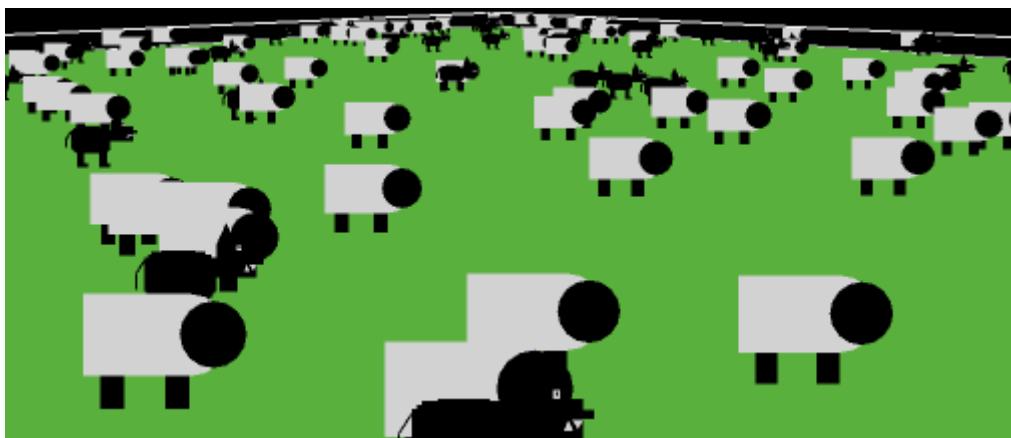
有些图形在 3D 视图中有 3D 对应物（3D 圆实际是一个球），因此自动映射图形。

图形名称	3D 图形
default	3D turtle shape
circle	sphere
dot	small sphere
square	cube
triangle	cone
line	3D line
cylinder	3D cylinder
line-half	3D line-half
car	3D car

所有其他图形解释为 2D 图形。如果图形可旋转，则假设是顶视图，像是从甜饼切割器中挤出的，与 xy 平面平行，就像 Ants 中那样。



如果图形不能旋转，则假设是侧视图，总是面对观察者画出（没有厚度），就像 Wolf Sheep Predation 中那样。

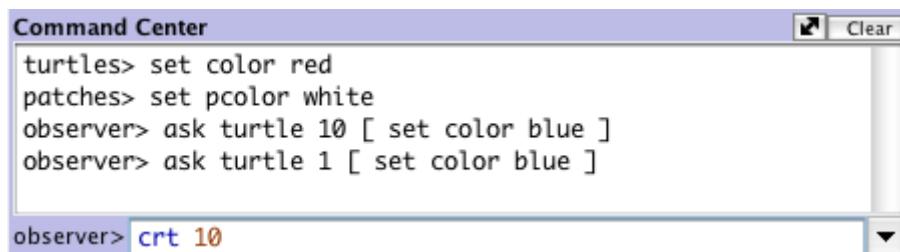


命令中心

命令中心用来直接发出命令，而不需将这些命令加入模型的例程。（命令是给模型中的主体发出的指令）。这对运行时监视和操纵主体很有用。

([教学#2:命令](#)介绍如何在命令中心使用命令)

看看命令中心的设计。



大框下面的小框用来输入命令，输入后按下回车键运行命令。

在输入文本的左侧是一个弹出菜单，初始是“observer>”，可以在 observer, turtles, patches 之中选择，指定哪个主体运行你输入的命令。

提示：使用 tab 键快速切换 observer, turtles, patches。

访问先前的命令

你输入的命令出现在命令行上方的滚转框中。可以用 Edit 菜单中的 Copy 命令拷贝该处的命令粘贴到其他地方，如例程页。

也可使用历史弹出菜单访问先前输入的命令，在输入命令框的右边有个小三角。单击这个小三角，出现弹出菜单，包含以前输入的命令，选择某项重用。

提示：可以使用键盘上的上下光标键更快速的访问先前的命令。

清除

单击右上角的“clear”，清除包含以前输入命令和输出的大滚转区。

要清除弹出式菜单上的历史命令，选择该菜单的“Clear History”。

安排

使用 Tools 菜单的 Hide Command Center 和 Show Command Center，隐藏或显示命令中心。

要改变命令中心的大小，拖动分隔命令中心和模型界面的边界，或者单击边界左部的小箭头使得命令中心很大或完全隐藏。

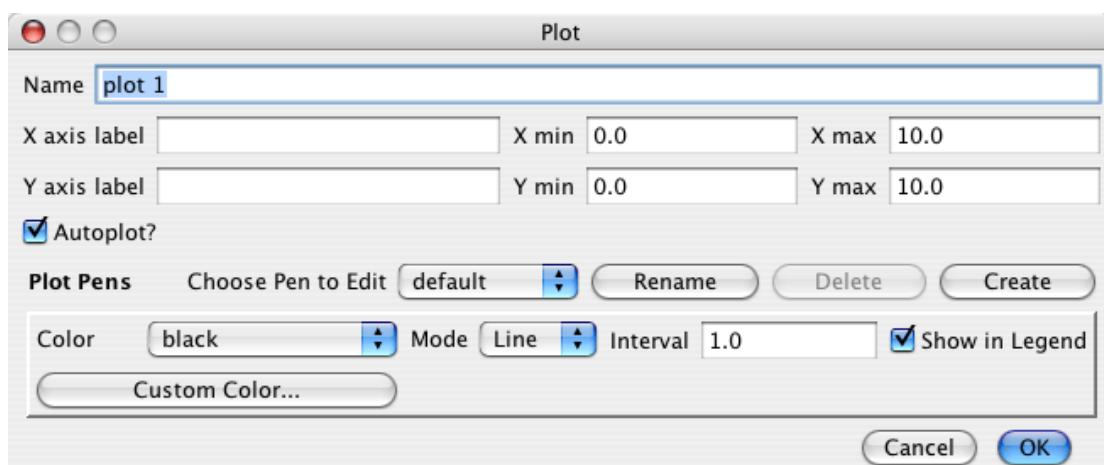
要在垂直和水平布局之间切换，单击画有双端箭头的按钮，它就在“Clear”的左方。

绘图

在绘图右上角的单词“Pens”上单击，将隐藏或显示画笔图例。

如果在绘图的白色区域上移动鼠标，鼠标的 x 和 y 坐标会显示。（注意鼠标位置可能和数据点不是精确对应，如果想知道绘图点的精确坐标，使用 Export Plot 菜单项，在其他程序中查看输出文件）

当创建一个绘图时，就像其他小部件一样，编辑对话框自动出现。



上面许多域是自明的，如绘图的 name，x 和 y 轴的 label，坐标范围等。

如果 Autoplot? 选中，如果新增的点超出当前范围，则 x, y 自动调整。

在 plot pens 部分可以创建和定制不同的画笔，每个绘图至少有一个画笔。开始时有一个名为“default”的画笔，可以进行重命名，与模型的实际意义一致。

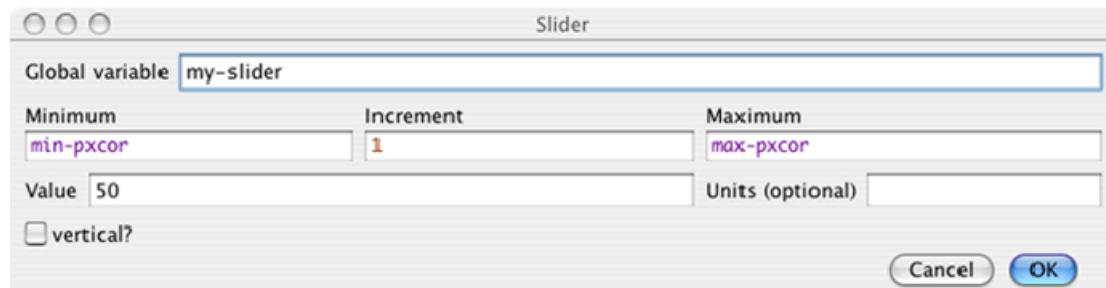
在画笔名下方的子面板部分是与该画笔相关的属性。

- 设置画笔的颜色为 NetLogo 的某个基色，或使用 color swatches 定制颜色
- Mode 用来设定画笔的外观，包括 line, bar (像棒图), point (虚线)
- Interval 是每次绘制新点时 x 的增量
- 如果 Show in Legend 选中，则选择的画笔是绘图右上角图例的一部分（在绘图上单击“Pens”就出现）

要了解这些特性的作用，参见编程指南的绘图部分。

滑动条

滑动条定义全局变量，用来方便的改变变量值，而不需改变底层代码。将滑动条放置到界面页时，自动出现编辑对话框，就像其他部件一样。多数域很熟悉，但要重点注意minimum, maximum 和 increment域可以接受任何报告器表达式，而不是只是常量。例如可以令minimum为`min-pxcor`，maximum 为`max-pxcor`，则当世界大小改变时，滑动条的边界自动调整。



信息页

信息页用来介绍模型、解释如何使用、说明要探索的事情、可能的扩展、NetLogo 特征等。首次探索模型时很有用。

WHAT IS IT?

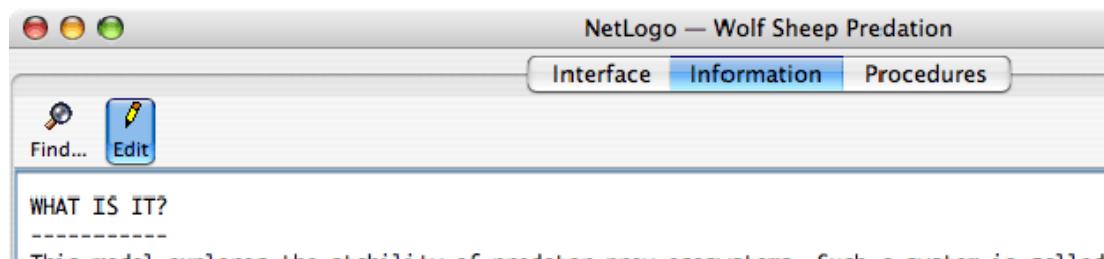
This model explores the stability of predator-prey ecosystems. Such a system is called unstable if it tends to result in extinction for one or more species involved. In contrast, a system is stable if it tends to maintain itself over time, despite fluctuations in population sizes.

HOW IT WORKS

There are two main variations to this model.

In the first variation, wolves and sheep wander randomly around the landscape, while the wolves look for sheep to prey on. Each step costs the wolves energy, and they must eat sheep in order to replenish their energy - when they run out of energy they die. To allow the population to continue, each wolf or sheep has a fixed probability of reproducing at each time

推荐在运行模型前阅读信息页。信息页解释建模原理和模型如何创建。显示的信息页不可编辑，要进行编辑单击“Edit”按钮，或在单词上双击，会自动滚转到该位置并加亮单词。



WHAT IS IT?

This model explores the stability of predator-prey ecosystems. Such a system is called unstable if it tends to result in extinction for one or more species involved. In contrast, a system is stable if it tends to maintain itself over time, despite fluctuations in population sizes.

HOW IT WORKS

There are two main variations to this model

可以像任何文本编辑器一样在此视图中编辑文本，然而，当切换出编辑视图后会特别显示几个不同的形式。

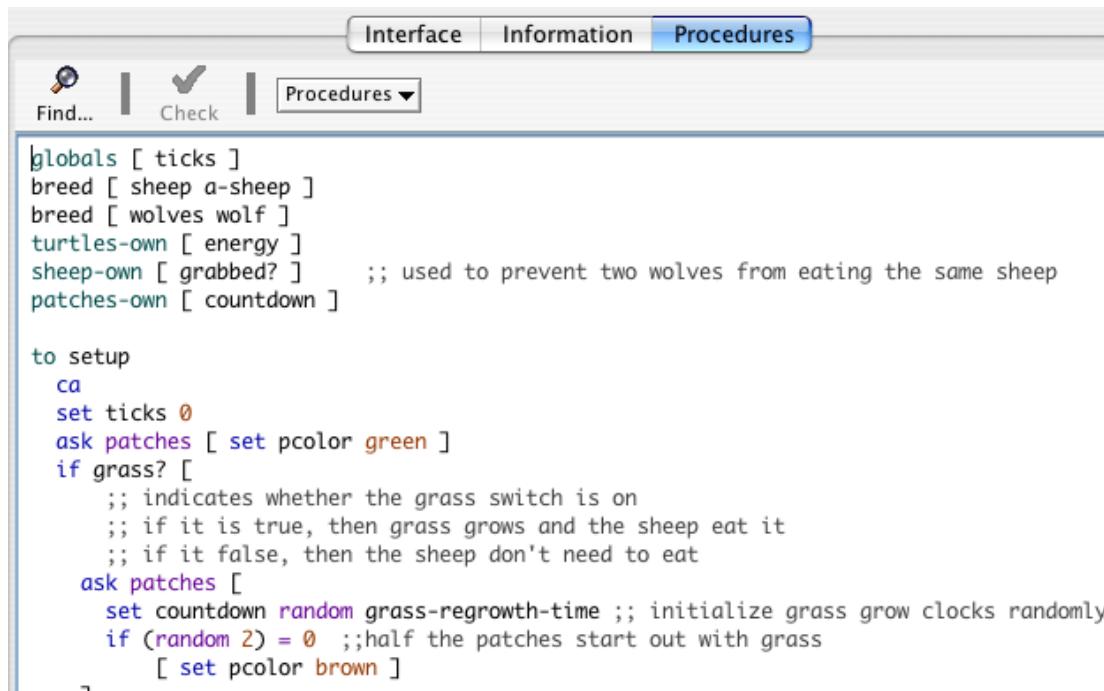
信息页标记

描述	编辑模式	视图模式
空行后面全大写的行成为节标题	WHAT IS IT	WHAT IS IT
破折号组成的行被忽略	-----	
以 http:// 开头的成为可点击的超链接	http://ccl.northwestern.edu	http://ccl.northwestern.edu
E-mail 地址成为可点击的"mailto:" 链接	bugs@ccl.northwestern.edu	bugs@ccl.northwestern.edu
以管道" " (shift + 反斜线\')开始的行成为等宽文本。这对图和复杂公式等有用。	this is preformatted text you can put spaces in it	this is preformatted text you can put spaces in it

单击 edit 按钮返回正常视图。

例程页

该页是模型代码存放的地方。对于马上想用的命令，可以在命令中心键入；对于要保存且以后会不断使用的则放在例程页。



The screenshot shows the NetLogo interface with the 'Procedures' tab selected. The code in the procedures editor is:

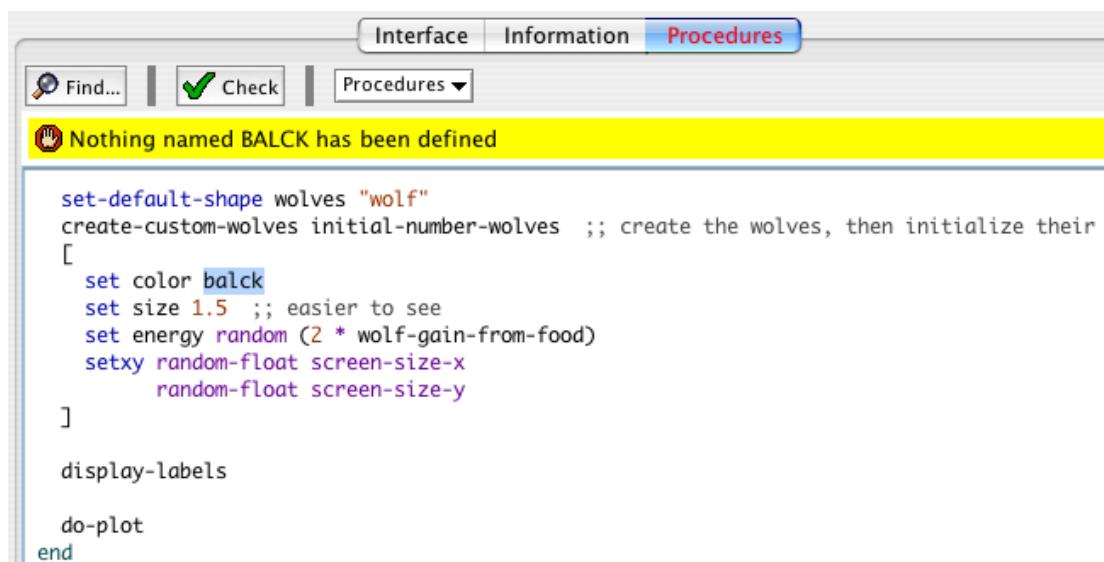
```

globals [ ticks ]
breed [ sheep a-sheep ]
breed [ wolves wolf ]
turtles-own [ energy ]
sheep-own [ grabbed? ]      ; used to prevent two wolves from eating the same sheep
patches-own [ countdown ]

to setup
  ca
  set ticks 0
  ask patches [ set pcolor green ]
  if grass? [
    ; indicates whether the grass switch is on
    ; if it is true, then grass grows and the sheep eat it
    ; if it false, then the sheep don't need to eat
    ask patches [
      set countdown random grass-regrowth-time ; initialize grass grow clocks randomly
      if (random 2) = 0 ; half the patches start out with grass
        [ set pcolor brown ]
    ]
  ]

```

要知道代码是否有错误，按下“Check”按钮。如果有语法错误，例程页标签会变红，有错的代码加亮，在顶部显示注释。切换标签页也会先进行语法检查，显示错误，因此切换前不需按 Check 按钮。



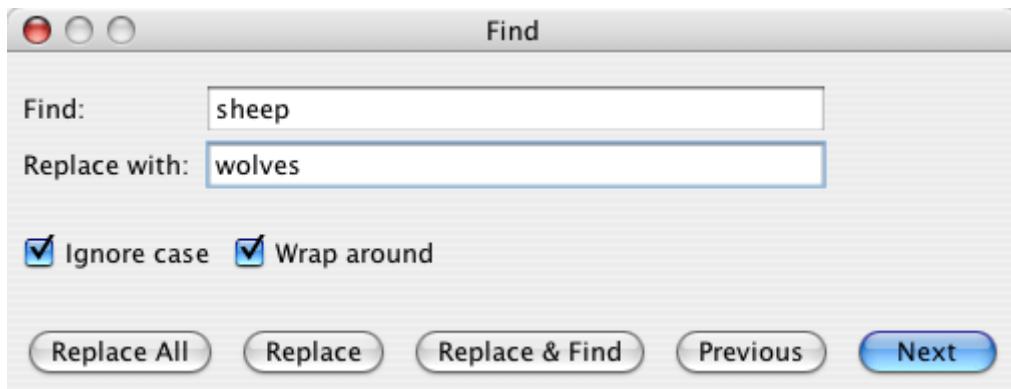
The screenshot shows the NetLogo interface with the 'Procedures' tab selected. A yellow bar at the top displays an error message: "Nothing named BALCK has been defined". The code in the procedures editor is:

```

set-default-shape wolves "wolf"
create-custom-wolves initial-number-wolves ; create the wolves, then initialize their [
  set color balck
  set size 1.5 ; easier to see
  set energy random (2 * wolf-gain-from-food)
  setxy random-float screen-size-x
    random-float screen-size-y
]
display-labels
do-plot
end

```

要查找代码片段，在例程工具条上单击“Find”按钮，出现 Find 对话框。



可以输入单词或短语进行查找或替换。“Ignore case”勾选框用来决定查找时是否大小写敏感。“Wrap around”选中后，将在整个例程页中查找，从光标位置查到底部，然后返回头部再查到光标处，否则只从光标处到底部。“Next”和“Previous”上下翻滚短语出现的位置。“Replace”将当前选择的短语替换，“Replace & Find”替换然后寻找下一个。“Replace all”替换搜索范围内所有匹配短语。

要寻找特定的例程定义，使用例程工具条的“Procedures”弹出菜单，菜单以字母顺序列出所有例程。

Edit 菜单的 Shift Left”，“Shift Right”，“Comment”，“Uncomment”用来对例程页改变代码的缩进层次，增加或去除分号，分号表示注释。

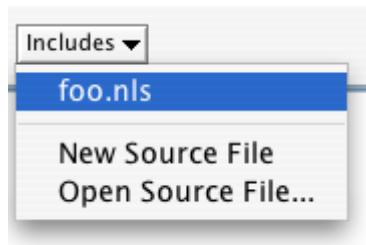
要了解编程的更多信息，阅读[教学#3：例程和编程指南](#)。

Includes 菜单

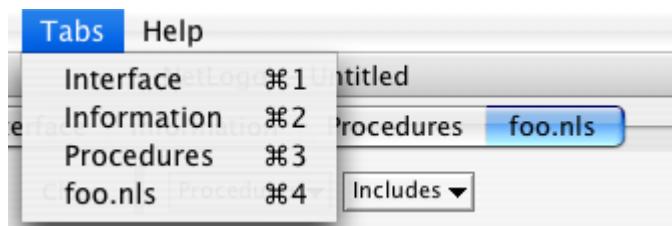
如果模型中加上 [includes](#) 关键词，procedures 菜单的右侧会出现 includes 菜单，列出本文件 (.nlogo 或 .nls) 包含的所有 NetLogo 源文件 (.nls)



在该菜单的文件名上单击，会打开一个新页包含该文件，或者使用 New Source File 和 Open Source File 分别打开新文件或打开文件系统中已有的一个或多个文件。



一旦打开新页，就可以与其他页一样进行导航。可以通过 Tabs 菜单访问它们，也可使用键盘从一个页切换到另一个页（Mac 上使用 Command + 数字，其他操作系统使用 Control+ 数字）。



编程指南 (Programming Guide)

下面的材料解释 NetLogo 编程的一些重要特征。

材料中经常提及的代码实例 (Code Example) 在模型库的 Code Example 部分。

- [Agents](#)
- [Procedures](#)
- [Variables](#)
- [Colors](#)
- [Ask](#)
- [Agentsets](#)
- [Breeds](#)
- [Buttons](#)
- [Lists](#)
- [Math](#)
- [Random Numbers](#)
- [Turtle Shapes](#)
- [Link Shapes](#)
- [Tick Counter](#)
- [View Updates](#)
- [Plotting](#)
- [Strings](#)
- [Output](#)
- [File I/O](#)
- [Movies](#)
- [Perspective](#)
- [Drawing](#)
- [Topology](#)
- [Links](#)
- [Ask-Concurrent](#)
- [Tie](#)
- [Multiple source files](#)
- [Syntax](#)

主体(Agents)

NetLogo 世界由主体(agent)构成，主体是能执行指令的个体。每个主体都同时执行各自的行为。

NetLogo 中有四类主体：海龟、瓦片、链和观察者。海龟是在世界中移动的主体，世界是 2 维的，是由瓦片组成的网格。每个瓦片是一块正方形的“地面”(ground)，海龟在它上面移动。链是连接两个海龟的主体。观察者没有具体位置 — 想象成它俯视着整个由海龟

和瓦片组成的世界。

当 NetLogo 启动后没有任何海龟。观察者可以创建新海龟，瓦片也可以创建新海龟。(瓦片不能移动，除此之外与海龟和观察者一样，也是“活的”)。

瓦片有坐标。坐标 (0, 0) 处的瓦片称为原点 (origin)，其他瓦片的坐标就是与原点的水平和垂直距离。瓦片的坐标用 `pxcor` 和 `pycor` 表示，像标准坐标平面一样，向右移动 `pxcor` 增加，向上移动 `pycor` 增加。

瓦片的总数由 `min-pxcor`, `max-pxcor`, `min-pycor`, 和 `max-pycor` 的设置决定。NetLogo 启动后，`min-pxcor`, `max-pxcor`, `min-pycor` 和 `max-pycor` 分别是 -16, 16, -16 和 16。也就是说 `pxcor` 和 `pycor` 的范围都是从 -16 到 16，因此共有 $33 \times 33 = 1089$ 个瓦片。(通过 Setting 按钮，可以改变瓦片数)。

海龟也有坐标：`xcor` 和 `ycor`。瓦片的坐标总是整数，但海龟的坐标可以有小数，这意味着海龟可以位于瓦片上的任何一点，不一定恰好在瓦片的中心。

链只有两个端点(每个端点是一个海龟)，没有坐标。链出现在两个端点之间，沿着可能的最短路连接，这意味着有时候甚至要沿世界回绕。

瓦片世界的连接方式可以改变。默认情况下，世界是一个环面 (torus)，无边，但回绕 — 因此当海龟移出边界时消失了，又出现在对面的边界，这也使得每个瓦片都有相同数目的“邻元”瓦片。一个位于世界边缘的瓦片，它的有些邻元在对面的边上。可以使用 Settings 按钮改变回绕设置。如果禁止某方向 (x 或 y) 的回绕，世界就是有界的。边界上的瓦片邻元少于 8 个，海龟也移不出边界。要详细了解参见 [Topology](#) 部分。

例程(Procedures)

NetLogo 命令和报告器告诉主体做什么。命令 (command) 是主体执行的行动。报告器 (reporter) 计算并返回结果。

多数命令由动词开头 ("create", "die", "jump", "inspect", "clear")，而多数报告器是名称或名词短语。

NetLogo 内建的命令和报告器叫做原语 (primitive)。NetLogo 词典 ([The NetLogo Dictionary](#)) 完整列出了内置命令和报告器。

你自己定义的命令和报告器称为例程。每个例程有一个名字，前面加上关键词 `to`。关键词 `end` 标志例程的结束。定义了例程后，就可以在程序的其他任何地方使用它。

许多命令和报告器有输入参数 (inputs)，就是命令或报告器执行动作所需的一些值。

例子：两个命令例程：

```
to setup
  clear-all      ;; clear the world
  crt 10        ;; make 10 new turtles
end

to go
  ask turtles
    [ fd 1          ;; all turtles move forward one step
```

```

rt random 10      ;; ...and turn a random amount
lt random 10 ]
end

```

使用分号给程序增加注释，注释让程序更容易阅读和理解。

在这个程序中，

- `setup` 和 `go` 是用户定义的命令。
- `clear-all`, `crt` ("create turtles"), `ask`, `lt` ("left turn") 和 `rt` ("right turn") 是命令原语。
- `random` 和 `turtles` 是报告器原语，`random` 使用一个输入参数，返回小于等于该参数的一个随机整数 (此处是 0 和 9 之间)。 `turtles` 返回所有海龟组成的agentset (`agentsets` 后面再解释。)

`setup` 和 `go` 可以被其他例程或按钮调用。许多 NetLogo 模型有一个一次性按钮调用一个名为 `setup` 的例程，还有一个永久性按钮调用一个名为 `go` 的例程。

在 NetLogo 里必须指定每条命令由哪个/哪些主体执行，包括海龟、瓦片、链和观察者。(如果不指定，则由观察者执行)。在上面的代码中，观察者使用 `ask` 让所有海龟执行 [] 中的命令。

`clear-all` 和 `crt` 只能由观察者执行。`fd` 只能由海龟执行。其他一些命令和报告器，例如 `set`，可以由不同类型的主体执行。

定义例程时还有一些高级特征可以使用。

带输入的例程，Procedures with inputs

你的例程也可以像原语一样有输入参数。要定义接受输入的例程，需要在例程名后面的 [] 中列出输入参数名。例如：

```

to draw-polygon [num-sides len]
  pen-down
  repeat num-sides
    [ fd len
      rt 360 / num-sides ]
end

```

在程序的其他地方，你可以让每个海龟以自己的 who number 为边长画出一个正 8 边形。

```
ask turtles [ draw-polygon 8 who ]
```

报告器例程，Reporter procedures

就像命令一样，也可定义自己的报告器。这需要做两件特别的事情，一是使用 `to-report`，而不是 `to`，开始例程定义；二是在例程体中，使用 `report` 返回你要报告的值。

```
to-report absolute-value [number]
```

```

ifelse number >= 0
  [ report number ]
  [ report (- number) ]
end

```

变量(Variables)

变量用来存储值（例如数字）。变量可以是全局变量、海龟变量或瓦片变量。

一个全局变量只有一个值，任何主体都可访问它。对于海龟变量，每个海龟都有一个自己的值。对于瓦片变量，每个瓦片都有一个自己的值。

有些变量是NetLogo内置的。例如，所有海龟都有一个[color](#)变量，所有瓦片都有一个[pcolor](#)变量。（瓦片变量以p开头，以免与海龟变量混淆）。如果你设置这些变量，海龟或瓦片就变色。

其他内置海龟变量包括[xcor](#), [ycor](#)和 [heading](#), 其他内置瓦片变量包括[pxcor](#) 和[pycor](#)。
(有个完整列表[here](#))

你可以定义自己的变量。通过创建开关和滑动条创建全局变量，或者在程序的开头使用[globals](#)关键字，像这样：

```
globals [ score ]
```

使用[turtles-own](#) , [patches-own](#) 和[links-own](#)关键词，定义自己的新的海龟变量、瓦片变量和链变量。像这样：

```

turtles-own [energy speed]
patches-own [friction]
links-own [strength]

```

在模型里这些变量可以随便使用。使用[set](#)命令设置它们。（如果不设置的话，初值为0）

任何主体在任何时候都可以读取、设置全局变量。海龟可以读取、设置它所处瓦片的瓦片变量。例如代码：

```
ask turtles [ set pcolor red ]
```

使得每个海龟将所处的瓦片变为红色。（由于瓦片变量以这种方式被海龟更新，因此海龟变量和瓦片变量不能重名）。

其他情况下，当你要一个主体读取其他主体的变量时，使用[of](#)。例如：

```

show [color] of turtle 5
;; prints current color of turtle with who number 5

```

除了将变量名与[of](#)一起使用外，还可使用复杂的表达式，例如：

```

show [xcor + ycor] of turtle 5
;; prints the sum of the x and y coordinates of

```

```
;; turtle with who number 5
```

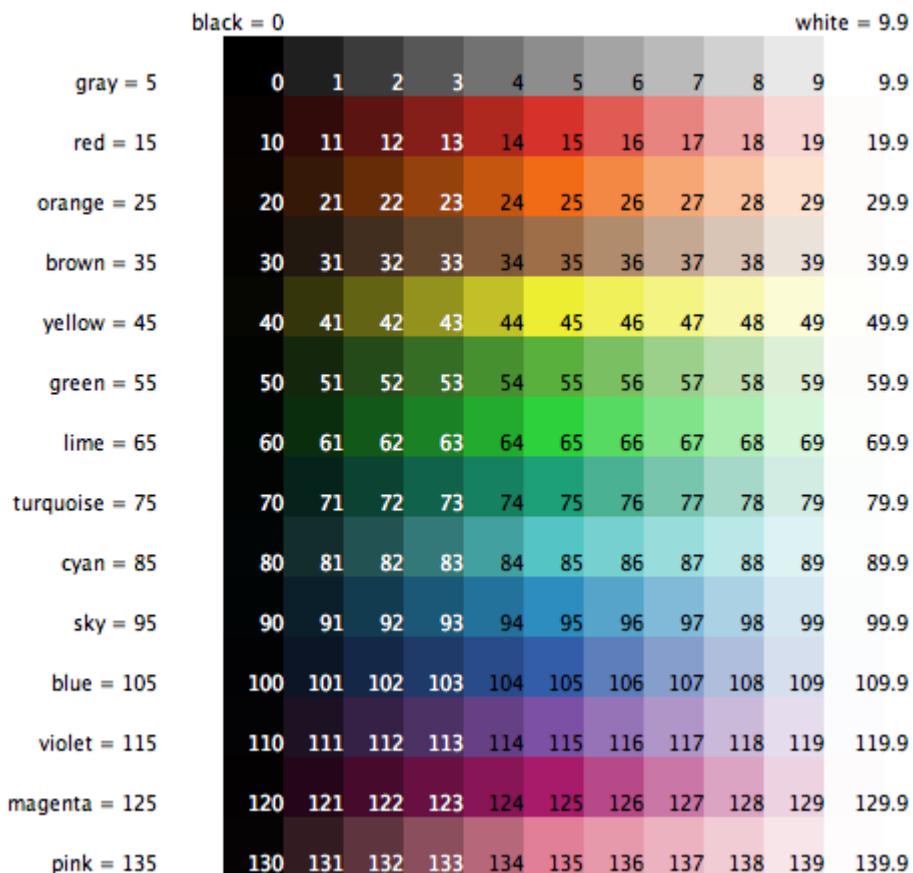
局部变量, Local variables

局部变量仅用在特定的例程或例程的一部分。使用[let](#)命令创建局部变量，该命令可在任何地方使用。如果在例程的最前面使用，则变量在整个例程中都存在。如果在[]中使用，例如在ask里面，它只在该[]内部存在。

```
to swap-colors [turtle1 turtle2]
  let temp [color] of turtle1
  ask turtle1 [ set color [color] of turtle2 ]
  ask turtle2 [ set color temp ]
end
```

颜色(Colors)

NetLogo 有两种表示颜色的方式。第一种是使用 0–140（不包括 140）之间的数字。下面是一张 NetLogo 可用颜色范围的图。



该图表明：

- 有些颜色有名字 (在代码里可以使用)
- 除了 black 和 white 外, 有名颜色的末位数是 5
- 有名颜色的两侧是同一种颜色, 但更深或更浅
- 0 是纯黑, 9.9 是纯白
- 10, 20, 等太深, 看起来是黑的
- 19.9, 29.9 等太浅, 看起来是白的

代码示例：这个颜色图是用 NetLogo 的 Color Chart Example 模型生成的.

如果使用的数不在 0-140 之间, 则NetLogo重复增加或减去 140 直到符合范围。例如 25 是橙色, 165, 305, 445 等也是橙色, -115, -255, -395 也是。当你设置海龟[color](#)或瓦片[pcolor](#)时, 自动做上述计算。如果在别处你需要做这样的运算, 使用[wrap-color](#)原语。

如果你要图上没有的颜色, 就使用整数之间的值, 例如 26.5 是 26 和 27 的中间色。这并不是说你可以在 NetLogo 里使用任何颜色, NetLogo 的颜色空间仅是全部颜色的一个子集, 仅包括有限的固定的离散色调 (hue) 集合(图上每行是一个色调)。对每个色调可以减少亮度或减少饱和度, 但不能同时减少亮度和饱和度。并且只有小数点后第一位有意义, 因此颜色值四舍五入到 0.1, 例如 26.5, 26.52 和 26.58 视觉上无区别

Color primitives, 颜色原语

有些原语与颜色打交道。

我们已经提过 [wrap-color](#) 原语。

原语[scale-color](#)将数值转换为颜色。

原语[shade-of?](#)告诉我们两个颜色是否属于同一色调。例如shade-of? orange 27

代码例子 (Code Example) : Scale-color 例子演示了 scale-color 原语的使用.

RGB Colors, RGB 颜色

NetLogo第二种颜色表达方式是RGB(红/绿/蓝), 使用RGB时所有颜色都可以得到。RGB 颜色由 3 个 0-255 之间的数组成, 若数字超出 255 则重复减去 255, 直到落到范围之内。可以将RGB表示的颜色用于使用颜色变量 (海龟和链使用[color](#), 瓦片使用[pcolor](#)) 的任何地方。因此下面的代码将瓦片 0 0 设为纯红:

```
set pc当地 [255 0 0]
```

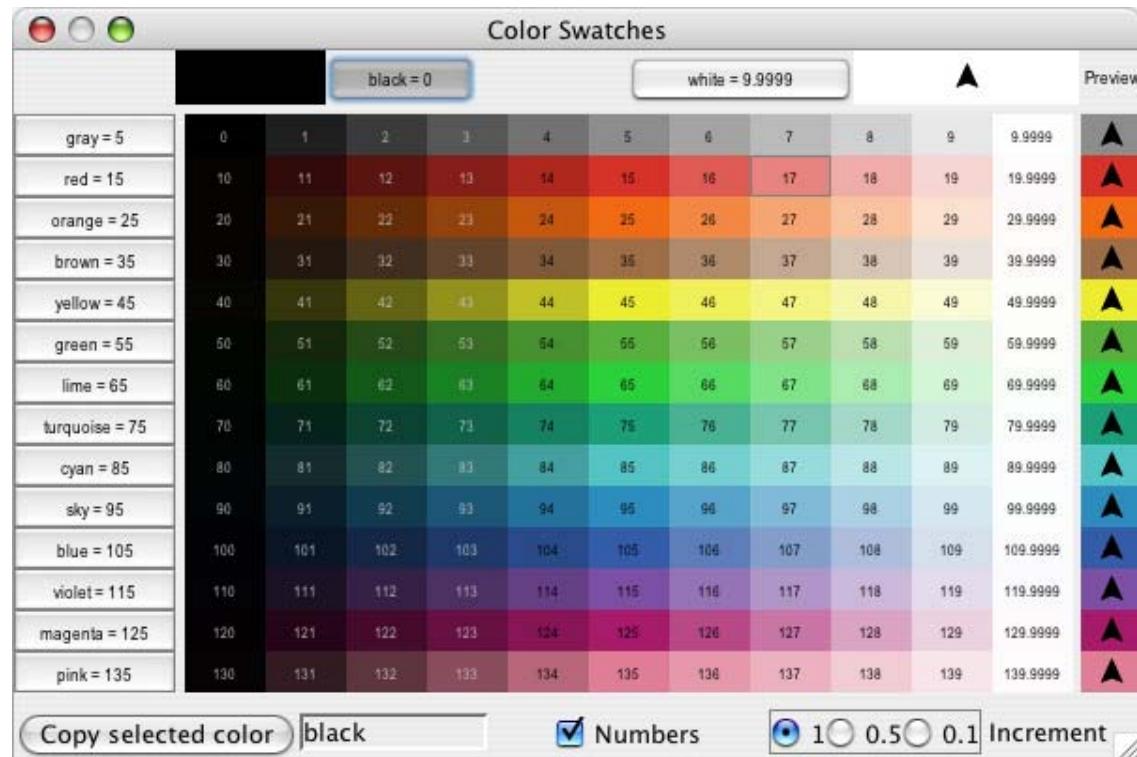
可以在RGB和 HSB (色调/饱和度/亮度)之间转换。NetLogo 使用[approximate-hsb](#)和[approximate-rgb](#) 将颜色从 RGB/HSB映射到NetLogo颜色。而[extract-hsb](#) 和 [extract-rgb](#) 进行反向转换。使用[rgb](#)产生rgb颜色表, 使用[hsb](#)实现从HSB到RGb 的转换。

因为NetLogo颜色空间缺很多颜色, [approximate-hsb](#) 和 [approximate-rgb](#)得到的颜色可能不太准确, 但尽量接近。

代码示例 (Code Example) : HSB 和 RGB 模型例子让你实验 HSB 和 RGB 颜色系统。

Color Swatches dialog , 色样对话框

色样对话框让你体验、选择颜色。在 Tools 菜单选择 Color Swatches 打开对话框。



当在色样（或颜色按钮）上点击时，颜色会凸显。左下部显示当前颜色的代码（例如，red + 2），这样你可以将它复制、粘贴到程序中。右下部有三个颜色增量选项，1, 0.5, 0.1。增量为 1 时，每行 10 个颜色，0.1 时每行 100 个颜色，0.5 是中间设置。

请求(Ask)

NetLogo用[ask](#)向海龟、瓦片和链发出命令。由海龟执行的命令**必须**置于海龟上下文(context)。用下面三种方式之一建立海龟上下文：

- 使用按钮时在弹出菜单选择“Turtles”，放在按钮中的代码将由所有海龟执行。
- 在命令中心，在弹出菜单中选择“Turtles”，则输入的命令由所有海龟执行。
- 使用 ask turtles.

瓦片、链、观察者也一样，只是不能[ask](#)观察者。不在[ask](#)内的代码默认是由观察者执行。这里有一个在NetLogo例程中使用[ask](#)的例子：

```
to setup
  clear-all
  crt 100          ;;; create 100 turtles
  ask turtles
```

```

[ set color red      ;; turn them red
  rt random-float 360    ;; give them random headings
  fd 50 ]           ;; spread them around
ask patches
[ if pxcor > 0       ;; patches on the right side
  [ set pcolor green ] ]  ;; of the view turn green
end

```

模型库里这样的例子很多，最好先看 Code Examples 部分。

通常观察者使用[ask](#)请求所有海龟、所有瓦片或所有链执行命令。也可以使用[ask](#)让单个海龟、瓦片或链执行命令。报告器[turtle](#), [patch](#), [link](#)和 [patch-at](#)是经常使用的技术。

```

to setup
  clear-all
  crt 3
  ask turtle 0
  [ fd 1 ]
  ask turtle 1
  [ set color green ]
  ask turtle 2
  [ rt 90 ]
  ask patch 2 -2
  [ set pcolor blue ]
  ask turtle 0
  [ ask patch-at 1 0
    [ set pcolor red ] ]
  ask turtle 0
  [ create-link-with turtle 1 ]
  ask link 0 1
  [ set color blue ]
end

```

创建的每个海龟都有who number号，第一个是0，第二个是1，依次类推。[turtle](#) 原语报告器使用who number作为输入，返回该海龟。[patch](#)原语报告器使用pxcor 和pycor，返回该处的瓦片。[link](#)原语使用两个端点海龟的who number做输入。[patch-at](#)原语使用与第一个主体的offsets，即x, y方向的距离。上面的例子里请求0号海龟去获得它东面(没有北面的瓦片)的瓦片。

也可选择某些海龟、瓦片、链等让它们执行动作。这就涉及到一个称为主体集合(agentset)的概念，下一部分详细解释这个概念。

当你命令主体集合执行多条命令时，只有当一个主体执行完所有这些命令后，才轮到下

一个主体执行。一个主体执行完，然后下一个主体，…，诸如此类。例如，写代码：

```
ask turtles
  [ fd 1
    set color red ]
```

首先一个海龟移动、变红，然后是下一个海龟移动、变红，…。

但是若写成这样：

```
ask turtles [ fd 1 ]
ask turtles [ set color red ]
```

首先所有海龟移动，都移动完成后，再都变红。

(还有另一种排序规则不同的[ask](#) 命令，参看下面的[Ask-Concurrent](#)。)

主体集合(Agentsets)

顾名思义，主体集合就是主体组成的集合。主体集合可以由海龟、瓦片或链组成，但只能同时包含一种类型的主体。

主体集合内部元素没有任何特定顺序，总是随机排列。每次使用时都会是不同的随机顺序。这使你避免对集合中的主体做任何特定处理（除非你非要这样）。因为每次的顺序都是随机的，没有哪个主体会总是排在第一个。

你已经知道原语[turtles](#)返回所有海龟组成的主体集合，[patches](#)是所有瓦片组成的主体集合，[links](#)是所有链组成的主体集合。

主体集合概念的作用在于你可以构造由某些海龟、某些瓦片或某些链组成的集合。例如，所有红色海龟、或pxcor能被 5 整除的瓦片，或者站在第一象限绿色瓦片上的海龟，或者与 0 号海龟相连的链等。这些集合可以用在[ask](#)中，或者用在将主体集合作为输入的报告器中。

使用[turtles-here](#)得到一个由当前瓦片上所有海龟构成的主体集合。使用[turtles-at](#)得到距当前位置x, y处瓦片上的海龟构成的主体集合。使用[turtles-on](#)得到给定瓦片上的海龟集合，或者得到与给定的海龟站在同一瓦片上的海龟集合。

下面是一些构造主体集合的例子：

```
; all other turtles:
other turtles
;; all other turtles on this patch:
other turtles-here
;; all red turtles:
turtles with [color = red]
;; all red turtles on my patch
turtles-here with [color = red]
;; patches on right side of view
patches with [pxcor > 0]
;; all turtles less than 3 patches away
```

```

turtles in-radius 3
;; the four patches to the east, north, west, and south
patches at-points [[1 0] [0 1] [-1 0] [0 -1]]
;; shorthand for those four patches
neighbors4
;; turtles in the first quadrant that are on a green patch
turtles with [(xcor > 0) and (ycor > 0)
               and (pcolor = green)]
;; turtles standing on my neighboring four patches
turtles-on neighbors4
;; all the links connected to turtle 0
[my-links] of turtle 0

```

注意使用[other](#)就将调用主体排除在外。这是通用的。

创建了主体集合后，可以做一些事情，例如

使用[ask](#) 让主体集合中的主体做事

使用[any?](#) 查看主体集合是否为空

使用[all?](#) 查看是否主体集合中的每个主体都满足条件

使用[count](#) 得到主体集合中主体的数量

也可以做一些更复杂的事情：

使用[one-of](#)在集合中随机选一个主体，例如我们让一个随机选择的海龟变绿：

```
ask one-of turtles [ set color green ]
```

或者让随机选定的一个瓦片生出一个新海龟：

```
ask one-of patches [ sprout 1 ]
```

使用[max-one-of](#) 或 [min-one-of](#) 报告器找出某个指标最大或最小的主体。例如移动最富的海龟：

```
ask max-one-of turtles [sum assets] [ die ]
```

使用[histogram](#)命令做一个关于主体集合的直方图 (与[of](#)一起)

使用[of](#)得到主体集合中每个主体的一系列值。然后使用list原语做一些事情(看下面的List部分)。例如，显示海龟平均财富，

```
show mean [sum assets] of turtles
```

使用 [turtle-set](#),[patch-set](#)和[link-set](#) 报告器从多个来源收集主体形成主体集合。

使用 `=` 或 `!=` 判断两个主体集合是否相等。

使用 `member?` 判断某个特定主体是否属于一个主体集合。

这只是隔靴搔痒。在模型库里例子很多，还可以查 NetLogo 词典获得有关主体集合原语更多的知识。

在 NetLogo 词典里，每个原语的条目下都提供了更多的例子。在熟悉了 NetLogo 编程之后，要进一步考虑复合命令，特别是要注意复合命令的各个构件之间如何传递信息。在这样的概念模式里，主体集合发挥重要作用，它提供了强大而灵活的功能，并且与自然语言类似。

代码例子 Code Example: Ask Ordering

前面我们说过主体集合总是采用随机顺序，每次都不一样。如果要主体遵循固定的顺序，你需要主体列表（list），而不是集合。参看下面 list 部分。

种类(Breeds)

NetLogo 允许定义不同种类（breeds）的海龟或链。定义了种类后，可以让它们有不同的行为。例如有两个种类：羊（sheep）和狼（wolves），让狼吃羊。或者不同种类的链：马路和人行道，人走人行道，车走马路。

在例程页使用 `breed` 关键字定义海龟种类，定义必须放在所有例程之前。

```
breed [wolves wolf]
breed [sheep a-sheep]
```

用单数形式引用种类的成员，就像 `turtle` 那样。打印时，种类成员使用单数形式的标签。

有些命令或报告器使用复数形式的种类名，例如 `create-⟨breeds⟩`。其他的使用单数形式种类名，如 `⟨breed⟩`。

种类定义的顺序决定了它们在视图上分层显示的顺序。后定义的种类在先定义的种类上面。上面狼和羊的定义决定了羊会绘制在狼的上层。

当你定义了羊这样的种类后，这个种类的主体集合自动产生了。上面叙述的主体集合的功能，现在对关于羊的主体集合马上可用。

一旦定义了羊（sheep）这个种类，其他的一些新原语自动创建：`create-sheep`, `hatch-sheep`, `sprout-sheep`, `sheep-here`, `sheep-at`, `sheep-on`, 和 `is-a-sheep?`

你也能使用 `sheep-own` 定义属于该种类的海龟变量。

一个海龟种类主体集合存在该种类的海龟变量中（A turtle's breed agentset is stored in the `breed` turtle variable）。因此可以测试海龟的种类：

```
if breed = wolves [ ... ]
```

注意海龟也可以改变种类。狼也不一定一辈子都是狼，随机选一个狼变成羊：

```
ask one-of wolves [ set breed sheep ]
```

使用原语 `set-default-shape` 将特定形状与特定种类联系起来。参见下面的 shape 部分。

下面是一个使用种类的小例子：

```

breed [mice mouse]
breed [frogs frog]
mice-own [cheese]
to setup
  clear-all
  create-mice 50
    [ set color white
      set cheese random 10 ]
  create-frogs 50
    [ set color green ]
end

```

代码例子 Code Example: Breeds and Shapes

链的种类(Link Breeds)

链的种类与海龟种类很相似，但有一些区别。

声明链种类时，必须声明是有向还是无向，分别使用[directed-link-breed](#) 和 [undirected-link-breed](#) 关键词。

```

directed-link-breed [streets street]
undirected-link-breed [friendships friendship]

```

一旦你创建了有种类的链，就不能再创建无种类的链，反之亦然。（但是，可以同时有有向链和无向链，但不能属于同一种类）。

与海龟种类不同，链种类需要单数形式种类名，因为许多链命令和报告器使用单数名，如[`<link-breed>-neighbor?`](#)

一旦定义了上面的有向链种类，下面的原语就自动可用：[create-street-from](#) [create-streets-from](#) [create-street-to](#) [create-streets-to](#) [in-street-neighbor?](#) [in-street-neighbors](#) [in-street-from](#) [my-in-streets](#) [my-out-streets](#) [out-street-neighbor?](#) [out-street-neighbors](#) [out-street-to](#)

一旦定义了上面的无向链种类，下面的原语就自动可用：[create-friendship-with](#) [create-friendships-with](#) [friendship-neighbor?](#) [friendship-neighbors](#) [friendship-with](#) [my-friendships](#)

与海龟种类一样，链种类声明的顺序决定了它们绘制的顺序，因此friendships在streets上面（如果因某种原因，它们在一个模型里）。可以用[`<link-breeds>-own`](#)为每个链种类分别声明变量。

像海龟一样，也可以改变链的种类。然而，不能让有种类的链变为无种类的，以免在世界中同时出现有种类和无种类的链。

```
ask one-of friendships [ set breed streets ]
```

```
ask one-of friendships [ set breed links ] ;; produces a runtime error
```

使用[set-default-shape](#) 将特定形状与特定种类的链联系起来。

代码例子 Code Example: Link Breeds

按钮(Buttons)

界面页中的按钮用来方便的控制模型。一般模型至少有一个“setup”按钮，设置世界初始状态，还有一个“go”按钮运行模型。一些模型有更多按钮执行其他行为。

按钮包含一些 NetLogo 代码，按下按钮代码运行。

按钮可以是一次性的或永久性的，编辑按钮通过勾选/不选“Forever”项来决定。一次性按钮执行代码一次，然后停止并弹回。永久性按钮不断重复执行代码，直到遇到[stop](#)命令，或再次按下按钮。按下按钮，代码并不马上停止，而是直到代码执行完毕才弹回。

一般用代码名字命名按钮。例如名为“go”的按钮执行的代码就是“go”，含义为“执行 go 例程”（例程在例程页定义）。但也可以编辑按钮，输入一个显示名（display name），显示名出现在按钮上。如果你觉得实际代码不太容易搞懂的话，可以指定一个显示名。

在按钮里放入代码时，必须指定哪个主体来执行。可以选择观察者、所有海龟、所有瓦片或所有链。（如果只想让部分海龟或部分瓦片执行的话，需要制作一个观察者按钮，然后让观察者使用[ask](#)请求那些海龟或瓦片执行动作）

编辑按钮时，可以指定快捷键，按下快捷键等于按下按钮。如果是永久性按钮，按下快捷键后按钮一直处于压下状态，直到再次按下快捷键（或单击该按钮）。快捷键对游戏，或需要快速触发按钮的模型很有用。

按钮次序 (Buttons take turns)

可以同时有多个按钮被按下，这种情况下按钮按顺序执行，即每次只有一个按钮在运行。每个按钮执行完它的代码后，下一个按钮接着进行。

下面的例子中，“setup”是个一次性按钮，“go”是永久性按钮。

例子#1: 按下“setup”，在“setup”弹起前立即按下“go”。结果：在“go”开始前“setup”完成。

例子#2: 当“go”处于按下状态时，按下“setup”。结果是：“go”完成本次循环，然后执行“setup”，然后“go”再次运行。

例子#3: 同时有两个永久性按钮按下，结果是：首先一个按钮执行代码一遍，然后另一个按钮执行代码一遍，如此交替执行。

注意如果一个按钮陷入死循环，则其他按钮无法执行。

海龟、瓦片和链永久性按钮，Turtle, patch, and link forever buttons

将命令置入海龟、瓦片、链的永久性按钮，与将相同的命令置入观察者按钮的 ask 中有微妙的区别。只有所有主体执行完 ask 中的命令后，ask 才完成。因此当主体并行执行命令

时，它们之间可能不同步，但当 ask 结束时又同步了。在海龟、瓦片和链的永久性按钮中并非如此。因为没有使用 ask，每个海龟或瓦片不断重复执行各自代码，因此相互之间变得（并保持）不同步。

目前模型库中的模型很少利用这种能力。确实使用了这个能力的模型是 Sample Models 中 Biology 部分的 Termites，其中“go”是海龟永久性按钮，因此每个白蚁(termit)相互独立运行，没有涉及观察者。这意味着，例如，要加上一个图的话，需要增加第二个永久性按钮（观察者永久性按钮），同时有两个永久性按钮在运行。

目前 NetLogo 没办法让一个永久性按钮启动另一个按钮，只有按下按钮启动它。

列表 (Lists)

在最简单的模型里，每个变量保存一项信息，一般是数字或字符串。列表使得一个变量能存储多项信息，只要将它们收集在一个列表里。列表中的值可以是任何类型：数字、字符串、主体、主体集合，甚至另一个列表。

列表使你可以方便的对信息打包。如果主体对多个变量进行重复计算，使用列表就很方便，不用使用多个数值变量。有些原语简化了对列表中每个值进行相同计算的过程。

NetLogo词典[NetLogo Dictionary](#)列出了与列表有关的所有原语。

常数列表，Constant lists

把所需要的值都写在[]里就简单的创建了一个列表，像这样set mylist [2 4 6 8]。注意每个值用空格分开。用这种方式能创建包括数值和字符串的列表，以及包含列表的列表，如[[2 4] [3 5]]。

[]中间什么都不放，就创建一个空列表，例如[]。

当场创建列表，Building lists on the fly

要创建由报告器返回值构成的列表，而不是常数组成的列表，要使用[list](#)报告器。[list](#)报告器接受两个其他报告器做输入，运行这些报告器，返回作为列表。

要得到包括两个随机数的列表，使用下面的代码：

```
set random-list list (random 10) (random 20)
```

每次运行时将random-list 设置为由两个随机整数形成的列表。

要得到更长或更短的列表，可以给[list](#)报告器少于或多于两个输入参数，但是必须将整个调用用括号括起来，例如：

```
(list random 10)
(list random 10 random 20 random 30)
```

更多信息，见 [Varying number of inputs](#).

有些种类的列表可以通过[n-values](#)报告器很容易的创建，它允许你重复运行某个给定的报告器产生指定长度的列表。列表可以包括相同值，或某个范围内的所有值，许多随机值，

等等。查看NetLogo词典了解详细内容。

原语[of](#) 用来从主体集合创建列表，它返回一个列表，其中包括对每个主体应用给定报告器后的值。（报告器可以是简单的变量名，或复杂的表达式 – 甚至可以是由[to-report](#)定义的过程）。一种常用的形式如：

```
max [...] of turtles
sum [...] of turtles
```

等等。

使用[sentence](#)可以将两个列表组合成一个更大的列表。像[list](#)一样，[sentence](#)一般有两个输入参数，但也可以有任意多个输入参数，只是整个调用要用括号括起来。

改变列表项，Changing list items

技术上讲，列表不能修改，但可以基于旧列表创建新列表。如果想用新列表替换旧列表，使用[set](#)。例如：

```
set mylist [2 7 5 Bob [3 0 -2]]
; mylist is now [2 7 5 Bob [3 0 -2]]
set mylist replace-item 2 mylist 10
; mylist is now [2 7 10 Bob [3 0 -2]]
```

报告器[replace-item](#) 有三个输入项，第一项指定列表的哪个元素要修改，0 是第 1 个，1 是第 2 个，…。

要在列表尾部加一项，使用[lput](#)。（[fput](#)在列表首部加一项）

```
set mylist lput 42 mylist
; mylist is now [2 7 10 Bob [3 0 -2] 42]
```

如果你改主意了怎么办？[but-last](#)（简写为b1）返回除最后一项外的列表。

```
set mylist but-last mylist
; mylist is now [2 7 10 Bob [3 0 -2]]
```

假设要除去项 0，即列表首部的 2，

```
set mylist but-first mylist
; mylist is now [7 10 Bob [3 0 -2]]
```

假设你要将列表中的第 3 项列表的第 3 项从-2 改为 9。关键是认识到用来调用嵌入列表 [3 0 -2] 的语法是 item 3 mylist，然后[replace-item](#)可以嵌入改变列表中的列表。为清晰起见，加上了括号。

```
set mylist (replace-item 3 mylist
(replace-item 2 (item 3 mylist) 9))
; mylist is now [7 10 Bob [3 0 9]]
```

列表遍历，Iterating over lists

要对列表中的每个元素依次进行某项操作，使用[foreach](#)命令和[map](#)报告器。

[foreach](#)用来对列表中的每个元素执行命令，他的输入参数包括输入列表和命令块，如下：

```

foreach [2 4 6]
[ crt ?
  show (word "created " ? " turtles") ]
=> created 2 turtles
=> created 4 turtles
=> created 6 turtles

```

命令块中的变量?保存输入列表中的当前值。

更多关于[foreach](#)的例子：

```

foreach [1 2 3] [ ask turtles [ fd ? ] ]
;; turtles move forward 6 patches
foreach [true false true true] [ ask turtles [ if ? [ fd 1 ] ] ]
;; turtles move forward 3 patches

```

[map](#)与[foreach](#)相似，但它是报告器。它的输入参数包括一个输入列表和一个其他报告器。

与[foreach](#)不同，报告器在前，如下：

```

show map [round ?] [1.2 2.2 2.7]
;; prints [1 2 3]

```

[map](#)返回一个列表，这个列表对输入列表中的每一项应用报告器的结果。也是使用?引用列表的当前项。

关于[map](#)的另一个例子：

```

show map [? < 0] [1 -1 3 4 -2 -10]
;; prints [false true false false true true]

```

并不是在操作整个列表时都要使用[foreach](#) 和 [map](#)。有时需要使用其他技术，如[repeat](#) 或 [while](#)，以及递归等。

原语[sort-by](#)的语法与[map](#) 和 [foreach](#)相似，但由于要比较两个对象，使用了两个特殊变量?1 和 ?2，而不是?。

一个[sort-by](#)的例子：

```

show sort-by [?1 < ?2] [4 1 3 2]
;; prints [1 2 3 4]

```

可变数量输入， Varying number of inputs

有些涉及列表和字符串的命令和报告器输入参数的数量可变。这种情况下，为了传给它们一系列参数而不是默认数量的参数，原语和输入参数必须全部用括号括起来。一些例子如下：

```

show list 1 2
=> [1 2]

```

```

show (list 1 2 3 4)
=> [1 2 3 4]
show (list)
=> []

```

注意这些特殊命令都有默认数量输入，这时不用括号。有这种能力的原语有[list](#), [word](#), [sentence](#), [map](#)和[foreach](#)。

主体列表，Lists of agents

以前我们说过主体集合总是随机顺序，每次的顺序都不一样。如果要主体们按固定的顺序执行动作，需要构造一个主体列表。

这样做需要两个原语[sort](#) 和 [sort-by](#)的帮助。

[sort](#) 和[sort-by](#)以主体集合为输入，结果是一个新列表，它包括主体集合中的所有主体，但主体按特定顺序排列。

如果在海龟主体集合上应用[sort](#)，得到的是按 `who number`升序排列的主体列表。

如果在瓦片主体集合上应用[sort](#)，得到的是按从左到右，从上到下排列的瓦片列表。

如果在链主体集合上应用[sort](#)，得到的是先按[end1](#)再按[end2](#)升序排列的链列表，剩下的节（ties）分解为种类，按种类的声明顺序排列。

如果要降序排列，需要将[reverse](#) 和 [sort](#)组合起来，例如`reverse sort turtles`。

如果要按非标准的其他准则排列，需要使用[sort-by](#)。

例子：

```
sort-by [[size] of ?1 < [size] of ?2] turtles
```

得到的列表按海龟变量[size](#)升序排列。

请求主体列表，Asking a list of agents

有了主体列表后，可能要请求列表中的所有主体执行动作。方法是组合使用[foreach](#) 和 [ask](#) 命令，像这样：

```

foreach sort turtles [
  ask ? [
    ...
  ]
]
```

这就按 `who number` 升序依次请求每个海龟。把“turtles”换成“patches”后，就从左到右、从上到下依次请求瓦片。

如果这样使用 `foreach`，列表中的主体就顺序执行 `ask` 中的命令，而不是并行。一个主体执行完后，下一个才开始执行。

不能直接对海龟列表应用[ask](#)，[ask](#)只能对主体集合或单个主体工作。

列表性能，Performance of lists

如果你的模型大量使用列表，特别是长列表，你需要知道 NetLogo 各种列表操作的速度，

这样能帮你写出运行速度较快的代码。

NetLogo 列表是单链表。这是个计算机术语，意味着如果要查找表中的某项，必须从头开始一个一个查找，直到找到。例如要找到第 100 项，NetLogo 必须走过前面 99 项。

这也意味着某些操作的效率特别高，就是那些在表头的操作。[first](#), [but-first](#) 和 [fput](#) 操作很快，不管表有多长，花费时间相同。因此如果通过每次增加一个新项来构建列表，则使用 [fput](#) 要快于 [lput](#)。（如果表的顺序与你期望的相反，可以使用 [reverse](#) 反转列表）。

[length](#) 报告器也很快，NetLogo 总是跟踪列表的长度，因此不需计算。

长列表上的一些报告器较慢，包括 [item](#), [lput](#), [but-last](#), [last](#) 和 [one-of](#)。

数学 (Math)

NetLogo 的所有数值在内部都是双精浮点数，遵循 IEEE 754 标准。每个数有 64 位，包括 1 个符号位，11 位指数，52 位尾数。细节见 IEEE 754。

NetLogo 的整数是恰好没有小数部分的数，3 和 3.0 没区别。（这与人们日常使用一致，但与一些编程语言不同，有些语言对整数和浮点数区别对待）。

NetLogo 打印输出整数时没有尾部的".0":

```
show 1.5 + 1.5
observer: 3
```

在需要整数的部分使用带有小数部分的数，则只是简单的将小数部分丢弃。例如，crt 3.5 创建 3 个海龟，额外的 0.5 忽略。

整数的范围是 $+/-9007199254740992$ (2^{53} , 约 9 乘 10 的 15 次幂)。计算时超出范围不会引起运行错误，但可能会损失精度，因为 64 位限制，最后 1 位会舍入。数太大时，会导致不精确的结果：

```
show 2 ^ 60 + 1 = 2 ^ 60
=> true
```

如果有小数部分，对小一点的数的计算也会产生令人惊讶的结果，因为不是所有分数都能精确表示，还有舍入发生。例如：

```
show 1 / 6 + 1 / 6 + 1 / 6 + 1 / 6 + 1 / 6 + 1 / 6
=> 0.9999999999999999
show 1 / 9 + 1 / 9 + 1 / 9 + 1 / 9 + 1 / 9 + 1 / 9 + 1 / 9 + 1 / 9 + 1 / 9
=> 1.0000000000000002
```

任何产生无穷大或非数的操作会引起运行错误。

科学计数法，Scientific notation

很大或很小的数用科学计数法表示，例如：

```
show 0.000000000001
=> 1.0E-12
show 50000000000000000000000000000000
=> 5.0E19
```

科学计数法表示的数通过“E”识别，表示10的几次幂。例如1.0E-12就是1.0乘以10的-12次幂：

```
show 1.0 * 10 ^ -12
=> 1.0E-12
```

在你的代码里也可使用科学计数法：

```
show 3.0E6
=> 3000000
show 8.123456789E6
=> 8123456.789
show 8.123456789E7
=> 8.123456789E7
show 3.0E16
=> 3.0E16
show 8.0E-3
=> 0.0080
show 8.0E-4
=> 8.0E-4
```

这些例子展示了有分数部分的数当指数小于-3或大于6时的科学计算法表示。超过NetLogo整数范围-9007199254740992—9007199254740992($+/-2^{53}$)的数也可以用科学计算法表示。

```
show 2 ^ 60
=> 1.15292150460684698E18
```

输入数时，字母E大小写都行。打印数值时，NetLogo总是使用大写E：

```
show 4.5e20
=> 4.5E20
```

浮点精度，Floating point accuracy

因为NetLogo中的数值受限于浮点数的二进制表示，有时会得到不精确的结果。例如：

```
show 0.1 + 0.1 + 0.1
=> 0.3000000000000004
show cos 90
=> 6.123233995736766E-17
```

这是浮点算术固有的问题，所有使用浮点数的编程语言都有这样的问题。

如果要处理固定精度的数，例如美元和分，一个常用技术是在内部只使用整数（分），在显示时除以100得到元。

如果必须使用浮点数，有些情况下不要使用 $x = 1$ [...]这样的相等比较，而是能有一点容限，如采用 $if abs(x - 1) < 0.0001$ [...]。

另外，使用[precision](#)可以方便的控制数字显示。NetLogo的监视器根据配置的小数位对数字进行舍入后显示。

随机数 (Random Numbers)

NetLogo 使用伪随机数（计算机编程基本都这样）。意思是看起来随机，实际上是由确定性过程决定的。确定性是指如果每次从同一个随机数种子开始，你可以得到同样的结果。一会儿我们解释随机数种子的含义。

在科学模型里，伪随机数实际上正是所期望的。这是因为科学实验的可重复性非常重要，任何人都可以去尝试，得到相同的结果。因此 NetLogo 使用伪随机数，你做的“实验”可以被别人重现。

工作原理如下。NetLogo随机数发生器从一个种子数开始，种子数可以是任何整数。通过[random-seed](#)给定种子后，随机数发生器总是产生同样的随机数序列。例如下面的命令：

```
random-seed 137
show random 100
show random 100
show random 100
```

你总是得到数列 95, 7, 54。

注意只有 NetLogo 版本相同才能这样保证。有时我们改变 NetLogo 版本时采用了不同的随机数发生器。（目前使用 Mersenne Twister 发生器）

使用[new-seed](#)报告器产生适合随机数发生器使用的种子值。[new-seed](#)基于当前日期和时间产生均匀分布的种子。在同一行里它不会产生两个相同的种子。

代码例子 Code Example: Random Seed

如果没有设置随机数种子，NetLogo 基于当前日期和时间产生种子。没办法找到到底使用了哪个种子，因此如果你想让模型可重现的话，必须自行设置随机数种子。

NetLogo 中包含“random”的原语 (random, random-float 等等) 使用伪随机数，除此之外还有其他操作也做随机选择。例如，主体集合 (agentset) 总是采用随机顺序，[one-of](#) 和 [n-of](#) 随机选择主体，[sprout](#) 产生海龟，其颜色和方向是随机的，[downhill](#) 当有多个瓦片时随机选一个。所有这些随机选择都受随机数种子的控制，因此模型可再现。

[random](#) 产生均匀分布整数，[random-float](#) 产生均匀分布浮点数。此外还有几个随机分布。在词典里查看[random-normal](#), [random-poisson](#), [random-exponential](#) 和 [random-gamma](#).

辅助随机数发生器，Auxiliary generator

按钮和命令中心的代码使用主随机数发生器。

监视器中的代码使用辅助随机数发生器。因此即使监视器执行使用随机数的计算，对模型输出也没有影响。滑动条的代码也是如此。

局部随机化， Local randomness

你可能要明确指定部分代码不要影响主随机数发生器的状态，使得模型输出不受影响。使用[with-local-randomness](#)达到该目的，具体见词典。

海龟图形（Turtle shapes）

NetLogo 里海龟图形是矢量化的，它们是由基本几何形状，方、圆、线等组成的，而不是由像素点阵构成的。矢量图形完全可缩放和旋转。NetLogo 缓存矢量图形大小为 1, 1.5, 2 的位图，这样能加快执行速度。

海龟的图形保存在变量[shape](#)中，可以用[set](#)命令设置。

新海龟有个图形“default”。[set-default-shape](#)原语可改变默认图形，或让每个海龟种类有各自的默认图形。

[shapes](#)原语返回模型里当前可用图形的列表。这很有用，例如要给海龟分配一个随机图形：

```
ask turtles [ set shape one-of shapes ]
```

使用海龟图形编辑器 Turtle Shapes Editor 创建自己的海龟图形，或从图形库里选图形加入模型，或者在模型之间传递图形。详情参见本手册的 Shapes Editor 部分。

画矢量图形所使用的线宽由[set-line-thickness](#) 控制。

代码例子 Code Examples: Breeds and Shapes, Shape Animation

链图形（Link Shapes）

链图形与海龟图形相似，只不过要用链图形编辑器（Link Shape Editor）创建和编辑。链图形由 0-3 条线¹和方向标志组成，每条线可以有不同的线型，方向标志由与海龟图形一样的元素组成。链也有一个[shape](#)变量，可以设置为模型中的任何一个链图形。默认情况下，链有个默认（“default”）图形，当然你可以使用[set-default-shape](#)做出改变。[link-shapes](#)报告器返回当前模型中的所有链图形。

链变量[thickness](#)控制链图形中的线宽。

时钟计数器（Tick Counter）

许多 NetLogo 模型中，时间是一个时间步一个时间步的向前推进，每个时间步称为一个“滴答”（tick）。NetLogo 内置时钟计数器，可以跟踪已经运行了多少个滴答。

目前的滴答值显示在视图上部（可以使用 Settings 隐藏它，或者将“ticks”修改为其他文字）

在程序中要获取当前时钟计数器的值，使用[ticks](#)报告器。[tick](#)命令将时钟计数器加 1。

¹ 译者注：链图形的三条线称为left,middle和right，它们并行排列。每条线可以为空，因此有 0-3 条。

[clear-all](#)命令将时钟计数器重设为 0²。如果只想让时钟变成 0，不想清除其他事情，使用[reset-ticks](#)命令。

如果模型设为基于时钟更新 (tick-based update)，[tick](#)命令通常也会更新视图。参见下一部分视图更新[View Updates](#)。

什么时候用 tick ， When to tick

我们建议将[tick](#)命令用在所有主体移动和行为结束之后，以及绘图和统计计算之前。这种情况下，当绘图或统计代码引用时钟时，得到的是新值，反映 tick 已完成。例如：

```
to go
  ask turtles [ move ]
  ask patches [ grow ]
  tick
  do-plots
end
to do-plots
  plotxy ticks count turtles
end
```

通过将[tick](#)放在do-plots之前，绘图代码使用[ticks](#)得到了正确的时钟计数。

非整数时钟， Fractional ticks

多数模型里时钟从 0 开始每次加 1，因此总是整数。但是时钟也可以为浮点数。

如果将时钟推进一个非整数值，需要使用[tick-advance](#)命令，该命令需要一个数值型输入参数指明时钟推进多少。

非整数时钟值的一类典型应用是逼近连续或曲线运动。例如模型库中的 GasLab 模型（在 Chemistry & Physics 部分）。这些模型计算下一事件发生时间的精确值，将时钟推进到那一刻。

视图更新 (View Updates)

视图是用来在计算机屏幕上查看模型中的主体的。当主体移动和改变时，在视图里能看到。

当然不是直接看到主体，视图是 NetLogo 绘制的图像，给你显现某一瞬间主体怎样的。当该时刻过去后，主体移动、改变了，图像需要重新绘制。图像的重画就叫做更新 ("updating") 视图。

视图什么时候更新？本节讨论 NetLogo 怎么决定何时更新，更新时你能怎么影响它。

NetLogo 有连续 (continuous) 和基于时钟 (tick-based) 两种更新模式。在界面页顶

² 译者注：clear-all 清除很多东西，时钟只是其中之一。

部的弹出菜单里进行模式切换。

启动 NetLogo 或启动一个新模型后，默认是连续更新模式。然而模型库中的几乎所有模型都采用基于时钟的模式。

连续更新模式最简单，而基于时钟模式让你对更新时刻及更新频率有更多的控制权。

到底什么时刻进行更新非常重要，因为更新时刻决定了屏幕显示。如果更新发生在不合适的时刻，你可能看到不希望看到的结果 – 也许乱糟糟或误导你。

更新频率也很重要，因为更新需要花费时间。如果在更新上花费时间太多，你的模型运行就很慢。更新越少，模型运行越快。

连续更新， Continuous updates

连续更新很简单，NetLogo 每秒更新视图确定的次数。默认情况下，当速度滑动条位于中间时，每秒更新 50 次。

如果放慢速度滑动条，更新就多于 50 次/秒，模型显著变慢。放快速度滑动条，更新就少于 50 次/秒。最快设置下，几秒才更新一次。

在极慢设置下，NetLogo 更新的非常频繁，能看到主体每次的移动（或变色等）。

如果要临时关闭连续更新，使用no-display命令。display命令打开更新，也强制立即更新（除非用户使用速度滑动条快进）

基于时钟的更新， Tick-based updates

在时钟计数器部分我们说过，NetLogo 许多模型的时间是按小间隔推进的，一个小间隔叫滴答（ticks）。一般情况下你希望每个滴答视图更新一次。这就是基于时钟更新的默认行为。

如果需要额外的更新，可以使用[display](#)命令强制更新（如果使用速度滑动条快进，这些更新会被跳过）。

使用基于时钟的更新时，你不必非要使用时钟计数器。如果时钟计数器不推进，视图只在你使用[display](#)时更新。

如果将速度滑动条设快，NetLogo 会跳过正常情况下会发生的一些更新。然而将速度滑动条设慢，却不会增加额外的更新，只是在每次更新之间插入暂停。速度设置的越慢，暂停时间越长。

即使在基于时钟的模式下，当按钮（一次性和永久性）弹起以及在命令中心完成命令输入时，视图也会更新。因此不必为那些不推进时钟的一次性按钮增加[display](#)命令。许多不推进时钟的永久性按钮确实需要[display](#)命令。例如模型库中的Life 模型（在Computer Science → Cellular Automata部分）。用户需要进行绘图的永久性按钮要使用[display](#)命令，才能让用户看到所绘图形，尽管没有推进时钟。

选择更新模式， Choosing a mode

基于时钟的模式的优点有：

1. 在不同计算机上，不同重复运行中视图更新行为都是一致的、可预测的。
2. 连续更新模式会让用户看到不想看到的内容，可能会让用户发懵或造成误导。
3. 运行更快。如果每个滴答更新一次就足够的话，使用该模式能减少花费在更新上的时间。
4. 因为setup按钮不推进时钟，它不受速度滑动条的影响，这正是我们期望的行为。

上面已经说过，模型库中的模型多数采用基于时钟的视图更新方式。

时钟没有按小间隔推进的模型适合采用连续更新模式。例如模型库中的 Termites 模型。（然而在 State Machine Example 模型里，显示了怎样将 Termites 重写为基于滴答的方式。）

即使那些正常情况下采用基于时钟的视图更新模型，为了调试也可以临时改为连续模式。观察滴答之间发生了什么，而不是仅观察滴答时的情况，有助于解决麻烦。切换为连续模式后，可能要拖慢速度滑动条，使你能看清每个主体的移动。由于模式存储在模型里，记住切换回来。

绘图（Plotting）

NetLogo 有绘图功能，你可以通过图形（plot）来了解模型中所发生的事情。

绘图前要先在界面页中创建一个或多个图形。每个图形应该有唯一的名字。在例程页编写的代码中通过图名指定所要操作的图形

要了解关于在界面页中编辑图形的更多信息，参见界面指南（[Interface Guide](#)）。

指定图形， Specifying a plot

如果模型中只有一个图形，那么马上就可以绘图。但是如果多个图形的话，必须指定要在哪个图形上绘制，方法是使用 `set-current-plot` 命令，加上用双引号括起来的图名，像这样：

```
set-current-plot "Distance vs. Time"
```

所用的图名必须与创建图形时输入的图名完全一样。注意如果你以后修改了图名，则也必须修改 `set-current-plot` 调用，以使用这个新名字。（使用复制、粘贴很方便）。

指定画笔， Specifying a pen

新建图形后只有一支画笔。如果当前的图形只有一支画笔，则马上可以开始绘图。

但是也可以让一个图形拥有多支画笔。使用图形编辑对话框下部的“Plot Pens”控件可以增添画笔。每支画笔的名字必须唯一，在例程页的代码里使用画笔名指定画笔。

对于拥有多支画笔的图形，必须指定使用哪支画笔进行绘图。如果不指定的话，就使用

第一支画笔。要用不同的画笔，需要使用[set-current-plot-pen](#)命令加上由双引号括起来的画笔名。像这样：

```
set-current-plot-pen "distance"
```

画点，Plotting points

实际画图时要使用的两个基本命令是[plot](#) 和 [plotxy](#)。

使用[plot](#)命令只需给定y值。所画第一个点的x值自动为0，第二点为1，…。（如果笔的间隔（interval）是默认值时就是这样的，你也可以改变间隔值）。

当模型的每个时间步要画一个点时，使用[plot](#)命令特别方便。例如：

```
to setup
  ...
  plot count turtles
end

to go
  ...
  plot count turtles
end
```

注意此处在“setup”和“go”例程里都进行了绘制，这是因为我们希望该图形包含系统初始状态。我们在go例程结束时绘制，而不是在开始时，这是因为当go按钮停止时，我们得到的是系统的最新状态。

如果要同时指定所绘点的x值和y值，应该使用[plotxy](#)。

代码实例 Code Example: Plotting Example

其他绘图种类，Other kinds of plots

默认时NetLogo画笔使用线形模式，所画的点用线连起来。

如果只想移动画笔而不画图，就使用[plot-pen-up](#)命令。该命令发出后，[plot](#) and [plotxy](#)只移动画笔，不画图。当画笔移动到你希望画图的地方，再使用[plot-pen-down](#)将画笔放下。

如果想只画点、不画线，或者想画条形（bar），那就需要改变画笔模式。共有三种画笔模式：线、条形、点。默认模式是线。

一般通过编辑图形（plot）来改变画笔模式，这样就改变了画笔的默认模式。也可使用[set-plot-pen-mode](#)命令临时改变画笔模式，该命令需要一个数值型参数：0是线，1是条，2是点。

直方图，Histograms

直方图是一类特殊的图，用来表示一组数值中的每个数或不同范围内的数出现的频率。

例如，假设模型里的海龟有年龄变量，可以使用[histogram](#)命令产生海龟年龄分布

```
histogram [age] of turtles
```

直方图所表达的数不一定来自主体集合 (agentset)，也可以来自数值型列表 (list of numbers)。

注意使用[histogram](#)命令并不自动将画笔模式切换为条形模式，必须自己设置画笔模式为条形。（前面说过，可以在界面页中编辑图形，改变画笔默认模式）。

直方图中条形的宽度由画笔间隔决定。可以在界面页中编辑图形，改变画笔默认间隔。也可使用[set-plot-pen-interval](#) 或[set-histogram-num-bars](#)命令临时改变画笔间隔。使用[set-histogram-num-bars](#)时，NetLogo根据当前x范围内给定的条形数目计算出条形的近似宽度。

代码实例 Code Example: Histogram Example

清除和重设，Clearing and resetting

使用[clear-plot](#)命令清除当前图形，[clear-all-plots](#)命令清除所有图形。[clear-all](#)命令清除模型中的所有东西，当然也包括所有图形。

如果要清除当前图上已经画上的点，使用[plot-pen-reset](#)。

清除图形或重设画笔不只是清除已经画过的数据，还将图形或画笔重设为默认设置，这些默认设置是在界面页中创建或最后编辑图形时决定的。因此像[set-plot-x-range](#) 和[set-plot-pen-color](#) 这样的命令，其效果是临时的。

自动绘图，Autoplotting

默认情况下 NetLogo 图形的自动绘图 (autoplotting) 功能是激活的。自动绘图的含义是：当要画的点超过当前显示范围时，图形的范围（一个坐标轴或两个坐标轴）自动增加，以使新点可见。

为避免每次增加新点时图形范围都要变化，增加时留有一定的余地。增加时水平方向多增加 25%，垂直方向多增加 10%。

若要关闭自动绘图功能，则编辑图形反选 Autoplot 项。目前不能只对一个坐标轴选择激活/关闭，而是必须对两个坐标轴同时设定。

临时画笔，Temporary plot pens

多数图形始终有固定数目的画笔。但有些图形有更复杂的需求，需要画笔数量随条件而变。这种情况下你可以编写代码创建临时画笔，然后使用它们。这些画笔称为“临时”的原因在于当清除图形（使用[clear-plot](#), [clear-all-plots](#)或[clear-all](#)）后，它们就消失了。

使用[create-temporary-plot-pen](#)命令创建临时画笔，一旦创建，使用方法与其他画笔无异。画笔默认是放下的、黑色、间隔为 1、使用线模式，有些命令用来改变它们，参见NetLogo词典的Plotting部分。

使用图例，Using a Legend

在图形编辑对话框中勾选 “Show legend” 显示图例。如果不让某支笔的图例显示出

来，在编辑框里反选那支笔的“Show in Legend”选项即可。

结论，Conclusion

此处对绘图的介绍并不全面，更多的信息参见 NetLogo 词典的 Plotting 部分。

模型库的模型实例演示了许多高级绘图技术。看看下面的例子：

代码实例 Code Examples: Plot Axis Example, Plot Smoothing Example, Rolling Plot Example

字符串（Strings）

要输入字符串常量，用双引号括起来。

在一对双引号内什么都没有，则是空串，即“”。

大多数关于列表(list)的原语同样可用于字符串：

```
but-first "string" => "tring"
but-last "string" => "strin"
empty? "" => true
empty? "string" => false
first "string" => "s"
item 2 "string" => "r"
last "string" => "g"
length "string" => 6
member? "s" "string" => true
member? "rin" "string" => true
member? "ron" "string" => false
position "s" "string" => 0
position "rin" "string" => 2
position "ron" "string" => false
remove "r" "string" => "sting"
remove "s" "strings" => "tring"
replace-item 3 "string" "o" => "strong"
reverse "string" => "gnirts"
```

有些字符串专用的原语，如[is-string?](#), [substring](#)和[word](#):

```
is-string? "string" => true
is-string? 37 => false
substring "string" 2 5 => "rin"
word "tur" "tle" => "turtle"
```

字符串可以进行比较，操作符有 `=`, `!=`, `<`, `>`, `<=`, and `>=`。

如果要嵌入特殊字符，使用转义序列：

- `\n` = 新行
- `\t` = tab
- `\"` = 双引号
- `\\"` = 反斜线

输出 (Output)

这一部分讲屏幕输出。使用[export-output](#)命令可以将屏幕保存到文件。如果需要更灵活的将数据输出到外部文件的方法，参见下一部分文件输入输出 ([File I/O](#))。

NetLogo中产生屏幕输出的基本命令是[print](#), [show](#), [type](#) 和[write](#)，这些命令将输出送到命令中心。

要了解这些命令的细节，参见 NetLogo 词典的条目。下面是典型用法：

- [print](#) 多数情况下都可以使用
- [show](#) 让你看到每个主体在输出什么
- [type](#) 让你一行里输出几项内容
- [write](#) 按某种格式输出数值，这些值可以被[file-read](#)读回

NetLogo模型也可以在界面页中有一个输出区域 ("output area")，这个区域是与命令中心分开的。如果要输出到这个区域而不是命令中心，使用[output-print](#), [output-show](#), [output-type](#) 和[output-write](#)命令。

输出区域可以使用[clear-output](#) 命令清除，也可使用[export-output](#) 存入文件。输出区域的内容使用[export-world](#) 命令保存。[import-world](#) 命令将清除输出区域，将其内容设为输入世界文件 (imported world file) 的内容。注意将太多的数据送到输出区域会导致输出世界文件较大。

如果模型没有独立的输出区域，而使用[output-print](#), [output-show](#), [output-type](#), [output-write](#), [clear-output](#)或[export-output](#) 命令，这些命令将作用到命令中心。

文件输入输出 (File I/O)

NetLogo 提供了一些与外部文件进行交互的原语，这些原语的前缀都是 `file-`。

主要有两种处理文件的方式：读和写，区别在数据流的方向。读是指将数据从文件中取出，流到模型中。写是指将数据从模型输出到文件中。

当 NetLogo 模型作为 applet 在浏览器中运行时，只能从模型所在的目录文件读数据，而不能写到任何文件。

当操作文件时，必须首先使用[file-open](#) 原语指定要操作哪个文件。除非先打开文件，否则其他文件操作无法进行。

下面的 `file-` 原语指明文件的模式，直到关闭、读或写。要切换模式，关闭然后重新打开文件。

读原语包括 [file-read](#), [file-read-line](#), [file-read-characters](#) 和 [file-at-end?](#) 。注意在打开和读操作之前，这些文件必须存在。

代码实例 Code Examples: File Input Example

文件写原语与屏幕输出原语类似，只不过输出到文件，包括 [file-print](#), [file-show](#), [file-type](#) 和 [file-write](#) 。注意不可能覆盖（overwrite）数据。也就是说，如果写入一个已经有数据的文件，新数据只能添加在尾部。（如果要覆盖一个文件，使用 [file-delete](#) 删除它，然后打开进行写操作）

代码实例 Code Examples: File Output Example

不再使用文件时，使用 [file-close](#) 结束会话。如果要删除文件，使用 [file-delete](#) 。要关闭多个打开的文件，先用 [file-open](#) 选择文件，然后关闭。

```
;; Open 3 files
file-open "myfile1.txt"
file-open "myfile2.txt"
file-open "myfile3.txt"

;; Now close the 3 files
file-close
file-open "myfile2.txt"
file-close
file-open "myfile1.txt"
file-close
```

或者，如果知道要全部关闭的话，使用 [file-close-all](#) 。

有两个命令 [file-write](#) 和 [file-read](#) 值得注意，它们用来方便的保存、获取NetLogo 常量，如数值、列表、布尔值等。`file-write` 总是以 `file-read` 可以正确解释的方式输出变量。

```
file-open "myfile.txt" ;; Opening file for writing
ask turtles
[ file-write xcor file-write ycor ]
file-close

file-open "myfile.txt" ;; Opening file for reading
ask turtles
[ setxy file-read file-read ]
```

`file-close`

代码实例 Code Examples: File Input Example and File Output Example

用户选择, Letting the user choose

原语[user-directory](#), [user-file](#)和[user-new-file](#) 让用户选择一个文件或目录, 你的代码在用户选择的文件或目录上操作。

电影 (Movies)

这一部分介绍将 NetLogo 模型捕获为 QuickTime 电影。

首先使用[movie-start](#)命令开始新电影, 所提供的文件名后缀为.mov, 这是QuickTime 电影的扩展名。

要向电影中加一帧, 使用[movie-grab-view](#)或[movie-grab-interface](#), 前者只显示视图, 后者显示整个界面页。在一个电影里, 必须只使用一种[movie-grab-](#)原语, 不能混用。

所有帧加完后, 使用[movie-close](#)。

```
;; export a 30 frame movie of the view
setup
movie-start "out.mov"
movie-grab-view ;; show the initial state
repeat 30
[ go
  movie-grab-view ]
movie-close
```

电影的默认播放速度是 15 帧/秒, 用[movie-set-frame-rate](#) 设定不同的帧率。你必须在[movie-start](#) 之后和抓帧之前设定帧率。

[movie-status](#) 用来检查帧率或看看已抓了多少帧, 它返回一个描述当前电影状态的字符串。

要放弃电影或删除电影, 调用[movie-cancel](#)。

代码实例 Code Example: Movie Example

NetLogo电影是无压缩QuickTime文件。从Apple下载免费播放软件[QuickTime Player](#)播放它。

由于文件未压缩, 很占磁盘空间, 你可能想用第三方软件对它进行压缩。压缩时有多种选择, 有些是有损压缩, 有些是无损压缩。有损是指压缩时丢弃了一些细节, 文件较小但图像质量下降。有些模型不应使用有损压缩, 例如视图中包含精细的像素级的细节。

QuickTime Pro 是一款能在 Mac 和 Windows 平台上压缩 QuickTime 电影的软件。在 Mac 平台上, iMovie 也不错。

视角 (Perspective)

2 维和 3 维视图都采用观察者 (Observer) 视角。默认时，观察者在正Z轴向下俯视世界。可以使用[follow](#), [ride](#) 和[watch](#)这些观察者命令和 [ride-me](#) and [watch-me](#)海龟命令改变观察者的视角。在跟随 (follow) 或乘骑(ride)模式，观察者和目标主体一起在世界中移动。跟随和乘骑模式只在 3 维视图中有区别。在 3 维视图里，用户可以使用鼠标改变跟随距离，当跟随距离为 0 时就是乘骑模式。当观察者处于观看 (watch) 模式，它跟踪一个海龟的移动而自己不动。在这些模式里你会看到目标上的聚光灯，并且在 3 维视图里观察者面对目标主体。要知道哪个主体是焦点，使用[subject](#)报告器。

代码实例 Code Example: Perspective Example

绘画 (Drawing)

在绘画层，海龟可以制作可见标记。

在视图里，绘画层处于瓦片之上和海龟之下。初始时刻绘画层是空的、透明的。

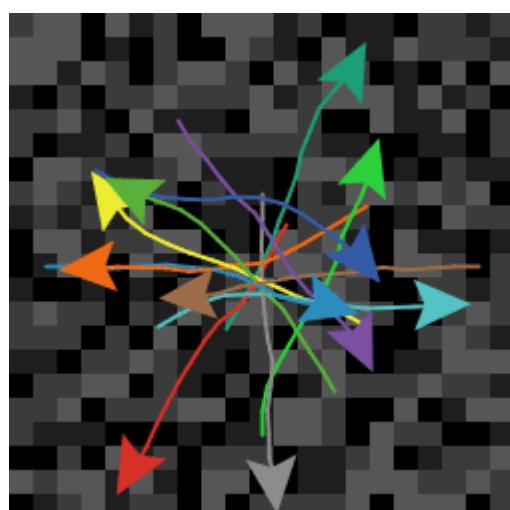
你能看到绘画层，但海龟（和瓦片）察觉不到绘画层，也不能对绘画层中的对象做出反应。绘画层只是用来给人看的。

海龟使用[pen-down](#) 和[pen-erase](#)命令在绘画层画线或擦除。如果海龟的画笔放下（或擦除），当他移动时就在身后画出（或擦除）线，线的颜色与海龟颜色一致。要停止画图（或擦除），使用[pen-up](#)。

正常情况下海龟画的线 1 个像素宽。在画图（或擦除）之前，设置海龟变量[pen-size](#) 可以改变线的粗细。新海龟该变量为 1。

当海龟没有沿固定方向移动时，例如[setxy](#) 或 [move-to](#)，所画的线就是按拓扑性质得到的最短路线。

这里有一些在随机灰度的瓦片上移动的海龟进行绘制的例子。注意海龟如何盖住了线，线如何盖住了瓦片。这里的[pen-size](#) 是 2。



[stamp](#) 命令让海龟在身后留下自身图像, [stamp-erase](#)擦除下面绘画层的像素。

要擦掉整个绘画层, 使用观察者命令[clear-drawing](#)。(也可使用[clear-all](#), 它也清除所有其他东西)

输入图像, Importing an image

观察者命令[import-drawing](#) 允许你从磁盘输入图像到绘画层。

[import-drawing](#) 只提供让人们观看的背景。要想实现海龟和瓦片与图像的交互, 使用[import-pcolors](#) 或 [import-pcolors-rgb](#)。

与其他 Logo 的比较, Comparison to other Logos

绘画层在 NetLogo 里与其他 Logo 有所不同。

显著区别包括:

- 新海龟的画笔是抬起的而不是放下的
- 要限制海龟绘画的边界, 需要编辑世界设置, 关闭回绕, 而其他Logo使用fence 命令来限制海龟绘画边界
- 没有screen-color, bgcolor, 或 setbg。可以通过指定瓦片颜色设定单色背景, 例如ask patches [set pcolor blue]

NetLogo 不支持的绘画功能:

- 没有 window命令。有些Logo使用该命令实现海龟在无限平面上的漫游
- 没有 flood 或fill命令着色封闭区域

拓扑 (Topology)

NetLogo 世界有四种拓扑类型: 环面(torus)、盒子(box)、垂直柱面(vertical cylinder)和水平柱面(horizontal cylinder)。通过打开或关闭 x, y 方向的回绕设定拓扑。世界默认是环面, 就像 NetLogo3.1 之前的设置一样。

环面在两个方向都回绕, 即世界的上下边界连在一起, 左右边界连在一起。因此如果海龟移出右边界就会出现在左边界, 上边界和下边界也是如此。

盒子在两个方向都不回绕, 世界是有界的, 因此海龟没法移出边界。注意边界上的瓦片少于 8 个邻元, 角上的只有 3 个邻元, 其他的有 5 个。

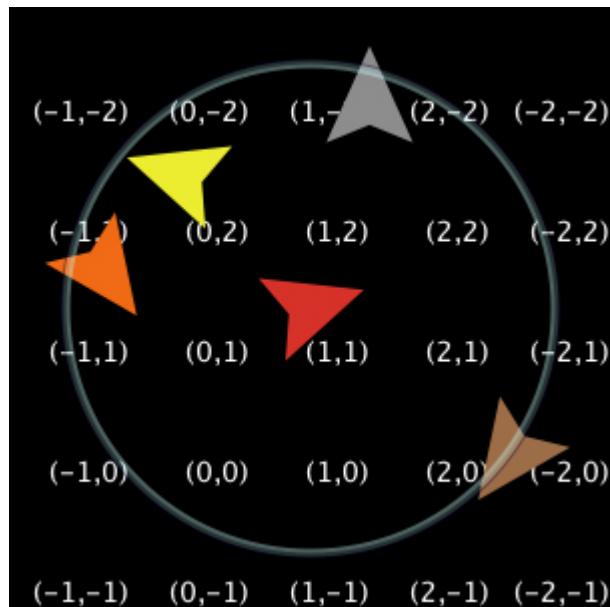
水平或垂直柱面只在一个方向回绕, 而另一个方向不回绕。水平柱面是垂直回绕, 即上下边界相连, 而左右不连。垂直柱面与此相反, 是水平回绕, 即左右边界相连, 但上下边界不连。

代码示例 Code Example: Neighbors Example

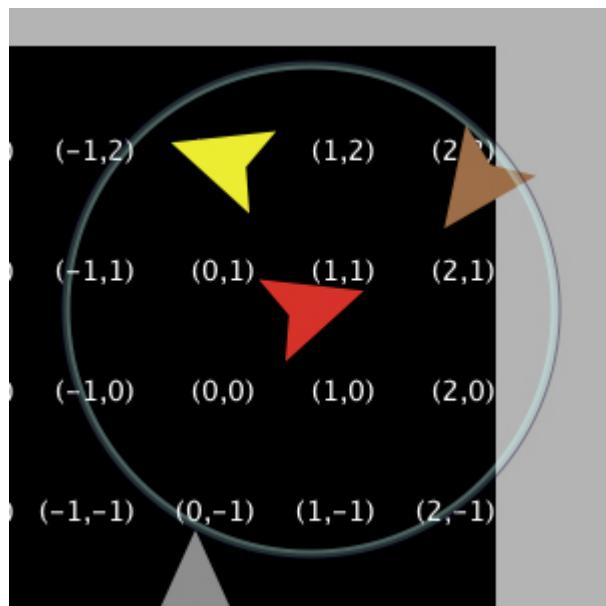
从 NetLogo 3.0 以后有设置用来激活回绕的可视化, 因此如果海龟图形跨越边界时, 海

龟的部分图形会显示在对面边界。(海龟本身是不占空间的点, 不可能处在世界的两个边界, 但海龟图形有大小, 所以看起来占用空间。)

当你跟随海龟时, 回绕会影响视图显示。在环面上, 不管海龟走到哪里, 你总是看到周围的世界:



然而在盒子或柱面世界, 世界有边, 超出世界范围的部分在视图里显示为灰色。



代码例子 Code Example: Termites Perspective Demo (torus), Ants Perspective Demo (box)

NetLogo 3.0 里的设置只控制回绕在视图上的显示, 而在 3.1 可以控制世界是否真的回绕, 即对向边是否相连。这些新设置决定了世界拓扑, 即环面 (torus)、盒子 (box)、垂

直柱面 (vertical cylinder) 和水平柱面 (horizontal cylinder)。这影响行为，而不仅影响模型的显示。

过去建模人员需要使用“no-wrap”原语，编写额外的代码模拟盒子型世界，提供了 `distance(xy)`, `in-radius`, `in-cone`, `face(xy)` 和 `towards(xy)` 的 No-wrap 版本。在 3.1 里不再需要这些版本，而是由世界拓扑决定这些原语是否回绕。它们总使用拓扑允许的最短路径。例如，给定 `pxcor` 和 `pycor` 最大最小值为 $+/-2$ ，那么各种拓扑下从右下角瓦片 (`min-pxcor`, `min-pycor`) 中心到左上角瓦片 (`max-pxcor`, `max-pycor`) 中心的距离为：

- 环面 - $\sqrt{2} \sim 1.414$ (不管世界规模多大，总是这个值。因为在环面上这两个瓦片对角相邻)
- Box - $\sqrt{\text{world-width}^2 + \text{world-height}^2} \sim 7.07$
- 垂直柱面 - $\sqrt{\text{world-height}^2 + 1} \sim 5.099$
- 水平柱面 - $\sqrt{\text{world-width}^2 + 1} \sim 5.099$

其他原语的行为和 `distance` 相似。如果你以前使用 no-wrap 原语，我们建议你删掉它，改用设置世界拓扑原语。

有几点原因推荐你这么做：

第一，我们期望如果你使用 no-wrap 原语时，你实际建立的模型世界不是环面。如果你使用适合世界性质的拓扑，NetLogo 自动进行边界检查，这样你的日子更好过，代码简单易理解，并且有可视线索帮助用户理解你的模型。注意即使有 no-wrap 原语，也很难对柱面世界建模，因为 no-wrap 原语只有两个方向都不回绕时才报告距离和方向。

你的模型可能有 bug。如果你混合使用了 no-wrap 和 wrap，你的模型或者没事，或者有 bug。(我们的模型就发现了几个 bug)。例如 Conductor 模型通过比较 `distance-no-wrap` 和 `distance` 判断下一个位置是否回绕，这样的话电子退出系统。这是个聪明的方法，但不幸的是有漏洞。电子在 y 方向回绕也会退出系统，而这是不对的。唯一正确的退出方式是到达线路左端的阴极。

如果去除 no-wrap 命令，则拓扑就不是模型的硬编码了，因此不需增加太多代码就可以对各种类型世界下的模型进行测试。(从环面到盒子还是要加上一些检查的，这一点在怎么转换部分解释。)

注意尽管我们把词典中的 no-wrap 删除了，但你还是可以使用它。我们这样做的目的是保证旧模型不加修改仍然可以运行。

怎样转换模型，How to convert your model

在 NetLogo 3.1，当第一次打开模型时，NetLogo 自动将各种情况下的 (`-screen-edge-x`)³ 变为 `min-pxcor`，将 `screen-edge-x` 变为 `max-pxcor` (对 y 同样如此)。尽管这与拓扑的改变没有直接关系，但你还是要考虑原点偏离中心是否会导致模型不合理。在 3.1 之前，世界必须是关于原点对称的，即世界的高度和宽度必须是奇数 (译者：因为中心瓦片占一个位置)。现在不需要这样了，你可以设定任何 min、max 组合，只要保证 (0, 0) 仍然在世界中。如果你的模型只需要一个或两个象限，或者只使用正坐标值以使模

³ 译者注： `-screen-edge-x` 指世界左边界 x 坐标，而 `screen-edge-x` 指右边界 x 坐标

型更简单，你可以考虑改变模型。如果你过去使用程序实现偶数网格，现在当然可以把这些程序去掉了。

Code Examples: Lattice Gas Automaton, Binomial Rabbits, Rugby

NetLogo3.1 增加了新的原语，使用这些原语改变拓扑很方便，当然其他时候也很方便。[random-pxcor](#), [random-pycor](#), [random-xcor](#) 和 [random-ycor](#) 返回最小最大(x和y)范围内的随机数。过去要实现海龟随机位置要依赖回绕，使用 `setxy random-float screen-size-x random-float screen-size-y`。如果某个方向不允许回绕，那就不行了。（产生运行错误，原因是视图将海龟布置在世界之外）。现在使用 `setxy random-xcor random-ycor`，不用管拓扑，也更简单、直接。

将模型转换为使用拓扑，必须首先决定哪种设置最合适。如果基于真实世界得到的答案不是太明显，（房间是盒子，导线是柱面），有几个线索对你有帮助。如果程序某处检查世界边界，或者某些瓦片不是视图上对面瓦片的邻居，大概是你没有使用环面。如果在 x 和 y 方向都进行边界检查，那就是盒子，如果只检查 x 方向，那就是水平柱面，如果只检查 y 方向，则是垂直柱面。

如果你使用了 `no-wrap`，可能你没有使用环面。然而如果既有 `no-wrap`，也有 `wrap`，则使用这个准则时要谨慎。也许是仅对可视元素使用 `no-wrap`，而其他部分仍是环面。

在你选定了拓扑并通过编辑视图改变后，也许还要对代码做一些修改。如果世界是环面，可能不需任何修改。如果模型只使用瓦片和扩散，可能不需任何修改。

如果模型有海龟到处移动，下一步是当它们达到边界时要决定发生什么。有几个常见的选择：海龟反射回世界（对称或随机），海龟退出系统（死亡），或海龟隐藏。这些时候就不需使用海龟坐标检查边界，而是判断海龟是否位于世界边界。有几个判断方法，最简单的是使用[can-move?](#)原语。

```
if not can-move? distance [ rt 180 ]
```

仅当海龟前方 `distance` 仍在世界之内时 `can-move?` 返回 `true`，否则返回 `false`。这种情况下，如果海龟位于边界，则原路返回。也可使用 `patch-ahead 1 != nobody` 代替 [can-move?](#)。如果要做更复杂的事情，而不是简单的返回，则使用 `patch-at with dx and dy`。

```
if patch-at dx 0 = nobody [
  set heading (- heading)
]
if patch-at 0 dy = nobody [
  set heading (180 - heading)
]
```

这是测试海龟是否撞上水平或垂直墙，并且从墙上弹回。

在一些模型里如果海龟不能移动则死亡（退出系统，例如例子 Conductor 或 Mousetraps）

```
if not can-move? distance[ die ]
```

如果使用 `setxy` 而不是 `forward` 移动海龟，则应该测试将移动到的瓦片是否存在，因为如果 `setxy` 给定的坐标不在世界之内则抛出运行错误。这常见于模拟无限平面时，超出视图的海龟只是简单隐藏。

```

let new-x new-value-of-xcor
let new-y new-value-of-ycor

ifelse patch-at (new-x - xcor) (new-y - ycor) = nobody
  [ hide-turtle ]
  [ setxy new-x new-y
    show-turtle ]

```

模型库里有几个模型使用这项技术。如 Gravitation, N-Bodies 和 Electrostatics。

通过使用不同的拓扑，你可以自由扩散 (diffuse) (过去很难做到)。每个瓦片扩散将等量的扩散变量给它的邻居，如果少于 8 个邻居 (使用 diffuse4 则是 4 个) 则剩余部分保持在消散瓦片上。这意味着整个世界瓦片变量之和保持常数。如果有特殊代码处理扩散则去掉它们。但是如果扩散物质要从边界落下，则仍然需要每一步进行边界清除，如例子 Diffuse Off Edges。

链 (Links)

链是连接两个海龟的主体，这两个海龟称为节点 (node)，链总是画为两个海龟之间的连线。与海龟不同，链没有位置，也不在任何瓦片之上，也不能查找链和某个点之间的距离。

有两种风格的链：无向 (undirected) 和有向 (directed)。有向链出自 (*out*) 或源自 (*from*) 一个节点，进入 (*into*) 或到达 (*to*) 另一个节点。如父子关系就可以用有向链表示。无向链对两个节点看起来都一样，每个节点与另一个节点之间有链。配偶或同胞关系就是无向链。

和 turtles 或 patches 一样，所有链有全局主体集合。使用 [create-link-with](#) 和 [create-links-with](#) 命令创建无向链，创建有向链则使用 [create-link-to](#), [create-links-to](#), [create-link-from](#) 和 [create-links-from](#) 命令。一旦第一个链创建为有向或无向，则所有未分种类 (unbreeded) 的链必须一致 (译者：全为有向或全为无向)。（链也支持种类，后面简短讨论）。未分种类的链不可能有的有向，有的无向。如果这样会出现运行错误。（如果所有未分种类的链死亡了，则能创建与以前的链风格（译者：指有向、无向）不同的链）。

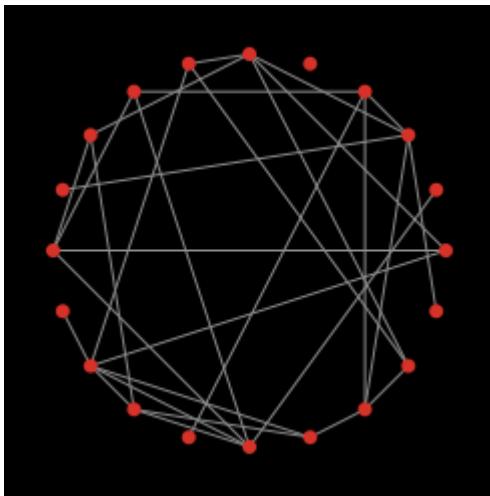
与有向链有关的原语名字里有“in”，“out”，“to”，“from”。无向链原语没有这些，或使用“with”。

链的种类，如海龟种类一样，使你可以定义不同种类的链。链种类必须设定为有向或无向。使用关键词 [undirected-link-breed](#) 和 [directed-link-breed](#) 声明链种类。使用 [create-<breed>-with](#) 和 [create-<breeds>-with](#) 创建无向的有种类链，使用 [create-<breed>-to](#), [create-<breeds>-to](#), [create-<breed>-from](#) 和 [create-<breeds>-from](#) 创建有向的有种类链。

一对主体之间的同种类的无向链不能超过 1 个 (无种类的链不能超过 2 个)，一对主体之间也不能有超过 1 个的同向的同种类的有向链。在一对主体之间，可以有两个同种类 (或者两个未分种类的链) 的反向的有向链。

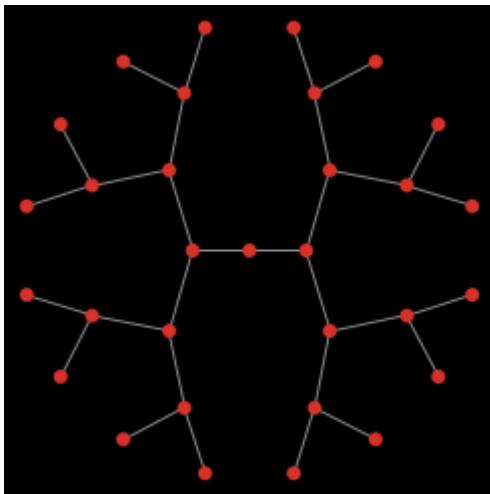
布局, Layouts

作为实验性的网络支持功能的一部分，我们也提供了几个不同的原语，帮你实现网络可视化。最简单的原语是[layout-circle](#)，它以世界中心为中点，根据给定的半径，均匀地将主体布局为圆形。

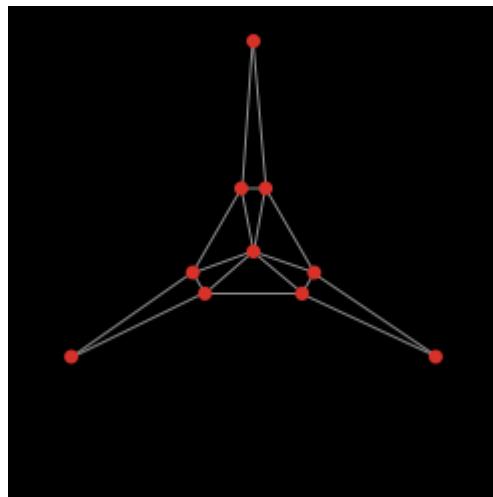


对某些类似树状的结构使用[layout-radial](#) 可以实现不错的布局。即使树中有一些圈，它也可以工作，只是圈越来越多看起来不太好。[layout-radial](#)将一个根主体（root agent）作为中心节点放在(0, 0)处，按同心圆模式安排其他节点。距离根节点1度的节点安排在离中心最近的圆上，2度的在第二层，依次进行。

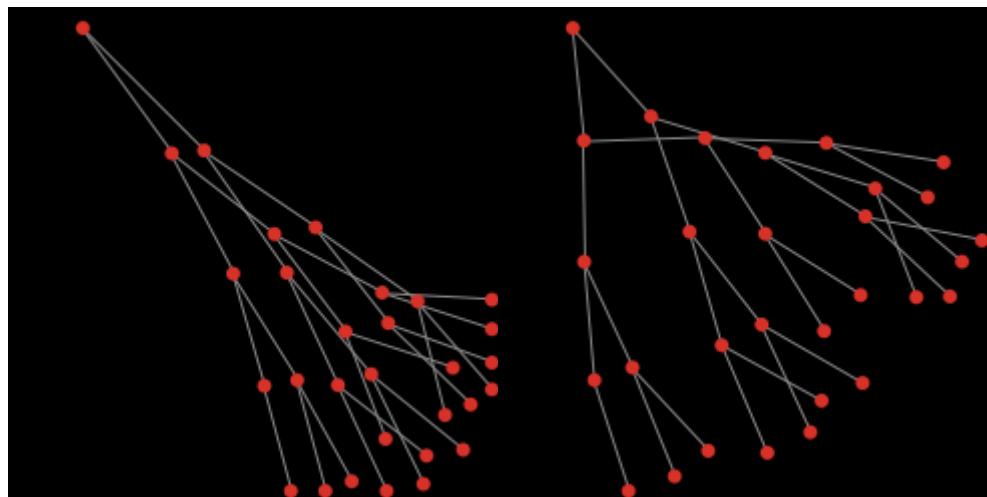
[layout-radial](#)试图考虑图的不均匀性，给较宽的分支分配较大的空间。[layout-radial](#) 也可以将某个种类做输入参数，这样只使用某个种类的链对网络进行布局。



给定一组支撑（anchor）点后，[layout-tutte](#)将其他节点布置在该节点所有相连节点的质心处。支撑点集根据用户定义的半径采用圆形布局，然后其他节点逐渐收敛到应该在的位置。（这意味着要运行几次后才能稳定）。



[layout-spring](#) 和 [layout-magspring](#) 很相似，对许多种类的网络很有用。缺点是相对较慢，因为要循环多次才能收敛。在这两种布局里，链就像弹簧，将所连接的两个节点向一起拉，而节点是互斥的。在磁性弹簧（magnetic spring）里，还有磁场在你选择的罗盘方向拉动节点。这些力的强度由原语中的输入参数决定，这些值都在 0-1 之间，记住即使很小的改变也可能影响网络形状。弹簧还有长度（用瓦片数为单位），然而因为力量并不恰好结束在节点之间的距离上。磁性弹簧布局还有一个布尔输入参数 `bidirectional?`，它指明弹簧是否在两个方向产生推力，如果是的话，网络分布就更均匀。



代码例子 [Code Examples](#): Network Example, Network Import Example, Giant Component, Small Worlds, Preferential Attachment

并发请求（Ask-Concurrent）

在以前的NetLogo里，[ask](#)默认是并发的。而在NetLogo 4.0，[ask](#)是串行的，即每次一个主体运行所请求的命令。

下面描述[ask-concurrent](#)的行为，这些行为与老的[ask](#)一样。

[ask-concurrent](#)通过轮转（turn-taking）机制模拟并发。首先是第一个主体，然后是第二

个，依次进行，直到所请求的每个主体都轮转一遍，然后返回到第一个主体。直到所有主体执行完所有命令。

如果主体执行的行为影响世界状态，它的“次序”(turn)结束。例如创建海龟，改变全局变量、海龟变量、瓦片变量或链变量等。（设置局部变量没关系）

命令[forward \(fd\)](#) 和 [back \(bk\)](#)受到特殊对待。当用在[ask-concurrent](#)中时，这些命令分成多轮执行。在每一轮海龟只移动1步。例如`fd 20`等价于`repeat 20 [fd 1]`，每个`fd`后海龟的轮次就结束。如果距离不是整数，则最后的小数部分当作一个轮次。例如`fd 20.3`等价于`repeat 20 [fd 1] fd 0.3`。

不管距离是多少，[jump](#)命令总是只用1轮

要理解[ask](#) 和 [ask-concurrent](#) 的区别，考虑下面的两个命令：

```
ask turtles [ fd 5 ]
```

```
ask-concurrent turtles [ fd 5 ]
```

用[ask](#)，第一个海龟前进5步⁴，然后第二个海龟前进5步，…。

用[ask-concurrent](#)，所有海龟前进1步，然后在前进1步，…。因此该命令等价于：

```
repeat 5 [ ask turtles [ fd 1 ] ]
```

代码实例Code Example: Ask-Concurrent Example 显示了[ask](#) 和 [ask-concurrent](#)的区别。

[ask-concurrent](#) 的行为并不总能简单的由[ask](#)产生。下面的命令：

```
ask-concurrent turtles [ fd random 10 ]
```

为了使用[ask](#)得到相同的行为，必须写成：

```
turtles-own [steps]
```

```
ask turtles [ set steps random 10 ]
```

```
while [any? turtles with [steps > 0]] [
```

```
  ask turtles with [steps > 0] [
```

```
    fd 1
```

```
    set steps steps - 1
```

```
  ]
```

```
]
```

要延长一个主体的“轮次”，使用[without-interruption](#)命令。（有些命令包含这个命令，如[create-turtles](#)和[hatch](#) 隐含了这条命令）。

注意[ask-concurrent](#)的行为是完全确定性的。给定相同的代码和相同的初始状态，总是发生相同的事情。（前提是使用相同的NetLogo版本，指定相同的随机数种子）

一般我们建议你的模型不要依赖于[ask-concurrent](#)的工作细节。我们不保证它的语意在将来的版本里仍然如此。

⁴ 译者注：原文为10步，此处有误。

捆绑 (Tie)

捆绑将两个海龟连在一起，当一个海龟运动时另一个的位置和方向也受到影响。捆绑是链的一个特性，因此要创建捆绑关系，两个海龟之间必须有链存在。

当链的[tie-mode](#)设为“fixed”或“free”，节点1([end1](#))和节点2([end2](#))就捆绑在一起了。如果链是有向的，[end1](#)指根主体(root agent)，[end2](#)指叶主体(leaf agent)。当[end1](#)移动时(使用[fd](#), [jump](#), [setxy](#)等)，[end2](#)也移动相同的距离和方向。然而当[end2](#)移动时，不影响[end1](#)。

如果是无向链则是相互捆绑关系，即只要一个海龟移动，另一个就移动。每个节点都可以被看作根或叶，取决于哪个移动。根海龟总是那个发起移动的海龟。

当根海龟左转或右转时，叶海龟围绕根海龟旋转相同的角度，就像二者之间有刚性连接。如果[tie-mode](#)设为“fixed”，叶海龟的方向也改变相同的量，如果[tie-mode](#)设为“free”，叶海龟的方向不变。

链的[tie-mode](#)可以用[tie](#)命令设为“fixed”，或用[untie](#)命令设为“none”(即海龟不再连接)，要设为“free”，使用set tie-mode “free”。

代码实例 Code Example: Tie System Example

多个源文件 (Multiple source files)

关键词[includes](#)使得可以在一个模型里使用多个源文件。

该关键词前面是两个下划线，表明这是实验性的，以后可能会修改。

当你打开使用了[includes](#)的模型，或将它加到模型的首部然后点击Check按钮，工具条里就会出现includes菜单。在includes菜单里选择模型包含的文件。

当你打开include文件时，它出现在附加的例程页里。细节参见界面指南[Interface Guide](#)。

任何可以出现在例程页中的东西都可以放在外部源文件(.nls)中，例如[breed](#), [turtles-own](#), [patches-own](#), [breeds-own](#)，例程定义等。然而注意这些声明共享同一个名字空间。即如果在例程页声明了一个全局变量my-global，你就不能在其他文件里再次声明，my-global可以在所有include文件里访问。如果在include文件里声明了my-global，也一样。

语法 (Syntax)

这一部分包括了许多读者不熟悉的术语。

关键词 (Keywords)

NetLogo语言仅有的关键词是[globals](#), [breed](#), [turtles-own](#), [patches-own](#), [to](#),

[to-report](#), 和 [end](#), 以及 [extensions](#) 和实验性的 [includes](#). (内置原语不能覆盖不能重定义, 实际上也是一种关键词)

标识符 (Identifiers)

所有原语、全局变量和海龟变量名、例程名共享单一的全局性的大小写不敏感名字空间。局部名 (let 变量与例程输入变量) 与全局变量不会相互覆盖。标志符由字母、数字, 以及 ASCII 字符:

[. ?=!*<>:#+/%\$^`&-]

组成。

目前不允许非 ASCII 字母做标识符。(我们意识到这对国际用户造成麻烦, 计划在未来解决它)

那些两个下划线开始的原语名表示它们是实验性的, 将来的版本里可能会改变或去掉。问号开始的标识符是保留的。

范围 (Scope)

NetLogo 是范围化的。局部变量 (包括例程的输入) 在它所声明的命令块里可访问。但不能被这些命令所调用的例程访问。

注释 (Comments)

分号表示注释的开始, 注释到行尾结束。没有多行注释语法。

结构 (Structure)

程序可以包括一些可选的声明 ([globals](#), [breed](#), [turtles-own](#), [patches-own](#), [<BREED>-own](#)), 顺序任意。然后是 0 个或多个例程定义。可以用 [breed](#) 声明多个种类, 而其他声明只能出现 1 次。

每个例程定义由 [to](#) 或 [to-report](#) 开始, 然后是例程名, 以及可选的方括号中的输入变量列表。每个例程定义由 [end](#) 结束。中间是 0 个或多个命令。

命令和报告器 (Commands and reporters)

命令有 0 个或多个输入, 输入是报告器, 报告器也可有 0 个或多个输入。不用标点分隔或终止命令或输入。标识符必须由空格或圆括号、方括号分开。(例如 a+b 是一个单一标识符, 而 a(b[c]d)e 包括 5 个标识符)

所有命令都是前缀式 (prefix)。所有用户定义的报告器是前缀式。多数原语报告器是前缀式, 但有些 (算术操作, 布尔操作和一些主体集合操作) 是中缀式 (infix)。

所有命令和报告器，不管是原语还是用户定义的，默认采用固定数量的输入。（这就是为什么没有标点分隔或终止命令或输入，而语言却可以被解析的原因）。一些原语有可变数量的输入，这时采用括号，例如(list 1 2 3)。（因为[list](#)默认 2 个输入）。括号也用来改变操作符默认优先序，例如(1 + 2) * 3。

有时原语的一个输入是一个命令块（方括号里 0 个或多个命令），或一个报告器块（方括号中一个报告器表达式）。用户定义的命令和报告器不能使用命令或报告器块做输入。

操作符优先序，从高到低，如下：

- [with](#), [at-points](#), [in-radius](#), [in-cone](#)
- (all other primitives and user-defined procedures)
- ^
- *, /, mod
- +, -
- <, >, <=, >=
- =, !=
- [and](#), [or](#), [xor](#)

与其他 Logo 的比较 (Compared to other Logos)

Logo 是个松散的语言家族，没有一致接受的 Logo 标准定义。我们相信 NetLogo 与其他 Logo 有足够的相同部分，可以配得上 Logo 之名。当然 NetLogo 与其他 Logo 有一些不同之处。最重要的区别如下：

表面区别 (Surface differences)

- 数学运算优先级不同。中缀数学操作（像+,*等）比有名报告器优先级低。例如在许多Logo里，`sin x + 1` 被解释为`sin (x + 1)`。相反NetLogo像多数其他语言那样解释，解释为`(sin x) + 1`。
- [and](#) 和 [or](#) 报告器是特殊形式，不是一般函数，进行“短路”计算，即如果必要的话仅评估第 2 项输入。
- 例程只能在例程页定义，不能在命令中心交互输入。
- 报告器例程必须用[to-report](#) 定义。返回值的命令是[report](#)。
- 定义例程时，例程输入必须用[]，例如`to square [x]`.
- 变量名不使用标点。

最后三项区别由下面的例程定义演示：

most Logos

```
to square :x  to-report square [x]
  output :x * :x report x * x
  end
```

NetLogo

深层区别 (Deeper differences)

- NetLogo 是固定范围，而不是动态范围。
- NetLogo没有“word”数据类型 (Lisp叫做 “symbols”)。最终可能会加上，但不太常用。我们有strings类型，有些Logo使用“word”的地方，我们使用strings。例如在Logo里可以写 [see spot run] (a list of words)，但在NetLogo你必须写“see spot run”串)或 [“see” “spot” “run”] (串列表)
- NetLogo’s 的[run](#) 可操作strings，，但不能操作列表，不允许重新定义例程。
- 如[if](#)和[while](#)的控制结构是特殊形式，你不能自己定义它们的形式，也就不同定义自己的控制结构 (NetLogo’s 的[run](#)在这没用)
- 像多数Logo一样，不能以函数为值。多数Logo提供了相似但一般性交叉的功能，允许以列表形式传递和操作代码片段。NetLogo这方面能力受限。一些内建的UCBLogo-风格的模板实现相似目的，如 sort-by [length ?1 < length ?2] string-list。在某些情况下使用[run](#) and [runresult](#)可以，但不像其他多数Logo那样，它们只对串而不是列表操作。

当然 NetLogo 还包括其他多数 Logo 没有的功能，其中最重要的就是主体和主体集合。

迁移指南（Transition Guide）

许多早期 NetLogo 版本创建的模型在 4.0 里能正常工作。然而，有些模型需要改动。如果你的旧模型不能正常工作，用户手册的本部分能帮你。

你需要了解的问题与模型多旧有关，模型越旧需要了解的问题越多。

本部分没有列出 NetLogo4.0 的所有变化，只是讨论了那些对用户可能造成问题的部分。在[更新历史](#)部分列出了所有的变动。

- [从NetLogo 3.1 迁移](#)
- [从NetLogo 3.0 迁移](#)

从 NetLogo 3.1 迁移

海龟编号（Who numbering）

在NetLogo4.0 之前，死亡海龟的编号（存储在海龟变量[who](#)）可以重新指派给新生的海龟。在 4.0，海龟编号不会重用，除非通过[clear-all](#) 或 [clear-turtles](#) 命令将编号重设为 0。这一变化会对一些旧模型造成问题。

海龟创建：随机与"有序"（ordered）

NetLogo 4.0 提供了两个观察者命令 [create-turtles](#) ([crt](#)) 和 [create-ordered-turtles](#) ([cro](#))，用来创建海龟。

crt 创建的新海龟有随机颜色和随机整数方向。cro 按顺序分配颜色，按顺序等间隔分配方向，第一个海龟的方向是正北（角度为 0）。

在 4.0 之前 crt 命令的行为与现在的 cro 一样。如果旧模型依赖“有序”行为，需要将代码中的 crt 改为 cro。

在旧模型常用 crt 包含额外的命令实现海龟方向随机化，例如 `rt random 360` 或 `set heading random 360`。当在 crt 中使用时，这些命令不再需要。

增加字符串和列表

在 4.0 之前 [+](#) (加号) 操作符可用于拼接字符串和连接列表，在当前版本，+ 只能用于数值。要拼接字符串使用[word](#)原语，要连接列表使用[sentence](#)原语。这些语言变化是为了增加使用+代码的运行速度。

旧代码：

```
print "There are " + count turtles + " turtles."
```

新代码：

```
print (word "There are " count turtles " turtles.")
```

同样，如果需要拼接列表，使用 SENTENCE。

当转换旧模型时，这些变动不会自动处理，需要用户手工改变代码。

我们知道对习惯旧语法的用户来说这个变动太笨拙。我们的目的是效率和一致性。当不是同时处理几个数据类型时，我们可以实现更高效率的数值相加运算。因为加是常见操作，因此 NetLogo 的总体速度提高了。

-at 原语

观察者不再使用 [patch-at](#), [turtles-at](#), [BREEDS-at](#)。相反使用 [patch](#), [turtles-on](#), [BREEDS-on](#)。注意瓦片对输入进行舍入（以前只接受整数输入）

链

NetLogo 3.1 支持使用链来连接海龟，用于创建网络、图和几何图形。链本身是海龟。

在 NetLogo 4.0，链是观察者、海龟、瓦片之外的第四种独立的主体类型。与链有关的原语不再是实验性的，现在是语言的一部分。

使用旧的、实验性的基于海龟的链原语需要改变为使用链主体。区别不是特别大，但需要手工更新。

链的文档在编程指南的 [链](#) 部分，链原语见 NetLogo 词典的有关条目。模型库的 Networks 部分有一些例子模型，还有一些基于链的代码实例。

首先如果只用一种类型 (type) 的链，你需要移除任何名为“links”的种类，然后根本不使用种类。如果使用多种类型的链，参见 [undirected-link-breeds](#) 和 [directed-link-breeds](#)。包括单词“links”(如 [_create-links-with](#), 等)的命令和报告器自动转换为不带下划线([create-links-with](#))的新形式，然而使用其他种类名(如“edges”)的原语不转换，你需要手工移去下划线。除非你的链种类名是“links”，否则需要将链的种类名改为“links”。

命令 [remove-link\(s\)-with/from/to](#) 已不存在了，相反应将所请求的链 [die](#)。

例如：

```
ask turtle 0 [ _remove-links-with link-neighbors ]
```

变为

```
ask turtle 0 [ ask my-links [ die ] ]
```

几个布局命令的输入略有不同，前两项一般是海龟主体集合和需要进行布局的链主体集合。细节见词典 [layout-spring](#), [layout-magspring](#) [layout-radial](#) [layout-tutte](#)。

你需要重新安排海龟变量的声明，因为以前链是海龟。任何用于链的变量应转移到 [links-own](#) 块。

由于链不是海龟，它们不再具有海龟的内置变量（尽管某些链变量的名字与海龟变量一样，如[color](#) 和 [label](#)）。如果以前使用链海龟的位置，现在需要计算链的中点，在非回绕的世界里这很简单）

```
to-report link-xcor
  report mean [xcor] of both-ends
end

to-report link-ycor
  report mean [ycor] of both-ends
end
```

在回绕世界稍微麻烦一点，但也简单

```
to-report link-xcor
  let other-guy end2
  let x 0
  ask end1
  [
    hatch 1
    [
      face other-guy
      fd [distance other-guy] of myself / 2
      set x xcor
      die
    ]
  ]
  report x
end
```

计算ycor与此相似。

如果你使用了链海龟的大小和方向，则现在使用[link-length](#)和[link-heading](#)报告器。

新 "of" 语法

我们用单一的[of](#) 结构（没有连字符）代替了三个不同的语言结构：-of（有连字符），value-from, values-from。

旧	新
color-of turtle 0	[color] of turtle 0
value-from turtle 0 [size * size]	[size * size] of turtle 0
mean values-from turtles [size]	mean [size] of turtles

对单个主体使用 of，返回单一值。对主体集合使用 of，返回值的列表（随机顺序，因为主

体集合总是随机顺序)。

注意当在新版本中打开旧模型时，`-of`, `value-from`, `values-from`自动转换为“of”。但某些嵌套使用这些结构的代码太复杂，不能自动转换，需要手工转换。

串行ask

现在的[ask](#) 命令是串行的而不是并行的。换句话说，每次一个主体运行。直到一个主体运行完所请求的命令块之后，下一个主体才开始运行。

注意老的[ask](#) 也不是真正并行的，我们使用了轮转机制让主体交错执行，来模拟并行执行，该机制见NetLogo常见问题及解答。

我们做出这一改变的原因是，有些用户因为模拟并行而写出不符合原意的代码，而没有从并行中得到好处。如果模型出现未期望的行为，一般可以通过在正确的地方添加[without-interruption](#)来解决，但用户很难了解是否以及在哪需要这个命令。

在NetLogo 4.0, [without-interruption](#)不再是必须的，除非模型使用[ask-concurrent](#)（或者，海龟或瓦片永久性按钮包含依赖于模拟并行的代码）。多数模型中的[without-interruption](#)可以移除。

以前“ask”使用的模拟并行仍然可以通过以下三种方式使用：

- 使用 [ask-concurrent](#)原语而不是 [ask](#)，得到旧的模拟并行方式
- 在命令中心直接向海龟、瓦片或链发出的命令隐含是ask-concurrent
- 海龟、瓦片或链永久性按钮隐含是ask-concurrent

注意不管在哪，ask总是串行的。

模型库使用并行的模型很少，然而 Termites 是一个优秀的使用并行的例子，它使用海龟永久性按钮。

滴答计数器 (Tick counter)

NetLogo 现在有一个内置的滴答计数器，用来表示仿真时间的流逝。

使用[tick](#)命令推进该计数器，如果需要读取它的值，则有报告器[ticks](#)。[clear-all](#) 重设计数器，[reset-ticks](#) 也是如此。

多数模型中滴答计数器是整数值，但如果要使用更小的时间增量，可以用[tick-advance](#)命令推进任何正的量，可以有分数部分。模型库中使用[tick-advance](#)的是Vector Fields 和 GasLab 模型。

滴答计算器的值显示在界面页顶部的工具条上。（可以使用工具条的 Settings... 按钮隐藏滴答计数器，或将“ticks”改为任何其他单词）

视图更新模式

过去NetLogo总是试图每秒更新视图 20 次，现在称之为“连续”视图更新。它最大的问题

是通常希望每个滴答视图进行更新，而不是在滴答之间更新，因此在按钮上有一个勾选框，（默认）在每个按钮循环强制进行显示更新。这确保了更新发生在滴答处，但没有去除中间更新。必须使用[no-display](#)和[display](#) 关掉更新。

我们仍然支持连续更新，它是启动NetLogo后的默认模式。但模型库中多数模型现在使用基于滴答的更新。基于滴答的更新模式下，视图仅在滴答计数器推进时更新。（[display](#)可用于强制附加更新，见下面）

我们认为基于滴答的更新的优点如下：

1. 不同计算机或不同的运行都有一致的、可预测的视图更新行为
2. 中间的更新会让用户糊涂，因为他们看到了不应看到的事情，可能误导用户
3. 提高速度。更新视图占用时间，如果每个滴答一次更新就够的话，强制他们每个滴答只更新一次，模型运行的更快
4. 不需像 NetLogo 3.1 那样，对每个按钮都有"force view update" 勾选框，我们只需一个用于整个模型的选择项
5. 使用速度滑动条让模型减速只需在滴答之间插入暂停。因此在基于滴答的更新模式下，`setup` 按钮不再受速度滑动条的影响，这是旧的滑动条恼人之处。（使用连续更新的模型仍然有此困扰）

如上所述，模型库中的多数模型现在使用基于滴答的更新。

即使正常时设为基于滴答更新的模型，调试时临时设为连续更新也有用。因为能看到在滴答中发生的事情，而不是仅看到一个滴答的结果，这样能帮助解决问题。

如果将模型转换到基于滴答的更新，你还要在代码中增加[tick](#) 命令，否则视图不更新。（注意当按钮弹起或在命令中心输入的命令完成时，视图仍会更新，因此视图不会永远不变）

模型如何使用滴答和基于滴答的更新

下面是在 NetLogo 4.0 将模型转换为使用滴答和基于滴答更新的步骤：

1. 在界面页的工具条右部的"update view:"，将设置从"continuously" 改成"on ticks"
2. 在`go`例程结束部分或接近结束的部分增加[tick](#)命令。模型库中的模型将[tick](#)放在主体移动之后，绘图命令之前。这是因为绘图命令可能会包含像`plotxy ticks ...`这样的东西，我们希望使用滴答计数器的新值。多数模型在绘图命令中不涉及滴答计数器，但不管怎样，为了一致和避免错误，我们建议总是将[tick](#)放在绘图命令之前。

有些模型还要做些改变：

1. 如果模型已经有了全局变量"ticks" 或 "clock" 或"time"，去掉它们。相反使用[tick](#)命令和[ticks](#) 报告器。（如果模型使用小数增量，使用[tick-advance](#)而不是[tick](#)）。如果为这些变量设置了监视器，去掉它，因为工具条上已经有滴答计数器了。
2. `clear-all`将滴答计数器重设为 0。如果在`setup`例程中没有使用`clear-all`，你需要增加[reset-ticks](#)将滴答计数器设为 0
3. 如果在`go`中间使用[no-display](#) 和 [display](#)防止视图更新，去掉它们。

4. 如果模型在时钟不推进时更新视图（例如Party, Dice Stalagmite, 使用动画布局的网络模型，使用鼠标交互按钮的模型），使用[display](#)命令强迫附加更新，因此用户可以看到进行的事情。

速度滑动条

以前的 NetLogo 版本有一个速度滑动条，可以让模型运行的慢一些，这样能看到模型的运行过程。

在 NetLogo4.0，滑动条也可用来加快模型运行，它通过让视图更新的更少实现这一点。更新视图要花时间，因此更新的更少，模型运行的更快。

滑块默认的位置在中间，当在中间时，滑动条是“正常速度”。

当滑块从中心移开时，模型逐渐加快或变慢。

在很高的速度时，视图更新变得很少，可能几秒才更新一次。因为更新太少，看起来就像模型运行的很慢。但看着滴答计数器或绘图等，会看到模型实际运行很快。如果更新太少令人不安，那就不要将滑块拖得太远。

如果使用基于滴答的更新，减慢模型不会引起额外的更新，相反，NetLogo 只是简单的在滴答之间加入暂停。

当使用连续更新时，减慢模型意味着视图更新间隔更接近。如果将滑块拖过左边一半的位置，模型运行太慢能看到一个海龟移动一步！这是 NetLogo4.0 新有的，在以前版本里不管模型运行多慢，也不会看到一次一步，而是在一个 ask 中的所有主体同时移动。

数值

NetLogo 内部不再区分整数和浮点数。例如：

旧：

```
observer> print 3
3
observer> print 3.0
3.0
observer> print 1 + 2
3
observer> print 1.5 + 1.5
3.0
observer> print 3 = 3.0
true
```

(最后一行显示尽管区分整数 3 和浮点数 3.0，但这两个数仍然认为相等)

新：

```
observer> print 3
3
observer> print 3.0
```

```

3
observer> print 1 + 2
3
observer> print 1.5 + 1.5
3
observer> print 3 = 3.0
true

```

我们希望只有极少的模型受到这一变动的负面影响。

这一变动的好处是 NetLogo 支持的整数范围更大。旧的范围是 -2,147,483,648 到 2,147,483,647 (大约 $+/- 2 \text{ billion}$)，新的范围是 $+/- 9,007,199,254,740,992$ (大约 $+/- 9 \text{ quadrillion}$)。

创建主体集合

NetLogo 3.1 (和一些早期版本) 包括原语 turtles-from 和 patches-from，偶尔用来创建主体集合。在NetLogo 4.0，这些原语被新原语 [turtle-set](#) 和 [patch-set](#) 替换，它们更灵活、强大。[\(link-set\)](#) 也存在。在NetLogo词典中查阅这些条目。使用老的 turtles-from 和 patches-from 的模型需要手工替换为新原语。

RGB 颜色

在NetLogo 3.1，RGB和HSB颜色可以使用rgb 和 hsb近似为NetLogo颜色。这些改名为 [approximate-rgb](#) 和 [approximate-hsb](#)，现在的输入参数范围是 0–255，不是 0–1。

现在 NetLogo 提供完整的 RGB 色系，因此这些原语不是必须的。可以使用三个元素的列表设置任何颜色变量，获得精确的颜色，列表项范围 0–255，。细节见编程指南的颜色部分。

捆绑 (Tie)

旧版本中 `_tie` 是实验性的。在NetLogo 4.0 中，链有一个 [tie-mode](#) 变量，可设为 “none”，“free”，或 “fixed”。在4.0 [tie](#) 现在是链独有的原语，意味着要将 turtle 1 捆绑到 turtle 0 需要写：

```
ask turtle 0 [ create-link-to turtle 1 [ tie ] ]
```

细节见编程指南的捆绑 ([Tie](#)) 部分。

HubNet 客户

HubNet 活动的客户界面不再存储在单独的模型文件中。要从老的模型导入客户端，选择 File → Import → Import HubNet Client，当请求时从界面页导入。不再需要外部的客户

模型，以及当设置客户界面时不再需要像这样指向它：

```
hubnet-set-client-interface "COMPUTER" [ "my-client.nlogo" ]
```

变为：

```
hubnet-set-client-interface "COMPUTER" []
```

列表性能

列表的内部实现变化了，列表的一些性能特征改变了，当前实现的细节见编程指南。注意[fput](#)比[lput](#)快得多，简单的改为[fput](#)就能提高性能。如果性能仍然是个问题，考虑使用数组和表扩展。

从 NetLogo 3.0 迁移

主体集合

如果模型行为怪异或不正确，也许是因为从NetLogo 3.1 开始，主体集合是随机顺序。以前版本中主体集合总是固定顺序。如果代码依赖固定顺序，则不会正常运行。如何修改模型能以随机主体集合工作，取决于模型细节。有时使用[sort](#) 或 [sort-by](#) 将主体集合（随机顺序）转换为主体列表（固定顺序），会有所帮助。见编程指南列表部分的“主体列表”。

回绕

如果看到海龟片段在视图边界回绕，是因为 NetLogo 3.0 允许在视图关闭这样的回绕，而不影响模型的行为。从 NetLogo 3.1 起，如果不让视图回绕，则必须使用新的拓扑功能让世界不回绕。然而这一改变可能还要对模型做出其他改变。在编程指南的拓扑部分有详细讨论，告诉你怎样转换模型利用这一新功能的优势。

随机化海龟坐标

在NetLogo 3.0 或以前版本，许多模型使用 `setxy random world-width random world-height` 随机散布海龟，使用 `random` 或 `random-float`。只有世界回绕，它才能工作。

（为什么？因为当打开回绕时，可以设置海龟坐标超过世界边界，NetLogo 将对海龟进行回绕。但在不回绕的世界，设置 x 或 y 坐标超过世界边界会引起运行错误。在 NetLogo3.1 增加了回绕设置。更多信息见编程指南的拓扑部分）

要修改模型使它不管拓扑设置如何都能工作，使用下面的两个命令之一：

```
setxy random-xcor random-ycor
```

```
setxy random-pxcor random-pycor
```

这两个命令略有不同。第一个将海龟放置在世界的一个随机点，第二个将海龟放置到一个随机瓦片的中心。一种更简洁的将海龟放置到随机瓦片中心的方式是：

```
move-to one-of patches
```

NetLogo 词典 (NetLogo Dictionary)

字母顺序: [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [L](#) [M](#) [N](#) [O](#) [P](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [?](#)

分类: [Turtle](#) - [Patch](#) - [Agentset](#) - [Color](#) - [Control/Logic](#) - [World](#) - [Perspective](#)
[Input/Output](#) - [Files](#) - [List](#) - [String](#) - [Math](#) - [Plotting](#) - [Links](#) - [Movie](#) - [System](#) - [HubNet](#)

特殊: [Variables](#) - [Keywords](#) - [Constants](#)

分类 (Categories)

下面是近似分组。记住海龟相关的原语也可能被瓦片或观察者使用，反之亦然。要知道哪类主体(海龟、瓦片、链，观察者)实际使用哪个原语，请查找词典中的相应词条。

海龟相关 (Turtle-related)

[back](#) ([bk](#)) [<breeds>-at](#) [<breeds>-here](#) [<breeds>-on](#) [can-move?](#) [clear-turtles](#) ([ct](#))
[create-<breeds>](#) [create-ordered-<breeds>](#) [create-ordered-turtles](#) ([cro](#))
[create-turtles](#) ([crt](#)) [die](#) [distance](#) [distancexy](#) [downhill](#) [downhill4](#) [dx](#) [dy](#) [face](#) [facexy](#)
[forward](#) ([fd](#)) [hatch](#) [hatch-<breeds>](#) [hide-turtle](#) ([ht](#)) [home](#) [inspect](#) [is-<breed>?](#)
[is-turtle?](#) [jump](#) [left](#) ([lt](#)) [move-to](#) [myself](#) [nobody](#) [no-turtles](#) [of](#) [other](#) [patch-ahead](#)
[patch-at](#) [patch-at-heading-and-distance](#) [patch-here](#) [patch-left-and-ahead](#)
[patch-right-and-ahead](#) [pen-down](#) ([pd](#)) [pen-erase](#) ([pe](#)) [pen-up](#) ([pu](#)) [random-xcor](#)
[random-ycor](#) [right](#) ([rt](#)) [self](#) [set-default-shape](#) [set-line-thickness](#) [setxy](#) [shapes](#)
[show-turtle](#) ([st](#)) [sprout](#) [sprout-<breeds>](#) [stamp](#) [stamp-erase](#) [subject](#)
[subtract-headings](#) [tie](#) [towards](#) [towardsxy](#) [turtle](#) [turtle-set](#) [turtles](#) [turtles-at](#)
[turtles-here](#) [turtles-on](#) [turtles-own](#) [untie](#) [uphill](#) [uphill4](#)

瓦片相关 (Patch-related)

[clear-patches](#) ([cp](#)) [diffuse](#) [diffuse4](#) [distance](#) [distancexy](#) [import-pcolors](#)
[import-pcolors-rgb](#) [inspect](#) [is-patch?](#) [myself](#) [neighbors](#) [neighbors4](#) [nobody](#) [no-patches](#)
[of](#) [other](#) [patch](#) [patch-at](#) [patch-ahead](#) [patch-at-heading-and-distance](#) [patch-here](#)
[patch-left-and-ahead](#) [patch-right-and-ahead](#) [patch-set](#) [patches](#) [patches-own](#)
[random-pxcor](#) [random-pycor](#) [self](#) [sprout](#) [sprout-<breeds>](#) [subject](#)

主体集合 (Agentset)

[all?](#) [any?](#) [ask](#) [ask-concurrent](#) [at-points](#) [breeds](#) [at](#) [breeds](#) [here](#) [breeds](#) [on](#) [count](#) [in-cone](#) [in-radius](#) [is-agent?](#) [is-agentset?](#) [is-patch-set?](#) [is-turtle-set?](#) [link-heading](#) [link-length](#) [link-set](#) [link-shapes](#) [max-n-of](#) [max-one-of](#) [min-n-of](#) [min-one-of](#) [n-of](#) [neighbors](#) [neighbors4](#) [no-patches](#) [no-turtles](#) [of](#) [one-of](#) [other](#) [patch-set](#) [patches](#) [sort](#) [sort-by](#) [turtle-set](#) [turtles](#) [with](#) [with-max](#) [with-min](#) [turtles-at](#) [turtles-here](#) [turtles-on](#)

颜色 (Color)

[approximate-hsb](#) [approximate-rgb](#) [base-colors](#) [color](#) [extract-hsb](#) [extract-rgb](#) [hsb](#) [import-pcolors](#) [import-pcolors-rgb](#) [pcolor](#) [rgb](#) [scale-color](#) [shade-of?](#) [wrap-color](#)

控制流和逻辑 (Control flow and logic)

[and](#) [ask](#) [ask-concurrent](#) [carefully](#) [end](#) [error-message](#) [foreach](#) [if](#) [ifelse](#) [ifelse-value](#) [let](#) [loop](#) [map](#) [not](#) [or](#) [repeat](#) [report](#) [run](#) [runresult](#) ; (semicolon) [set](#) [stop](#) [startup](#) [to](#) [to-report](#) [wait](#) [while](#) [with-local-randomness](#) [without-interruption](#) [xor](#)

世界 (World)

[clear-all](#) [\(ca\)](#) [clear-drawing](#) [\(cd\)](#) [clear-patches](#) [\(cp\)](#) [clear-turtles](#) [\(ct\)](#) [display](#) [import-drawing](#) [import-pcolors](#) [import-pcolors-rgb](#) [no-display](#) [max-pxcor](#) [max-pycor](#) [min-pxcor](#) [min-pycor](#) [reset-ticks](#) [tick](#) [tick-advance](#) [ticks](#) [world-width](#) [world-height](#)

视角 (Perspective)

[follow](#) [follow-me](#) [reset-perspective](#) [\(rp\)](#) [ride](#) [ride-me](#) [subject](#) [watch](#) [watch-me](#)

HubNet

[hubnet-broadcast](#) [hubnet-broadcast-view](#) [hubnet-enter-message?](#) [hubnet-exit-message?](#) [hubnet-fetch-message](#) [hubnet-message](#) [hubnet-message-source](#) [hubnet-message-tag](#) [hubnet-message-waiting?](#) [hubnet-reset](#) [hubnet-send](#) [hubnet-send-view](#) [hubnet-set-client-interface](#)

输入/输出 (Input/output)

[beep](#) [clear-output](#) [date-and-time](#) [export-view](#) [export-interface](#) [export-output](#) [export-plot](#) [export-all-plots](#) [export-world](#) [import-drawing](#) [import-pcolors](#) [import-pcolors-rgb](#) [import-world](#) [mouse-down?](#) [mouse-inside?](#) [mouse-patch](#) [mouse-xcor](#) [mouse-ycor](#) [output-print](#) [output-show](#) [output-type](#) [output-write](#) [print](#) [read-from-string](#) [reset-timer](#) [set-current-directory](#) [show](#) [timer](#) [type](#) [user-directory](#) [user-file](#) [user-new-file](#) [user-input](#) [user-message](#) [user-one-of](#) [user-yes-or-no?](#) [write](#)

文件 (File)

[file-at-end?](#) [file-close](#) [file-close-all](#) [file-delete](#) [file-exists?](#) [file-flush](#) [file-open](#) [file-print](#) [file-read](#) [file-read-characters](#) [file-read-line](#) [file-show](#) [file-type](#) [file-write](#) [user-directory](#) [user-file](#) [user-new-file](#)

列表 (List)

[but-first](#) [but-last](#) [empty?](#) [filter](#) [first](#) [foreach](#) [fput](#) [histogram](#) [is-list?](#) [item](#) [last](#) [length](#) [list](#) [lput](#) [map](#) [member?](#) [modes](#) [n-of](#) [n-values](#) [of](#) [position](#) [one-of](#) [reduce](#) [remove](#) [remove-duplicates](#) [remove-item](#) [replace-item](#) [reverse](#) [sentence](#) [shuffle](#) [sort](#) [sort-by](#) [sublist](#)

字符串 (String)

[Operators](#) ([<](#), [>](#), [=](#), [!=](#), [<=](#), [>=](#)) [but-first](#) [but-last](#) [empty?](#) [first](#) [is-string?](#) [item](#) [last](#) [length](#) [member?](#) [position](#) [remove](#) [remove-item](#) [read-from-string](#) [replace-item](#) [reverse](#) [substring](#) [word](#)

数学 (Mathematical)

[Arithmetic Operators](#) ([+](#), [*](#), [-](#), [/](#), [^](#), <, >, =, !=, <=, >=) [abs](#) [acos](#) [asin](#) [atan](#) [ceiling](#) [cos](#) [e](#) [exp](#) [floor](#) [int](#) [ln](#) [log](#) [max](#) [mean](#) [median](#) [min](#) [mod](#) [modes](#) [new-seed](#) [pi](#) [precision](#) [random](#) [random-exponential](#) [random-float](#) [random-gamma](#) [random-normal](#) [random-poisson](#) [random-seed](#) [remainder](#) [round](#) [sin](#) [sqrt](#) [standard-deviation](#) [subtract-headings](#) [sum](#) [tan](#) [variance](#)

绘图 (Plotting)

[autoplot?](#) [auto-plot-off](#) [auto-plot-on](#) [clear-all-plots](#) [clear-plot](#)
[create-temporary-plot-pen](#) [export-plot](#) [export-all-plots](#) [histogram](#) [plot](#) [plot-name](#)
[plot-pen-exists?](#) [plot-pen-down](#) [plot-pen-reset](#) [plot-pen-up](#) [plot-x-max](#) [plot-x-min](#)
[plot-y-max](#) [plot-y-min](#) [plotxy](#) [set-current-plot](#) [set-current-plot-pen](#)
[set-histogram-num-bars](#) [set-plot-pen-color](#) [set-plot-pen-interval](#)
[set-plot-pen-mode](#) [set-plot-x-range](#) [set-plot-y-range](#)

链 (Links)

[both-ends](#) [clear-links](#) [create-<breed>-from](#) [create-<breeds>-from](#) [create-<breed>-to](#)
[create-<breeds>-to](#) [create-<breed>-with](#) [create-<breeds>-with](#) [create-link-from](#)
[create-links-from](#) [create-link-to](#) [create-links-to](#) [create-link-with](#)
[create-links-with](#) [in-<breed>-neighbor?](#) [in-<breed>-neighbors](#) [in-<breed>-from](#)
[in-link-neighbor?](#) [in-link-neighbors](#) [in-link-from](#) [is-directed-link?](#) [is-link?](#)
[is-undirected-link?](#) [layout-circle](#) [layout-magspring](#) [layout-radial](#) [layout-spring](#)
[layout-tutte](#) [<breed>-neighbor?](#) [<breed>-neighbors](#) [<breed>-with](#) [link-heading](#)
[link-length](#) [link](#) [links](#) [links-own](#) [<link-breeds>-own](#) [link-neighbors](#)
[link-with](#) [my-<breeds>](#) [my-in-<breeds>](#) [my-in-links](#) [my-links](#) [my-out-<breeds>](#)
[my-out-links](#) [no-links](#) [other-end](#) [out-<breed>-neighbor?](#) [out-<breed>-neighbors](#)
[out-<breed>-to](#) [out-link-neighbor?](#) [out-link-neighbors](#) [out-link-to](#) [show-link](#) [tie](#)
[untie](#)

电影 (Movie)

[movie-cancel](#) [movie-close](#) [movie-grab-view](#) [movie-grab-interface](#)
[movie-set-frame-rate](#) [movie-start](#) [movie-status](#)

系统 (System)

[netlogo-applet?](#) [netlogo-version](#)

内建变量 (Built-In Variables)

海龟 (Turtles)

[breed](#) [color](#) [heading](#) [hidden?](#) [label](#) [label-color](#) [pen-mode](#) [pen-size](#) [shape](#) [size](#) [who](#) [xcor](#)
[ycor](#)

瓦片 (Patches)

[pcolor](#) [plabel](#) [plabel-color](#) [pxcor](#) [pycor](#)

链 (Links)

[breed](#) [color](#) [end1](#) [end2](#) [hidden?](#) [label](#) [label-color](#) [shape](#) [thickness](#) [tie-mode](#)

其他 (Other)

[?](#)

关键词 (Keywords)

[breed](#) [directed-link-breed](#) [end](#) [extensions](#) [globals](#) [includes](#) [patches-own](#) [to](#) [to-report](#) [turtles-own](#) [undirected-link-breed](#)

常量 (Constants)

数学常量 (Mathematical Constants)

e = 2.718281828459045

pi = 3.141592653589793

布尔常量 (Boolean Constants)

false

true

颜色常量 (Color Constants)

black = 0

gray = 5

white = 9.9

red = 15

orange = 25

brown = 35

yellow = 45

```
green = 55
lime = 65
turquoise = 75
cyan = 85
sky = 95
blue = 105
violet = 115
magenta = 125
pink = 135
```

细节参见编程指南的颜色部分（[Colors](#)）。

A

abs

abs *number*

返回 *number* 的绝对值。

```
show abs -7
=> 7
show abs 5
=> 5
```

acos

acos *number*

返回给定数的反余弦值。输入数值必须在-1 到 1 之间。结果是度数，范围在 0 到 180 之间。

all?

all? *agentset* [*reporter*]

如果主体集合（agentset）中的所有主体对给定的报告器（reporter）都返回 true，则返回 true。否则返回 false。

给定的报告器必须对每个主体都返回布尔值（true 或 false），否则发生错误。

```
if all? turtles [color = red]
```

```
[ show "every turtle is red!" ]
```

另外见 [any?](#).

and

condition1 and *condition2*

如果 *condition1* 与 *condition2* 为 true，则返回 true。

注意如果 *condition1* 为 false，则不再检查 *condition2*（因为对结果没影响）

```
if (pxcor > 0) and (pycor > 0)
  [ set pcolor blue ] ;; the upper-right quadrant of
    ;; patches turn blue
```

any?

any? agentset

如果给定主体集合非空，返回 true，否则返回 false。

等价于“*count agentset > 0*”，但效率更高（也更易读）。

```
if any? turtles with [color = red]
  [ show "at least one turtle is red!" ]
```

注意 nobody 不是一个主体集合。只能在希望得到单个主体而不是整个主体集合的地方得到 nobody。（You only get nobody back in situations where you were expecting a single agent, not a whole agentset）。如果将 nobody 做为 any? 的输入，会导致错误。

另见 [all](#), [nobody](#).

approximate-hsb

approximate-hsb hue saturation brightness

返回 0–140（不包括 140）之间的一个数，表示 NetLogo 颜色空间中 HSB 色谱的某个颜色

三个输入值必须在 0–255 范围内。

返回的颜色可能只是一个近似，因为 NetLogo 颜色空间没有包括所有可能颜色。（只包括某些离散的色调（hue），对每个色调，饱和度或亮度可以变化，但二者不能同时变化 — 至少二者之一总是 255）

```
show approximate-hsb 0 0 0
=> 0 ;; (black)
show approximate-hsb 127.5 255 255
=> 85.2 ;; (cyan)
```

另见 [extract-hsb](#), [approximate-rgb](#), [extract-rgb](#).

approximate-rgb

approximate-rgb *red green blue*

返回 0-140（不包括 140）之间的一个数，表示 NetLogo 的 RGB 颜色空间中的某个颜色
三个输入值必须在 0-255 范围内。

返回的颜色可能只是一个近似，因为 NetLogo 颜色空间没有包括所有可能颜色。（参见 [approximate-hsb](#) 的说明，很难用 RGB 术语表达）

```
show approximate-rgb 0 0 0
=> 0 ;; black
show approximate-rgb 0 255 255
=> 85.2 ;; cyan
```

另见 [extract-rgb](#), [approximate-hsb](#), 和 [extract-hsb](#).

算子, Arithmetic Operators (+, *, -, /, ^, <, >, =, !=, <=, >=)

这些运算符都是中缀运算符（infix operators），都有两个输入参数。（把它置于两个输入参数之间，就像标准数学那样）。NetLogo 实现正确的中缀运算顺序。

运算符含义：+ 加, * 乘, - 减, / 除, ^ 幂, < 小于, > 大于, = 等于, != 不等于, <= 小于等于, >= 大于等于。

注意减法运算符一般采用两个输入，但是如果用括号括起来可以只有一个输入。例如对 x 取负，写为(- x)，要有括号。

可以对字符串运用比较操作。

比较操作可以用于主体。对海龟比较 who number，对瓦片从上到下、从左到右比较，因此瓦片 0 10 小于 0 9，瓦片 9 0 小于 10 0。链通过端点排序，如果是捆绑的话，则通过种类比较。因此，link 0 9 在 link 1 10 之前，因为 end1 较小，link 0 8 小于 link 0 9。如果链有多个种类，则具有相同端点的无种类链在有种类链之前，而有种类的链根据在例程页中声明顺序进行排序。

可以对主体集合测试相等或不等。如果两个主体集合是同一类型（海龟或瓦片）且包含相同的主体，则相等。

如果不太确定 NetLogo 怎样解释你的代码，应加上括号。

```
show 5 * 6 + 6 / 3
=> 32
show 5 * (6 + 6) / 3
=> 20
```

asin

asin *number*

返回给定数值的反正弦值。输入参数必须在 -1 到 1 之间。结果是度数，在 -90 到 90 范围。

ask

```
ask agentset [commands]
ask agent [commands]
```

指定的主体 (agent) 或主体集合 (agentset) 执行给定的命令。

```
ask turtles [ fd 1 ]
;; all turtles move forward one step
ask patches [ set pcolor red ]
;; all patches turn red
ask turtle 4 [ rt 90 ]
;; only the turtle with id 4 turns right
```

注意：只有观察者可以请求所有海龟或所有瓦片。这可以防止你因不小心而让所有海龟请求所有海龟，或所有瓦片请求所有瓦片，当你不太清楚哪个主体运行你的代码时，容易犯这个错误。

注意：仅有那些在 ask 开始时刻的主体集合中的主体运行这些命令。

ask-concurrent

`ask-concurrent agentset [commands]`

给定主体集合中的主体以轮换（turn-taking）机制运行给定命令，模拟并发执行。参见编程指南的[Ask-Concurrent](#)部分。

注意：仅有那些在 ask 开始时刻的主体集合中的主体运行这些命令。

另见 [without-interruption](#).

at-points

`agentset at-points [[x1 y1] [x2 y2] ...]`

返回给定主体集合的一个子集，该子集只包括那些离调用主体给定距离处的瓦片上的主体。距离以列表形式给出，列表的每个元素有两项，即 x 和 y 偏移。

如果调用主体是观察者，则距离是指到原点的距离，换句话说，就是瓦片的绝对坐标。

如果调用主体是海龟，距离是指到该海龟的精确距离，而不是到该海龟所在瓦片中心的距离

```
ask turtles at-points [[2 4] [1 2] [10 15]]
  [ fd 1 ] ;; only the turtles on the patches at the
  ;; distances (2,4), (1,2) and (10,15),
  ;; relative to the caller, move
```

atan

`atan x y`

返回 x 除以 y 的反正切值，是度数值（0–360）。

当 y 为 0 时，如果 x 为正，返回 90；如果 x 为负，返回 270；如果 x 为 0，出错。

注意这个版本的 atan 设计用来与 NetLogo 世界的几何一致。在 NetLogo 世界，0 方向是上，90 是右，顺时针旋转。（在一般几何里，0 是右，90 是上，逆时针旋转，atan 需要一致的实现）

```
show atan 1 -1
=> 135
show atan -1 1
```

=> 315

autoplot?

autoplot?

如果当前绘图的 auto-plotting 打开，则返回 true，否则返回 false.

auto-plot-off

auto-plot-on

auto-plot-off

auto-plot-on

这对命令控制当前绘图的自动绘图 (auto-plotting) 功能。当画笔超出当前边界时，自动绘图功能自动更新 x 和 y 轴。当要显示所有绘制点而不管图形范围时，会用到命令。

B

back

bk

back *number*



海龟后退 *number* 步。（如果 *number* 为负，则海龟前进）

海龟使用这个原语每个时间步最大移动 1 个单位。因此 bk 0.5 和 bk 1 都使用一个时间步，但 bk 3 用三个时间步。

如果因为当前拓扑的原因不能后退 *number* 步，则海龟尽可能移动整数步，停下。

另见 [forward](#), [jump](#), [can-move?](#).

base-colors

base-colors

返回由 14 个 NetLogo 基本色调构成的列表。

```

print base-colors
=> [5 15 25 35 45 55 65 75 85 95 105 115 125 135]
ask turtles [ set color one-of base-colors ]
;; each turtle turns a random base color
ask turtles [ set color one-of remove gray base-colors ]
;; each turtle turns a random base color except for gray

```

beep

beep

发出蜂鸣声。注意蜂鸣声很迅速，几个非常靠近的蜂鸣命令听起来就像一条声音。

例如：

```

beep          ;; emits one beep
repeat 3 [ beep ]      ;; emits 3 beeps at once,
                      ;; so you only hear one sound
repeat 3 [ beep wait 0.1 ] ;; produces 3 beeps in succession,
                           ;; separated by 1/10th of a second

```

both-ends



返回由该链的两个端点组成的主体集合。

```

crt 2
ask turtle 0 [ create-link-with turtle 1 ]
ask link 0 1 [
  ask both-ends [ set color red ] ;; turtles 0 and 1 both turn red
]

```

breed

breed



这是一个内建的海龟和链变量。它保存着所有与该海龟（链）同一种类的海龟（链）构成的主体集合。（对于没有特定种类的海龟或链，这个变量就是所有海龟的主体集合[turtles](#) 或所有链的主体集合[links](#)）。可以设置这个变量改变海龟或链的种类。

另见 [breed](#), [directed-link-breed](#), [undirected-link-breed](#)

例子：

```
breed [cats cat]
breed [dogs dog]
;; turtle code:
if breed = cats [ show "meow!" ]
set breed dogs
show "woof!"
directed-link-breed [ roads road ]
;; link code
if breed = roads [ set color gray ]
```

breed

`breed [<breeds> <breed>]`

这个关键词只能在例程页的首部使用，就像 `globals`, `turtles-own` 和 `patches-own` 一样。它定义一个种类。第一个输入参数定义该种类主体集合的名字，第二个参数定义该种类单个主体的名字。

给定种类的任何海龟：

- 都是由种类名命名的主体集合的一部分
- 拥有主体集合定义的种类内建变量

主体集合最常与 `ask` 一起，给特定种类的海龟发出命令。

```
breed [mice mouse]
breed [frogs frog]
to setup
  clear-all
  create-mice 50
  ask mice [ set color white ]
  create-frogs 50
  ask frogs [ set color green ]
  show [breed] of one-of mice    ;;= prints mice
  show [breed] of one-of frogs  ;;= prints frogs
```

```

end

show mouse 1
;; prints (mouse 1)
show frog 51
;; prints (frog 51)
show turtle 51
;; prints (frog 51)

```

另见 [globals](#), [patches-own](#), [turtles-own](#), [⟨breeds⟩-own](#), [create-⟨breeds⟩](#), [⟨breeds⟩-at](#), [⟨breeds⟩-here](#).

but-first

bf

but-last

bl

but-first *list*
but-first *string*
but-last *list*
but-last *string*

与列表一起使用时, but-first 返回 *list* 中除第一项外的所有项。but-last 返回 *list* 中除最后一项外的所有项。

与字符串一起使用时, but-first 和 but-last 返回忽略了原始字符串第一个或最后一个字符的字符串。

```

;; mylist is [2 4 6 5 8 12]
set mylist but-first mylist
;; mylist is now [4 6 5 8 12]
set mylist but-last mylist
;; mylist is now [4 6 5 8]
show but-first "string"
;; prints "tring"
show but-last "string"
;; prints "strin"

```

C**can-move?**`can-move? distance`

如果调用主体能够沿所面向的方向前进 *distance* 而不与拓扑冲突，则返回 true，否则返回 false。

它等价于：

`patch-ahead distance != nobody`**carefully**`carefully [commands1] [commands2]`

运行 *commands1*，如果出错，NetLogo 不停下来报警，而是抑制错误运行 *commands2*。

在 *commands2* 可以使用 `error-message` 报告器，找出在 *commands1* 中被抑制的错误信息。见 [error-message](#)。

注意：两组命令都不受中断的运行（与使用 `without-interruption` 一样）。

```
carefully [ show 1 / 1 ] [ print error-message ]
=> 1
carefully [ show 1 / 0 ] [ print error-message ]
=> division by zero
```

ceiling`ceiling number`

返回大于等于 *number* 的最小整数。

```
show ceiling 4.5
=> 5
show ceiling -4.5
=> -4
```

clear-all

ca

`clear-all`



将所有全局变量清 0，调用 `reset-ticks`, `clear-turtles`, `clear-patches`, `clear-drawing`, `clear-all-plots`, and `clear-output`

clear-all-plots

`clear-all-plots`



清除模型中所有绘图（plot）。更多信息参见[clear-plot](#)。

clear-drawing

cd

`clear-drawing`



清除海龟画出的所有线和图案。

clear-links

`clear-links`



删除所有链。

另见 [die](#).

clear-output

`clear-output`



如果模型有输出区域，则清除所有文本。否则什么都不做。

clear-patches

cp

clear-patches



将所有瓦片变量重设为默认初始值，包括将颜色设为黑。

clear-plot

clear-plot

对当前绘图, 重设所有画笔, 删除所有临时画笔, 将绘图设为默认值 (x, y 的范围等) , 将所有永久画笔设为默认值。绘图和永久画笔的默认值在绘图编辑对话框里设置。如果删除所有临时画笔后没有画笔了, 也就是说没有永久性画笔, 则使用下面的设置自动生成一个默认画笔:

- Pen: down
- Color: black
- Mode: 0 (line mode)
- Name: "default"
- Interval: 1

另见[clear-all-plots](#).

clear-turtles

ct

clear-turtles



删除所有海龟.

也重设 who number, 因此下一个创建的海龟号为 0。

另见 [die](#).

color**color**

这是一个内建海龟或链变量，保存海龟或链的颜色，设置该变量则海龟或链改变颜色。颜色可用NetLogo颜色（一个数字），或RGB颜色（有3个数的列表）。细节见编程指南的颜色部分（[Colors section](#)）。

另见 [pcolor](#).

cos**cos** *number*

返回给定角的余弦值。角的单位是度。

```
show cos 180
=> -1
```

count**count** *agentset*

返回给定主体集合的主体数量。

```
show count turtles
;; prints the total number of turtles
show count patches with [pcolor = red]
;; prints the total number of red patches
```

create-ordered-turtles**cro****create-ordered-<breeds>**

```
create-ordered-turtles number
create-ordered-turtles number [ commands ]
create-ordered<breeds> number
```

`create-ordered<breeds> number [commands]`



创建 *number* 个新海龟。新海龟位于 (0, 0) 处, 用 14 个主要颜色分别设定颜色, 方向在 0–360 均匀设置。

如果采用 `create-ordered-<breeds>` 形式, 则创建属于该种类的新海龟。

如果提供了 *commands*, 新海龟立即运行这些命令。使用这些命令可以给新海龟不同的颜色、方向或任何其他东西。(新海龟一次全部创建, 然后以随机顺序每次选择一个海龟运行命令)

```
cro 100 [ fd 10 ] ;; makes an evenly spaced circle
```

注意: 当命令运行时, 其他主体不能运行任何代码 (就像使用了 `without-interrupt` 命令)。这就确保了如果正在使用 `ask-concurrent`, 则新海龟在完全初始化之前不能与任何其他海龟交互。

create-<breed>-to

create-<breeds>-to

create-<breed>-from

create-<breeds>-from

create-<breed>-with

create-<breeds>-with

create-link-to

create-links-to

create-link-from

create-links-from

create-link-with

create-links-with

`create-<breed>-to turtle`

`create-<breed>-to turtle [commands]`

`create-<breed>-from turtle`

```

create-<breed>-from turtle [ commands ]
create-<breed>-with turtle
create-<breed>-with turtle [ commands ]
create-<breeds>-to turtleset
create-<breeds>-to turtleset [ commands ]
create-<breeds>-from turtleset
create-<breeds>-from turtleset [ commands ]
create-<breeds>-with turtleset
create-<breeds>-with turtleset [ commands ]
create-link-to turtle
create-link-to turtle [ commands ]
create-link-from turtle
create-link-from turtle [ commands ]
create-link-with turtle
create-link-with turtle [ commands ]
create-links-to turtleset
create-links-to turtleset [ commands ]
create-links-from turtleset
create-links-from turtleset [ commands ]
create-links-with turtleset
create-links-with turtleset [ commands ]

```



用来在海龟之间创建有种类或无种类的链。

`create-link-with` 在调用者和 *agent* 之间创建一个无向链。

`create-link-to` 创建一个从调用者到 *agent* 的一个有向链。

`create-link-from` 创建一个从 *agent* 到调用者的一个有向链。.

当使用复数形式的种类名时，需要给一个主体集合，在调用者和主体集合中的所有主体之间创建链。

可选的命令块是每个新构建的链要运行的命令。（所有链一次全部创建，然后以随机顺序每次运行一个）。

节点不能自连。在两个节点之间不能有多条同种类的无向链，在两个节点之间也不能有多个同向的同种类的有向链。

如果试图创建一条已存在的链（同种类），什么也不发生。如果试图创建一个海龟自连的链，则出现运行错误。

```

to setup
  crt 5

```

```

;; turtle 1 creates links with all other turtles
;; the link between the turtle and itself is ignored
ask turtle 0 [ create-links-with other turtles ]
show count links ;; shows 4
;; this does nothing since the link already exists
ask turtle 0 [ create-link-with turtle 1 ]
show count links ;; shows 4 since the previous link already existed
ask turtle 2 [ create-link-with turtle 1 ]
show count links ;; shows 5
end
directed-link-breed [red-links red-link]
undirected-link-breed [blue-links blue-link]

to setup
crt 5
;; create links in both directions between turtle 0
;; and all other turtles
ask turtle 0 [ create-red-links-to turtles ]
ask turtle 0 [ create-red-links-from turtles ]
show count links ;; shows 8
;; now create undirected links between turtle 0 and other turtles
ask turtle 0 [ create-blue-links-with turtles ]
show count links ;; shows 12
end

```

create-turtles**crt****create-<breeds>**

```

create-turtles number
create-turtles number [ commands ]
create-<breeds> number
create-<breeds> number [ commands ]

```



创建 *number* 个新海龟。新海龟的方向是随机整数，颜色从 14 个主色中随机选取。

如果使用 **create-<breeds>** 形式，新海龟就是给定种类的成员。

如果提供了 *commands*, 新海龟立即执行这些命令。使用这些命令可以给新海龟不同的颜色、方向或任何其他东西。(新海龟一次全部创建, 然后以随机顺序每次选择一个海龟运行命令)

```
crt 100 [ fd 10 ]      ;;= makes a randomly spaced circle
breed [canaries canary]
breed [snakes snake]
to setup
  clear-all
  create-canaries 50 [ set color yellow ]
  create-snakes 50 [ set color green ]
end
```

注意: 当命令运行时, 其他主体不能运行任何代码(就像使用了without-interrupt命令)。这就确保了如果正在使用ask-concurrent, 则新海龟在完全初始化之前不能与任何其他海龟交互。

另见 [hatch](#), [sprout](#).

create-temporary-plot-pen

create-temporary-plot-pen *string*

使用给定名称为当前绘图创建一个临时画笔, 该画笔设为当前画笔。

很少有模型使用这一原语, 因为当调用 clear-plot 或 clear-all-plots 后, 所有临时画笔都消失。创建画笔的正常方式是在绘图的编辑对话框中创建永久画笔。

如果当前绘图中存在同名的临时画笔, 则不再创建新画笔, 将已存在的画笔设为当前画笔。如果有一个同名的永久画笔, 则出现运行错误。

新的临时画笔的初始设置如下:

- Pen: down
- Color: black
- Mode: 0 (line mode)
- Interval: 1

见: [clear-plot](#), [clear-all-plots](#) 和 [set-current-plot-pen](#)。

D**date-and-time****date-and-time**

返回包含当前日期和时间的字符串。格式如下，所有的域（field）都是固定长度的，因此某个域总是出现在字符串的相同位置。时钟可能的精度是毫秒。（在不同的系统上得到的精度可能会不同，这取决于底层的 Java 虚拟机）

```
show date-and-time
=> "01:19:36.685 PM 19-Sep-2002"
```

die**die**

海龟死亡。

```
if xcor > 20 [ die ]
;; all turtles with xcor greater than 20 die
```

另见：[ct](#)

diffuse**diffuse patch-variable number**

告诉每个瓦片将瓦片变量 *patch-variable* 的(*number* * 100)% 均等的分配到它的 8 个相邻瓦片上去。*Number* 在 0-1 之间。不管拓扑如何，整个世界的 *patch-variable* 之和守恒。（如果一个瓦片的邻元少于 8 个，每个邻元仍然得到 1/8 的份额，剩余的该瓦片自己保留）

注意这是一个观察者命令，尽管你希望这是一个瓦片命令。（原因是该命令同时对所有瓦片起作用—而瓦片命令只能作用到单个瓦片）

```
diffuse chemical 0.5
;; each patch diffuses 50% of its variable
;; chemical to its neighboring 8 patches. Thus,
;; each patch gets 1/8 of 50% of the chemical
```

```
;; from each neighboring patch.)
```

diffuse4

diffuse4 *patch-variable number*



与 diffuse 类似，区别是对四个相邻瓦片进行扩散（北、南、东、西），而不包括对角邻元。

```
diffuse4 chemical 0.5
;; each patch diffuses 50% of its variable
;; chemical to its neighboring 4 patches. Thus,
;; each patch gets 1/4 of 50% of the chemical
;; from each neighboring patch.)
```

directed-link-breed

directed-link-breed [*<link-breeds>* *<link-breed>*]

与 globals 和 breeds 关键词一样，这个关键词只能在例程页的首部使用，位于所有例程定义之前。它定义一个有向链种类。某个种类的链必须都是有向的或都是无向的。第一个参数定义该种类链的主体集合名，第二个参数定义单个成员名。创建有向链时使用 [create-link\(s\)-to](#) 和 [create-link\(s\)-from](#)，而不使用 [create-link\(s\)-with](#)。

任何一个属于指定链种类的链：

- 是链种类名所定义的主体集合的一部分
- 有设定到该主体集合的内建变量 breed
- 由关键词决定是有向还是无向

最常见的用法是主体集合和 ask 命令组合在一起，向只属于特定种类的链发出命令。

```
directed-link-breed [streets street]
directed-link-breed [highways highway]
to setup
  clear-all
  crt 2
  ;; create a link from turtle 0 to turtle 1
  ask turtle 0 [ create-street-to turtle 1 ]
  ;; create a link from turtle 1 to turtle 0
  ask turtle 0 [ create-highway-from turtle 1 ]
end
```

```
ask turtle 0 [ show one-of in-links ]
;; prints (street 0 1)
ask turtle 0 [ show one-of out-links ]
;; prints (highway 1 0)
```

另见 [breed](#), [undirected-link-breed](#)

display

display

引起视图立刻更新。（例外：如果用户使用速度滑动条快进模型，更新可能被跳过）

另外，撤销 no-display 命令的效果，如果视图更新被 no-display 挂起，则恢复。

```
no-display
ask turtles [ jump 10 set color blue set size 5 ]
display
;; turtles move, change color, and grow, with none of
;; their intermediate states visible to the user, only
;; their final state
```

即使没有使用 no-display 命令，“display”也有用。因为正常情况下 NetLogo 会逃过一些视图更新，由于总的更新变少，模型会运行的快一些。该命令强迫视图更新，因此世界发生的一切都能被用户看到。

```
ask turtles [ set color red ]
display
ask turtles [ set color blue]
;; turtles turn red, then blue; use of "display" forces
;; red turtles to appear briefly
```

注意 display 和 no-display 与视图控制条上冻结视图的开关无关。

另见 [no-display](#).

distance

distance *agent*



返回本主体与给定的海龟或瓦片的距离。

离或到一个瓦片的距离是根据瓦片中心得到的。如果世界拓扑允许回绕，并且回绕距离更短，则海龟和瓦片使用回绕距离（围绕世界边缘）。

```
ask turtles [ show max-one-of turtles [distance myself] ]
;; each turtle prints the turtle farthest from itself
```

distancexy

distancexy *xcor* *ycor*



返回从本主体到给定点 (*xcor*, *ycor*) 的距离。

离开瓦片的距离根据瓦片中心计算。如果世界拓扑允许回绕，并且回绕距离更短，则海龟和瓦片使用回绕距离（围绕世界边缘）。

```
if (distancexy 0 0) > 10
[ set color green ]
;; all turtles more than 10 units from
;; the center of the world turn green.
```

downhill

downhill4

downhill *patch-variable*
downhill4 *patch-variable*



海龟移动到 *patch-variable* 最小的那个相邻瓦片上。如果没有哪个相邻瓦片变量比当前瓦片小，则保持不动。如果有几个瓦片有相同的最小值，则随机选择一个。非数值型值忽略。

Downhill 考虑 8 个相邻瓦片，而 downhill4 考虑四个相邻瓦片。

与下面的代码等价(假设变量是数值型)：

```
move-to patch-here ;; go to patch center
let p min-one-of neighbors [patch-variable] ;; or neighbors4
if [patch-variable] of p < patch-variable [
  face p
  move-to p
```

]

注意海龟总是停在瓦片中心，方向角是 45 (downhill) 或 90 (downhill4) 的倍数。

另见[uphill](#), [uphill4](#).

dx**dy**

dx

dy



返回海龟沿当前方向前进一步时的 x 增量或 y 增量(海龟的 xcor 或 ycor 的改变量)。

注意: dx 就是海龟方向角的正弦值, dy 就是余弦值。(这可能与你想的相反, 原因在于 NetLogo 的 0 方向是北, 90 是东, 与一般几何中的角度定义相反)

注意: 在NetLogo早期版本, 这些命令很常用, 现在新的patch-ahead原语更合适。

E**empty?****empty? list****empty? string**

如果给定的列表或字符串为空, 返回 true, 否则返回 false。

注意: 空列表写成[], 空字符串写成""。

end

end

用来结束一个例程。见[to](#) 和[to-report](#)。

end1**end1**

这是一个内置链变量。它指明链的第一个端点（海龟）。对有向链是指源端点，对无向链是指具有小的 who number 的端点。你不能设置 end1。

```
crt 2
ask turtle 0
[ create-link-to turtle 1 ]
ask links
[ show end1 ] ;; shows turtle 0
```

end2**end2**

这是一个内置链变量。它指明链的第二个端点（海龟）。对有向链是指目的端点，对无向链是指具有大的 who number 的端点。你不能设置 end2。

```
crt 2
ask turtle 1
[ create-link-with turtle 0 ]
ask links
[ show end2 ] ;; shows turtle 1
```

error-message**error-message**

返回被 carefully 抑制的描述错误信息的字符串。

该报告器只能用在 carefully 命令的第二部分。

另见 [carefully](#).

every**every** *number* [*commands*]

仅当在同一上下文中（in this context）距离上次运行给定命令超过 *number* 秒，才运行该组命令。否则命令被跳过。

`Every` 本身不能使命令不断重复运行。如果你想要重复运行，则需要将它放到一个循环里，或者放到永久性按钮里。`Every` 只限制命令运行的频率。

上面所谓的同一上下文（in this context）指相同的ask（或按钮按下或命令中心输入的命令）。因此如果写成`ask turtles [every 0.5 [...]]`就没什么意义，因为当ask完成时海龟就丢弃对“every”的计时器。正确的用法如下：

```
every 0.5 [ ask turtles [ fd 1 ] ]
;; twice a second the turtles will move forward 1
every 2 [ set index index + 1 ]
;; every 2 seconds index is incremented
```

另见 [wait.](#)

exp

`exp number`

返回 e 的 *number* 次幂值。

注意：与 $e^{\wedge} number$ 相同。

export-view

export-interface

export-output

export-plot

export-all-plots

export-world

```
export-view filename
export-interface filename
export-output filename
export-plot plotname filename
export-all-plots filename
export-world filename
```

`export-view` 将当前视图的当前内容输出到由 *filename* 命名的外部文件。文件存为 PNG (Portable Network Graphics) 格式，因此推荐文件后缀为 “.png”。

`export-interface` 相似，只是输出的是整个界面页。

`export-output` 将模型的输出区域内容输出到由 *filename* 命名的外部文件。（如果模型没有独立的输出区域，则输出命令中心的输出区域）

`export-plot` 将绘图 *plotname* 中所有画笔绘制的所有点的 x 和 y 值输出到由 *filename* 命名的外部文件。如果画笔是条型 (bar) 模式 (mode 0)，并且点的 y 值大于 0，则输出条形的左上角点，如果 y 值小于 0，则输出左下角点。

`export-all-plots` 将当前模型的所有绘图输出到由 *filename* 命名的外部文件。每个绘图的格式与 `export-plot` 输出时相同。

`export-world` 将所有变量的值，包括内建变量和用户定义的变量，所有观察者、海龟、瓦片变量，画图 (drawing)，输出区域 (如果有的话)，绘图 (plot) 内容，随机数发生器的状态，输出到由 *filename* 命名的外部文件。（可以被 [import-world](#) 读回 NetLogo）。

`export-world` 不保存打开文件的状态。

`export-plot`, `export-all-plots` 和 `export-world` 用无格式文本 (plain-text)、逗号分隔值 “comma-separated values” (.csv) 格式保存文件。许多常用的电子表格、数据库程序以及文本编辑器都能读取 CSV 文件。

如果文件已存在，则覆盖。

如果输出文件不在模型目录，则需给定输出文件的全路径。（使用 “/” 作为文件夹分隔符）。

注意这些功能可以直接在 NetLogo 的 File 菜单中使用。

```
export-world "fire.csv"
;; exports the state of the model to the file fire.csv
;; located in the NetLogo folder
export-plot "Temperature" "c:/My Documents/plot.csv"
;; exports the plot named
;; "Temperature" to the file plot.csv located in
;; the C:\My Documents folder
export-all-plots "c:/My Documents/plots.csv"
;; exports all plots to the file plots.csv
;; located in the C:\My Documents folder
```

extensions

`extensions [name ...]`

允许模型使用给定扩展库中的原语。详情见[Extensions guide](#)。

extract-hsb

extract-hsb *color*

返回指定 NetLogo 颜色的 HSB 值列表，指定颜色在 0–140(不包括)之间。返回的列表有三项，每项分别是色调、饱和度、亮度，范围在 0–255.

```
show extract-hsb red
=> [2.198 206.372 215]
show extract-hsb cyan
=> [127.5 145.714 196]
```

另见 [approximate-hsb](#), [approximate-rgb](#), [extract-rgb](#).

extract-rgb

extract-rgb *color*

返回指定 NetLogo 颜色的 RGB 值列表，指定颜色在 0–140(不包括)之间。返回的列表有三项，每项分别是红、绿、蓝，范围在 0–255.

```
show extract-rgb red
=> [215 50 41]
show extract-rgb cyan
=> [84 196 196]
```

另见 [approximate-rgb](#), [approximate-hsb](#), [extract-hsb](#).

F

face

face *agent*



设置调用者的方向为朝向 *agent*

如果世界拓扑允许回绕，并且回绕距离较短，face 将使用回绕路径。

如果调用者和 agent 恰好位于相同点，则调用者的方向不变。

facexy

facexy *number number*



设置调用者的方向为朝向点 (x, y)。

如果世界拓扑允许回绕，并且回绕距离较短，facexy 将使用回绕路径。

如果调用者恰好位于点 (x, y)，则调用者的方向不变。

file-at-end?

file-at-end?

如果已达到文件(已用[file-open](#) 打开)尾，返回true，否则返回false。

```
file-open "my-file.txt"  
print file-at-end?  
=> false ;; Can still read in more characters  
print file-read-line  
=> This is the last line in file  
print file-at-end?  
=> true ;; We reached the end of the file
```

另见 [file-open](#), [file-close-all](#).

file-close

file-close

关闭已用[file-open](#) 打开的文件。

注意该命令和 file-close-all 是重新指到已打开文件的起始位置和切换文件模式的仅有方法。

如果没有文件打开，则没有任何作用。

另见 [file-close-all](#), [file-open](#).

file-close-all

file-close-all

关闭所有以前由[file-open](#) 打开的文件（如果有的话）

另见 [file-close](#), [file-open](#)

file-delete

file-delete *string*

删除 *string* 指明的文件。

String 必须是一个用户有写权限的已有文件。另外文件不能打开。在删除之前用[file-close](#) 关闭打开的文件。

注意 *string* 可以是文件名，也可指绝对路径。如果是文件名，表示在当前目录。可以使用[set-current-directory](#) 改变当前目录，默认是模型目录。

file-exists?

file-exists? *string*

如果文件 *string* 存在则返回 true，否则返回 false。

注意 *string* 可以是文件名，也可指绝对路径。如果是文件名，表示在当前目录。可以使用[set-current-directory](#) 改变当前目录，默认是模型目录。

file-flush

file-flush

强迫文件刷新到磁盘。当使用写操作或其他输出命令时，可能不会立即写到磁盘，这样是为了提高文件输出性能。关闭文件能确保所有输出写入到磁盘。

有时需要在不关闭文件的前提下确保数据写入磁盘。例如你使用文件与计算机上的其他程序进行通信，想要其他程序立即看到输出。

file-open

file-open *string*

该命令将*string*解释为路径名并且打开文件。你可以使用 [file-read](#), [file-read-line](#) 和 [file-read-characters](#) 读取文件，也可使用[file-write](#), [file-print](#), [file-type](#), [file-show](#) 写入文件。

注意你可以打开文件进行读或写，但不能既读又写。接下来的文件输入输出命令决定了文件的打开模式。要切换模式需要使用[file-close](#)关闭文件。

另外，如果打开文件进行读的话，文件必须已经存在。

当打开文件进行写操作，所有新数据会追加到原始文件尾。如果没有原始文件，则在原位新建一个空文件（必须对该目录有写权限）。（如果不想追加，而是要替换现有内容，先使用[file-delete](#)删除它，如果不確定它是否存在的话，最好使用[carefully](#)）。

注意*string*可以是文件名，也可是绝对路径。如果是文件名，表示在当前目录。可以使用[set-current-directory](#) 改变当前目录，默认是模型目录。

```
file-open "my-file-in.txt"
print file-read-line
=> First line in file ;; File is in reading mode
file-open "C:\\NetLogo\\my-file-out.txt"
;; assuming Windows machine
file-print "Hello World" ;; File is in writing mode
```

另见 [file-close](#).

file-print

file-print *value*

将 *value* 打印到一个打开的文件中，后面跟一个回车。

在*value*前不打印调用主体，这一点与[file-show](#)不同。

注意这是与[print](#) 等价的文件i/o命令，在使用该命令前要先使用[file-open](#)。

另见 [file-show](#), [file-type](#), and [file-write](#).

file-read

file-read

这个报告器从打开的文件中读取一个常数，就像在命令中心输入它一样，执行它，返回结果。结果可能是数值、列表、字符串、布尔值、或者特殊值 nobody。

常数由空格分隔，file-read 调用时跳过前后空格。

注意字符串需要用引号，在使用[file-write](#) 时包括引号。

还要注意在使用该报告器之前要先使用[file-open](#) 命令，并且文件里还有数据。使用报告器[file-at-end?](#) 测试是否达到文件尾。

```
file-open "my-file.data"
print file-read + 5
;; Next value is the number 1
=> 6
print length file-read
;; Next value is the list [1 2 3 4]
=> 4
```

另见 [file-open](#) 和 [file-write](#).

file-read-characters

file-read-characters *number*

将打开文件中的 *number* 个字符作为一个字符串返回。如果剩余的字符数量不足，则返回所有剩下的字符。

注意它返回包括新行和空格在内的所有字符。

还要注意在使用该命令之前要先使用[file-open](#) 命令，并且文件里还有数据。使用报告器[file-at-end?](#) 测试是否达到文件尾。

```
file-open "my-file.txt"
print file-read-characters 5
;; Current line in file is "Hello World"
=> Hello
```

另见 [file-open](#).

file-read-line

file-read-line

读取文件中的下一行，作为字符串返回。通过回车、文件结束字符或者二者都在一行判定文件是否结束。（It determines the end of the file by a carriage return, an end of file character or both in a row.）

还要注意在使用该命令之前要先使用[file-open](#) 命令，并且文件里还有数据。使用报告器[file-at-end?](#) 测试是否达到文件尾。

```
file-open "my-file.txt"  
print file-read-line  
=> Hello World
```

另见 [file-open](#).

file-show

file-show *value*

将*value*打印到文件中，前面是调用主体，后面是一个回车。（包含调用主体用来帮你跟踪输出的每一行是哪个主体产生的）。与[file-write](#) 类似，字符串要有引号。

注意这是与[show](#) 等价的文件i/o命令。在使用该命令前，要先使用[file-open](#)。

另见 [file-print](#), [file-type](#)和 [file-write](#).

file-type

file-type *value*

将*value*打印到一个打开的文件中，后面不跟回车（与[file-print](#) 和 [file-show](#) 不同）。不用回车使你能在一行打印几个值。

与[file-show](#) 不同，前面没有调用主体。

注意这是与[type](#) 等价的文件i/o 命令，在使用该命令前，要先使用[file-open](#)。

另见 [file-print](#), [file-show](#) 和[file-write](#).

file-write

file-write *value*

将一个值输出到一个打开的文件，可以是数值、字符串、列表、布尔值、或nobody，后面不跟回车（与[file-print](#) 和 [file-show](#)不同）。

与[file-show](#)不同，前面没有调用主体。输出的字符串有引号，前面加一个空格。用这种方式输出，这样[file-read](#) 能解释它。

注意这是与[write](#)等价的文件i/o 命令，在使用该命令前，要先使用[file-open](#) 。

```
file-open "locations.txt"  
ask turtles  
[ file-write xcor file-write ycor ]
```

另见 [file-print](#), [file-show](#) 和[file-type](#) 。

filter

filter [*reporter*] *list*

返回 *list* 中由布尔型 *reporter* 为真的项组成的列表 — 也就是说，满足给定条件的项。

在*reporter*中，用 [?](#) 引用列表*list*的当前项。

```
show filter [? < 3] [1 3 2]  
=> [1 2]  
show filter [first ? != "t"] ["hi" "there" "everyone"]  
=> ["hi" "everyone"]
```

另见 [map](#), [reduce](#), [?](#).

first

first *list*
first *string*

对列表，返回列表的第一（0th）项。

对字符串，返回仅包含原始字符串中第一个字符的字符串。

floor**floor** *number*返回小于等于 *number* 的最大整数。

```
show floor 4.5
=> 4
show floor -4.5
=> -5
```

follow**follow** *turtle*与 *ride* 相似，但在 3 维视图中，观察者的位置是在 *turtle* 的后上方。另见 [follow-me](#), [ride](#), [reset-perspective](#), [watch](#), [subject](#).**follow-me****follow-me**

请求观察者跟随调用主体。

另见 [follow](#).**foreach**

```
foreach list [ commands ]
(foreach list1 ... [ commands ])
```

对单个列表，让列表中的每一项运行 *commands*。在 *commands* 中用 [?](#) 引用列表当前项。

```
foreach [1.1 2.2 2.6] [ show (word ? " -> " round ?) ]
=> 1.1 -> 1
=> 2.2 -> 2
=> 2.6 -> 3
```

对多个列表，对每个列表中的每一组项运行 *commands*。因此第一项运行一次，第二项运行一次，等等。所有列表长度必须相同。在 *commands* 里使用 [?1](#) 到 [?n](#) 引用每个列表的当前项。

下面的几个例子使得含义清楚一点：

```
(foreach [1 2 3] [2 4 6]
  [ show word "the sum is: " (?1 + ?2) ])
=> "the sum is: 3"
=> "the sum is: 6"
=> "the sum is: 9"

(foreach list (turtle 1) (turtle 2) [3 4]
  [ ask ?1 [ fd ?2 ] ])
;; turtle 1 moves forward 3 patches
;; turtle 2 moves forward 4 patches
```

另见 [map](#), [?](#).

forward

fd

forward *number*



海龟前进 *number* 步，每次 1 步。（如果 *number* 为负则后退）

fd 10 等价于 repeat 10 [jump 1]。fd 10.5 等价于 repeat 10 [jump 1] jump 0.5。

如果海龟因当前拓扑的限制不能前进 *number* 步，则前进尽可能大的整数步，然后停下。

另见 [jump](#), [can-move?](#).

fput

fput *item* *list*

将 *item* 加到列表首，返回新列表。

```
;; suppose mylist is [5 7 10]
set mylist fput 2 mylist
;; mylist is now [2 5 7 10]
```

G**globals**

```
globals [var1 ...]
```

这个关键词和 breed, <breeds>-own, patches-own, turtles-own 一样, 只能用在程序首部, 位于任何例程定义之前。它定义新的全局变量。全局变量是“全局”的, 因为能被任何主体访问, 能在模型中的任何地方使用。

一般全局变量用于定义在程序多个部分使用的变量或常量。

H**hatch****hatch-<breeds>**

```
hatch number [ commands ]
hatch-<breeds> number [ commands ]
```



本海龟创建 *number* 个新海龟。每个新海龟与母体相同, 处在同一个位置。然后新海龟运行 *commands*。可以使用 *commands* 给新海龟不同的颜色、方向、位置等任何东西。(新海龟同时创建, 然后以随机顺序每次运行一个)

如果使用 hatch-<breeds> 形式, 则新海龟是给定种类的成员。否则, 新海龟与母体种类相同。

注意: 当这个命令运行时, 其他主体不能运行任何代码(就像使用 without-interruption 命令)。这确保如果使用 ask-concurrent, 在新海龟全部初始化之前, 新海龟不能与任何其他主体交互。

```
hatch 1 [ lt 45 fd 1 ]
;; this turtle creates one new turtle,
;; and the child turns and moves away
hatch-sheep 1 [ set color black ]
;; this turtle creates a new turtle
;; of the sheep breed
```

另见 [create-turtles](#), [sprout](#).

heading

heading



这是一个内置海龟变量，指明海龟面向的方向，该值在[0, 360)。0是北，90是东，等等。
设置这个变量实现海龟转动。

另见 [right](#), [left](#), [dx](#), [dy](#).

例子：

```
set heading 45      ;;= turtle is now facing northeast
set heading heading + 10 ;; same effect as "rt 10"
```

hidden?

hidden?



这是一个内置的海龟或链变量，是一个布尔值（true 或 false），指明海龟或链当前是否隐藏（即不可见）。设置这个变量使海龟或链消失或出现。

另见 [hide-turtle](#), [show-turtle](#), [hide-link](#), [show-link](#)

例子：

```
set hidden? not hidden?
;; if turtle was showing, it hides, and if it was hiding,
;; it reappears
```

hide-link

hide-link



链使自己不可见。

注意：该命令等价于设置链变量“hidden?”为 true 。

另见 [show-link](#).

hide-turtle**ht****hide-turtle**

海龟使自己不可见。

注意：该命令等价于设置海龟变量“hidden?”为 true 。

另见 [show-turtle](#).**histogram****histogram** *list*

将给定列表中的值表达为直方图。

直方图显示列表值的频率分布。柱形的高度表示落到每个区间的数值数量。

在绘制直方图之前，首先以前所有当前画笔画出的点被删除。

列表中的非数值型值忽略。

直方图由当前画笔和当前画笔颜色在当前绘图上画出。使用 set-plot-x-range 控制要画成直方图的数值范围，设置画笔间隔（直接使用 set-plot-pen-interval，或用 set-histogram-num-bars 间接设置）控制要分成多少个区间。

确认如果要用柱形绘制，当前画笔应在 bar 模式（模式 1）。

对直方图，绘图的 X 范围不包括最大 X 值，等于最大 X 值的数落在直方图范围之外。

```
histogram [color] of turtles
;; draws a histogram showing how many turtles there are
;; of each color
```

home**home**

调用海龟移动到原点(0, 0)，等价于setxy 0 0。

hsb

hsb *hue saturation brightness*

返回用 HSB 格式给定颜色的 RGB 列表。Hue, saturation, brightness 是 0–255 范围内的整数。RGB 列表包含处于相同范围的三个整数。

另见[rgb](#)

hubnet-broadcast

hubnet-broadcast *tag-name value*

从 NetLogo 广播 *value*, 在计算器 HubNet 是到变量, 在计算机 HubNet 是到界面元素, 带着客户端的名字 *tag-name*。

详情和教学见 [HubNet Authoring Guide](#).

hubnet-broadcast-view

hubnet-broadcast-view

将 NetLogo 模型 2 维视图的当前状态广播到所有计算机 HubNet 客户。对计算器 HubNet 无效果。

注意：这是一个实验性原语，未来版本中行为可能有变。

详情和教学见 [HubNet Authoring Guide](#).

hubnet-enter-message?

hubnet-enter-message?

如果有一个新的计算机客户进入仿真, 返回true, 否则返回false。[hubnet-message-source](#) 将包含刚登录客户的用户名。

详情和教学见 [HubNet Authoring Guide](#).

hubnet-exit-message?

hubnet-exit-message?

如果有一个计算机客户退出仿真，返回true，否则返回false。[hubnet-message-source](#) 将包含刚退出客户的用户名

详情和教学见 [HubNet Authoring Guide](#).

hubnet-fetch-message

hubnet-fetch-message

如果有客户发来的新数据，则取出下一条数据，这样就可以被[hubnet-message](#),
[hubnet-message-source](#), [hubnet-message-tag](#)访问。如果没有来自客户的新数据则出错。

细节见 [HubNet Authoring Guide](#).

hubnet-message

hubnet-message

返回[hubnet-fetch-message](#)取得的消息。

细节见 [HubNet Authoring Guide](#).

hubnet-message-source

hubnet-message-source

返回由[hubnet-fetch-message](#)所获得消息的发送客户名。

细节见 [HubNet Authoring Guide](#).

hubnet-message-tag

hubnet-message-tag

返回[hubnet-fetch-message](#)取得数据相关联的标志（tag）。对计算器HubNet，返回由[hubnet-set-client-interface](#)设置的一个变量名。对计算机HubNet，返回客户界面上界面元素的一个显示名。

细节见 [HubNet Authoring Guide](#).

hubnet-message-waiting?

hubnet-message-waiting?

查看客户发送的新消息。如果有新消息，返回 true，否则返回 false.

细节见 [HubNet Authoring Guide](#).

hubnet-reset

hubnet-reset

启动HubNet系统。除了[hubnet-set-client-interface](#)外，HubNet必须启动才能使用其他的Hubnet原语。

细节见 [HubNet Authoring Guide](#).

hubnet-send

hubnet-send *string tag-name value*

hubnet-send *list-of-strings tag-name value*

对计算器HubNet，该原语与[hubnet-broadcast](#)完全一样。（将来NetLogo版本计划改变）

对计算机 HubNet，是这样的：

对字符串 *string*，从 NetLogo 发送 *value* 到具有用户名 *string* 的客户上的 tag *tag-name*。

对字符串列表 *list-of-strings*，从 NetLogo 发送 *value* 到所有列表中用户名客户上。

如果发送到不存在的客户，则产生hubnet-exit-message消息。

细节见 [HubNet Authoring Guide](#).

hubnet-send-view

hubnet-send-view *string*

hubnet-send-view *list-of-strings*

对计算器 HubNet 无任何效果。

对计算机 HubNet, 是这样的:

对 *string*, 将 2 维视图当前状态发送到具有用户名 *string* 的客户。

对 *list-of-strings*, 将 2 维视图当前状态发送到 *list-of-strings* 中各用户名的客户

将 2 维视图发送到不存在的客户, 则产生 hubnet-exit-message 消息

注意: 这是一个实验性原语, 未来版本中行为可能有变。

细节见 [HubNet Authoring Guide](#).

hubnet-set-client-interface

hubnet-set-client-interface *client-type* *client-info*

如果 *client-type* 是 "COMPUTER", *client-info* 是对计算机 HubNet 的一个空列表。

`hubnet-set-client-interface "COMPUTER"[]`

未来的 HubNet 会支持其他客户类型。即使是计算机 HubNet, 第 2 个输入项的含义也可能改变。

细节见 [HubNet Authoring Guide](#).

|

if

if *condition* [*commands*]

报告器必须返回一个布尔值 (true 或 false)。

如果 *condition* 为 true, 运行 *commands*。

报告器可能对不同的主体返回不同的值, 因此有些主体会执行 *commands*, 有些则不会。

```
if xcor > 0[ set color blue ]
;; turtles in the right half of the world
;; turn blue
```

另见 [ifelse](#), [ifelse-value](#).

ifelse

ifelse *reporter* [*commands1*] [*commands2*]

报告器必须返回一个布尔值(true 或 false)。

如果 *reporter* 返回 true, 运行 *commands1*; 如果 *reporter* 返回 false, 运行 *commands2*。

报告器可能对不同的主体返回不同的值, 因此有些主体会执行 *commands1*, 有些会执行 *commands2*。

```
ask patches
  [ ifelse pxcor > 0
    [ set pcolor blue ]
    [ set pcolor red ] ]
  ;;= the left half of the world turns red and
  ;;= the right half turns blue
```

另见 [if](#), [ifelse-value](#)。

ifelse-value

ifelse-value *reporter* [*reporter1*] [*reporter2*]

报告器必须返回一个布尔值(true 或 false)。

如果 *reporter* 返回 true, 结果是 *reporter1*的值。

如果 *reporter* 返回 false, 结果是 *reporter2*的值。

当报告器需要条件项, 而不允许使用命令 (如[ifelse](#)) 时, 该原语可以用上。

```
ask patches [
  set pcolor ifelse-value (pxcor > 0) [blue] [red]
]
 ;;= the left half of the world turns red and
 ;;= the right half turns blue
 show n-values 10 [ifelse-value (? < 5) [0] [1]]
=> [0 0 0 0 0 1 1 1 1 1]
 show reduce [ifelse-value (?1 > ?2) [?1] [?2]]
 [1 3 2 5 3 8 3 2 1]
=> 8
```

另见[if](#), [ifelse](#).

import-drawing

import-drawing *filename*



将一个图像文件读到画图层 (drawing)，对图像进行缩放使它的尺寸与世界一致，维持原来的长宽比。图像在画图层的中心。以前的画图不清除。

主体感知不到画图层的存在，因此不能处理由import-drawing读入的图像，也不能与图像交互。如果需要主体感知图像，使用[import-pcolors](#) 或 [import-pcolors-rgb](#)。

支持的图像文件格式有：BMP, JPG, GIF, PNG。如果图像格式支持透明 (alpha)，这些信息也会输入。

import-pcolors

import-pcolors *filename*



读入图像文件，将它缩放到与瓦片网格尺寸一致，维持原来的长宽比，将所得的像素颜色传递给瓦片。图像位于瓦片网格的中心。因为 NetLogo 的颜色空间没有包括所有颜色，所以最后得到的瓦片颜色可能有失真。（参见编程指南的颜色部分）。对有些图像 import-pcolors 可能较慢，特别是当瓦片很多、图像包括很多颜色时。

因为import-pcolors设置了瓦片的pcolor，因此主体可以感知到图像。如果主体需要分析、处理图像或以其他方式与图像交互，该命令很有用。如果只想简单的显示没有颜色失真的背景图像，则使用[import-drawing](#)。

支持的图像文件格式有：BMP, JPG, GIF, PNG。如果图像格式支持透明 (alpha)，则所有的透明像素被忽略。（部分透明像素被作为不透明处理）

import-pcolors-rgb

import-pcolors-rgb *filename*



读入图像文件，将它缩放到与瓦片网格尺寸一致，维持原来的长宽比，将所得的像素颜色传递给瓦片。图像位于瓦片网格的中心。与[import-pcolors](#)不同，该命令能得到准确的原图像颜色。所有瓦片的pcolor是一个RGB列表，而不是（近似的）NetLogo颜色。

支持的图像文件格式有：BMP, JPG, GIF, PNG。如果图像格式支持透明（alpha），则所有的透明像素被忽略。（部分透明像素被作为不透明处理）

import-world

`import-world filename`



从给定名字的外部文件读所有变量到模型之中，包括内置变量和用户定义变量，所有的观察者、海龟、瓦片变量。外部文件的格式应与[export-world](#)产生的格式一致。

注意该原语的功能也可在 NetLogo 的文件菜单直接得到。

在使用 import-world 时，为避免出错，应按顺序执行下面几步：

1. 打开创建了输出文件的模型。
2. 按下 Setup 按钮，让模型处于可运行状态。
3. 输入文件。
4. 再次打开所有模型里使用file-open命令打开的文件
5. 如果需要，按下 Go 按钮让模型在中断处开始继续运行。

如果要输入的文件不在模型目录，则使用全路径名。参见[export-world](#)的例子。

in-cone

`agentset in-cone distance angle`



这个报告器让海龟在它前方形成锥形视场⁵（"cone of vision"）。用两个输入参数定义锥形，即视距（半径）和视角。视角的中心是海龟的当前方向，大小从 0-360.（如果是 360，则等价于圆形视场in-radius）

in-cone 返回一个主体集合，该集合仅包含原集合中落在锥形之内的主体。（可能包含调用者自身）

到瓦片的距离从瓦片中心算起。

```
ask turtles
  [ ask patches in-cone 3 60
    [ set pcolor red ]
    ;; each turtle makes a red "splotch" of patches in a 60 degree
    ;; cone of radius 3 ahead of itself
```

⁵ 译者注：在 2 维视图里，实际是扇形视场。

in-<breed>-neighbor?**in-link-neighbor?**in-<breed>-neighbor? *agent*in-link-neighbor? *turtle*

如果从 *turtle* 到调用者有一条有向链，则返回 true。

```
crt 2
ask turtle 0 [
  create-link-to turtle 1
  show in-link-neighbor? turtle 1 ;; prints false
  show out-link-neighbor? turtle 1 ;; prints true
]
ask turtle 1 [
  show in-link-neighbor? turtle 0 ;; prints true
  show out-link-neighbor? turtle 0 ;; prints false
]
```

in-<breed>-neighbors**in-link-neighbors**

in-<breed>-neighbors

in-link-neighbors



返回一个海龟主体集合，如果存在从该海龟到调用者的有向链，则该海龟进入主体集合。

```
crt 4
ask turtle 0 [ create-links-to other turtles ]
ask turtle 1 [ ask in-link-neighbors [ set color blue ] ] ;; turtle 0 turns blue
```

in-<breed>-from**in-link-from**`in-<breed>-from turtle``in-link-from turtle`

返回从 *turtle* 到调用者的链。如果没有链存在则返回 nobody。

```
crt 2
ask turtle 0 [ create-link-to turtle 1 ]
ask turtle 1 [ show in-link-from turtle 0 ] ;; shows link 0 1
ask turtle 0 [ show in-link-from turtle 1 ] ;; shows nobody
```

_includes`_includes [filename ...]`

将外部NetLogo源文件（后缀为.nls）包含到模型之中。外部文件可以包括种类、变量、例程定义。每个文件只能使用一次`_includes`。

in-radius`agentset in-radius number`

返回原主体集合中那些与调用者距离小于等于 *number* 的主体形成的集合。（可能包含调用者自身）

与瓦片的距离根据瓦片中心计算。

```
ask turtles
  [ ask patches in-radius 3
    [ set pcolor red ] ]
  ;;= each turtle makes a red "splotch" around itself
```

inspect`inspect agent`

对给定的主体（海龟或瓦片）打开主体监视器（monitor）。

```
inspect patch 2 4
;; an agent monitor opens for that patch
inspect one-of sheep
;; an agent monitor opens for a random turtle from
;; the "sheep" breed
```

int

int *number*

返回数值的整数部分---小数部分丢弃。

```
show int 4.7
=> 4
show int -3.5
=> -3
```

is-agent?**is-agentset?****is-boolean?****is-<*breed*>?****is-directed-link?****is-link?****is-link-set?****is-list?****is-number?****is-patch?****is-patch-set?****is-string?****is-turtle?****is-turtle-set?****is-undirected-link?**

is-agent? *value*
is-agentset? *value*
is-boolean? *value*
is-<*breed*>? *value*
is-directed-link? *value*
is-link? *value*
is-link-set? *value*
is-list? *value*
is-number? *value*
is-patch? *value*
is-patch-set? *value*
is-string? *value*
is-turtle? *value*

`is-turtle-set? value`
`is-directed-link? value`

如果 *value* 是给定的类型，返回 true，否则返回 false。

item

`item index list`
`item index string`

对列表，返回列表某索引项的值。

对字符串，返回字符串某索引项的字符。

注意索引从 0 开始，而不是从 1 开始。（第一项索引为 0，第二项索引为 1，…）

```
;; suppose mylist is [2 4 6 8 10]
show item 2 mylist
=> 6
show item 3 "my-shoe"
=> "s"
```

J

jump

`jump number`



海龟立即前进 *number* 单位（而不是像forward命令那样每次 1 步）

如果因拓扑限制无法跳 *number* 步，则海龟根本不动。

另见 [forward](#), [can-move?](#).

L

label

`label`



这是一个内置的海龟或链变量，它可以保存任何类型的值。给定的值将以文本形式与海龟附着在一起，出现在视图中。通过设置变量值来增加、改变或去除海龟或链的标签。

另见 [label-color](#), [plabel](#), [plabel-color](#)。

例子：

```
ask turtles [ set label who ]
;; all the turtles now are labeled with their
;; who numbers
ask turtles [ set label "" ]
;; all turtles now are not labeled
```

label-color

label-color



这是一个内置的海龟或链变量，它保存一个大于等于 0 小于 140 的值。这个数值决定了海龟或链标签的颜色（如果有标签的话）。通过设置该变量的值改变海龟或链标签的颜色。

另见 [label](#), [plabel](#), [plabel-color](#).

例子：

```
ask turtles [ set label-color red ]
;; all the turtles now have red labels
```

last

last *list*

last *string*

对列表，返回最后一项。

对字符串，返回仅包含原字符串最后一个字符的单字符字符串。

layout-circle

```
layout-circle agentset radius
layout-circle list-of-turtles radius
```

以给定的半径将给定的海龟集合按圆形排列，圆心是世界中心处的瓦片。（如果世界尺寸是偶数，则圆心四舍五入到最近瓦片），海龟指向外部。

如果第一个输入参数是主体集合，则海龟以随机顺序排列。

如果第一个输入参数是列表，则从圆形的顶部开始，海龟们按给定的顺序顺时针排列。（列表中不是海龟的项被忽略）

```
;; in random order
layout-circle turtles 10
;; in order by who number
layout-circle sort turtles 10
;; in order by size
layout-circle sort-by [[size] of ?1 < [size] of ?2] turtles 10
```

__layout-magspring

__layout-magspring *turtle-set* *link-set* *spring-constant* *spring-length*
repulsion-constant *magnetic-field-strength* *magnetic-field-type*
bidirectional?

该命令与[layout-spring](#) 很相似，但增加了一层复杂性。*turtle-set*中的海龟相互吸引和排斥，吸力和斥力取决于他们之间的链（在*link-set*中）。仍然有磁场存在，而这些链试图与磁力线方向一致。

link-set 是链的集合，这些链对他们所连接的海龟施加力的作用。那些与链主体集合中的链相连但不在海龟主体集合中的海龟被看作锚点（anchor）。如果没有海龟具有固定位置，整个网络可能会坍塌。

spring-constant 弹簧紧度（"tautness"）的指标(见 [layout-spring](#))

spring-length 是弹簧的零力（"zero-force"）长度或自然长度。 (见 [layout-spring](#))

repulsion-constant 是节点间斥力的指标(见 [layout-spring](#))。

magnetic-field-strength 是磁场强度（合理值在 0 到 1 之间，但 0.05 是一个较好的默认值）

magnetic-field-type 是 0-10 之间的一个数，下表列出所有选项：

<i>magnetic-field-type</i>	描述
NONE = 0	如果不使用磁场，该命令与 layout-spring 一样。

NORTH = 1	磁力线向北
NORTHEAST = 2	磁力线向东北
EAST = 3	...
SOUTHEAST= 4	...
SOUTH = 5	...
SOUTHWEST= 6	...
WEST = 7	...
NORTHWEST = 8	...
POLAR = 9	磁力线从原点向外放射
CONCENTRIC = 10	磁力线是以原点为中心的同心圆，呈顺时针方向。

如果 *bidirectional?* 为 true，则链在磁场方向和相反方向同时推动与之相连的海龟，实现与磁场的顺应。否则，链只在一个方向推动。

```

to make-a-tree
  set-default-shape turtles "circle"
  crt 5
  ask turtle 0 [
    create-link-with turtle 1
    create-link-with turtle 2
  ]
  ask turtle 1 [
    create-link-with turtle 3
    create-link-with turtle 4
  ]
  ; layout with a fairly strong SOUTH magnetic field
  repeat 50 [ __layout-magspring
    turtles with [who != 0] links 0.3 4 1 .50 5 false ]
end

```

layout-radial

layout-radial *turtle-set* *link-set* *root-agent*

以放射树状布局排列 *turtle-set* 中的海龟，这些海龟通过 *link-set* 的链相连。布局的中心是 *root-agent*，该主体移动到世界视图的中心。

仅有 *link-set* 中的链用来决定布局。如果链所连接的海龟没有包含在 *turtle-set* 中，这些海龟将保持不动。

即使网络中有回路 (cycle) 并不是真正的树结构，该布局仍然可用，不过结果可能不美观。

```
to make-a-tree
  set-default-shape turtles "circle"
  crt 6
  ask turtle 0 [
    create-link-with turtle 1
    create-link-with turtle 2
    create-link-with turtle 3
  ]
  ask turtle 1 [
    create-link-with turtle 4
    create-link-with turtle 5
  ]
  ; do a radial tree layout, centered on turtle 0
  layout-radial turtles (turtle 0) links
end
```

layout-spring

layout-spring *turtle-set* *link-set* *spring-constant* *spring-length*
repulsion-constant

排列 *turtle-set* 中的海龟，*link-set* 中的链就像弹簧，海龟之间互斥。与 *link-set* 中的链有连接但却不在 *turtle-set* 中的海龟被当作锚点，保持不动。

spring-constant 是弹簧紧度 ("tautness") 指标，是改变长度时的阻力，是弹簧长度改变 1 个单位时产生的力。

spring-length 是弹簧的零力长度 ("zero-force") 或自然长度。这是弹簧节点受拉或受压时弹簧试图保持的长度。

repulsion-constant 是节点间斥力指标，是相距单位长度的两个节点之间的斥力。

斥力试图将两个节点尽量分开，以免挤在一起。弹簧试图将所连接的两个节点保持一定的距离。综合的结果就是整个网络布局突出了节点之间的关系，不太拥挤，又美观。

布局算法基于Fruchterman-Reingold算法。关于算法的详细信息参见[here](#)。

```
to make-a-triangle
```

```

set-default-shape turtles "circle"
crt 3
ask turtle 0 [
  create-links-with other turtles
]
ask turtle 1 [
  create-link-with turtle 2
]
repeat 30 [ layout-spring turtles links 0.2 5 1 ] ;; lays the nodes in a triangle
end

```

layout-tutte

layout-tutte *turtle-set link-set radius*

与 *link-set* 中的链有连接但却不在 *turtle-set* 中的海龟，按给定的 *radius*，呈圆形布局。主体集合中必须至少 3 个主体。

turtle-set 中的海龟以下列方式布局：每个海龟被放置到由与它相连的邻居所形成的多边形的质心（centroid）。（质心类似 2 维上邻居坐标的平均）

（锚点主体（“anchor agents”）形成的圆形用来防止所有海龟坍塌到一点）

经过几次迭代，布局变稳定。

该布局得名于数学家 William Thomas Tutte，是他提出了这种图布局方法。

```

to make-a-tree
  set-default-shape turtles "circle"
  crt 6
  ask turtle 0 [
    create-link-with turtle 1
    create-link-with turtle 2
    create-link-with turtle 3
  ]
  ask turtle 1 [
    create-link-with turtle 4
    create-link-with turtle 5
  ]
; place all the turtles with just one

```

```

; neighbor on the perimeter of a circle
; and then place the remaining turtles inside
; this circle, spread between their neighbors.
repeat 10 [ layout-tutte turtles
  (turtles with [count link-neighbors = 1]) 12 ]
end

```

left**lt****left** *number*海龟左转 *number* 度。 (如果 *number* 为负, 则右转)**length****length** *list***length** *string*

返回给定列表的项数, 或给定字符串的字符数。

let**let** *variable* *value*

创建一个新的局部变量并赋值。局部变量仅存在于闭合的命令块中。

如果后面要改变该局部变量的值, 使用[set](#)。

例子:

```

let prey one-of sheep-here
if prey != nobody
  [ ask prey [ die ] ]

```

link**link** *end1* *end2* <breed> *end1* *end2*

给定端点的who number，返回连接这两个海龟的链。如果没有满足条件的链，则返回nobody。要引用有种类的链，必须与端点一起使用种类的单数形式。

```
ask link 0 1 [ set color green ]
;; unbreeded link connecting turtle 0 and turtle 1 will turn green
ask directed-link 0 1 [ set color red ]
;; directed link connecting turtle 0 and turtle 1 will turn red
```

另见 [patch-at](#).

link-heading

link-heading



返回链从end1到 end2 所在方向的度数（至少为 0， 小于 360）。如果端点在相同位置，则抛出运行错误。

```
ask link 0 1 [ print link-heading ]
;; prints [[towards other-end] of end1] of link 0 1
```

另见 [link-length](#)

link-length

link-length



返回链的两个端点之间的距离。

```
ask link 0 1 [ print link-length ]
;; prints [[distance other-end] of end1] of link 0 1
```

另见 [link-heading](#)

link-set

link-set *value*
(**link-set** *value1* *value2* ...)

返回输入参数中的所有链组成的集合。输入可以是单个链、链主体集合、nobody，或者是包括上面所有类型的列表(或嵌套列表)。

```
link-set self
link-set [my-links] of nodes with [color = red]
```

另见 [turtle-set](#), [patch-set](#).

link-shapes

link-shapes

返回一个字符串列表，该列表包含模型中的所有链图形。

可以创建新图形，或从其他模型输入。在链图形编辑器 ([Link Shapes Editor](#)) 中操作。

```
show link-shapes
=> ["default"]
```

links

links

返回由所有链组成的主体集合。

```
show count links
;; prints the number of links
```

links-own

<link-breeds>-own

```
links-own [var1 ...]
<link-breeds>-own [var1 ...]
```

该关键词与 `globals`, `breed`, `<breeds>-own`, `turtles-own`, `patches-own`一样，只能在程序的首部使用，位于所有例程定义之前。它定义属于每个链的变量。

如果指定了种类，只有那个种类的链具有所定义的链变量。（多个种类可能有相同的变量）

```
undirected-link-breed [sidewalks sidewalk]
directed-link-breed [streets street]
links-own [traffic]    ;; applies to all breeds
sidewalks-own [pedestrians]
streets-own [cars bikes]
```

list

```
list value1 value2
(list value1 ...)
```

返回包含给定项的列表。列表项可以是任何类型，可以由任何种类的报告器生成。

```
show list (random 10) (random 10)
=> [4 9] ;; or similar list
show (list 5)
=> [5]
show (list (random 10) 1 2 3 (random 10))
=> [4 1 2 3 9] ;; or similar list
```

ln

```
ln number
```

返回 *number* 的自然对数，即以 e (2.71828...) 为底的对数。

另见 [e](#), [log](#).

log

```
log number base
```

返回以 *base* 为底 *number* 的对数。

```
show log 64 2
=> 6
```

另见 [ln](#).

loop

```
loop [ commands ]
```

不断重复运行命令块，或者一直运行，直到通过使用[stop](#)或[report](#)使当前例程退出。

注意：一般应使用永久性按钮实现重复运行，其优点在于可以通过点击按钮停止循环。

lput

lput *value list*

在列表尾部增加一项 *value*, 并返回新列表。

```
;; suppose mylist is [2 7 10 "Bob"]
set mylist lput 42 mylist
;; mylist now is [2 7 10 "Bob" 42]
```

M**map**

map [*reporter*] *list*
 $(\text{map } [\text{reporter}] \text{ } \textit{list}_1 \dots)$

如果给定一个列表, 则对列表的每一项运行给定的报告器, 运行结果收集到一个列表中并返回。

在报告器*reporter*, 使用 [?](#) 引用列表*list*的当前项。

```
show map [round ?] [1.1 2.2 2.7]
=> [1 2 3]
show map [? * ?] [1 2 3]
=> [1 4 9]
```

如果给定多个列表, 则对所有列表形成的每一组运行给定的报告器。因此对所有列表的第一项运行一次, 第二项运行一次, 等等。所有列表必须有相同的项数。

在报告器*reporter*, 使用 [?1](#) 到 [?n](#) 引用每个列表的当前项。

下面的例子帮你弄的明白一些:

```
show (map [?1 + ?2] [1 2 3] [2 4 6])
=> [3 6 9]
show (map [?1 + ?2 = ?3] [1 2 3] [2 4 6] [3 5 9])
=> [true false true]
```

另见 [foreach](#), [?](#)。

max**max** *list*

返回列表中的最大数值，忽略其他非数值型项。

```
show max [xcor] of turtles
;; prints the x coordinate of the turtle which is
;; farthest right in the world
```

max-n-of**max-n-of** *number agentset [reporter]*

返回一个包含 *number* 个主体的主体集合，该集合由 *agentset* 中具有最高 *reporter* 值的主体组成。主体集合的构造方法是：找出具有最高 *reporter* 值的主体，如果不足 *number* 个，则继续寻找次高值，依次进行。最后，如果具有同一数值的主体有多个，如果都选入导致总数超过 *number*，则随机选出所需个数的主体。

```
; assume the world is 11 x 11
show max-n-of 5 patches [pxcor]
;; shows 5 patches with pxcor = max-pxcor
show max-n-of 5 patches with [pycor = 0] [pxcor]
;; shows an agentset containing:
;; (patch 1 0) (patch 2 0) (patch 3 0) (patch 4 0) (patch 5 0)
```

另见 [max-one-of](#), [with-max](#)。**max-one-of****max-one-of** *agentset [reporter]*

返回主体集合中具有最高 reporter 返回值的主体。如果具有最高值的主体有多个，则随机选出一个。如果想得到所有具有最高值的主体，使用 [with-max](#)。

```
show max-one-of patches [count turtles-here]
;; prints the first patch with the most turtles on it
```

另见 [max-n-of](#), [with-max](#)。

max-pxcor**max-pycor****max-pxcor****max-pycor**

返回瓦片的最大 x 坐标和最大 y 坐标，它们决定世界的大小。

与老版本 NetLogo 不同，当前版本中原点不一定位于世界中心。然而，最大 x 坐标和最大 y 坐标必须大于等于 0。

注意：仅通过编辑视图就能设置世界大小— these are reporters which cannot be set.

```
crt 100 [ setxy random-float max-pxcor
            random-float max-pycor ]
;; distributes 100 turtles randomly in the
;; first quadrant
```

另见 [min-pxcor](#), [min-pycor](#), [world-width](#), [world-height](#)

mean**mean** *list*

返回给定列表各项的统计平均值，忽略非数值项。均值即各项之和除以项数。

```
show mean [xcor] of turtles
;; prints the average of all the turtles' x coordinates
```

median**median** *list*

返回给定列表中各数值项的中位数，忽略非数值项。中位数就是将各项按顺序排列后的中间项。（如果中间是两项，则取二者的平均）

```
show median [xcor] of turtles
;; prints the median of all the turtles' x coordinates
```

member?

member? *value list*
member? *string1 string2*
member? *agent agentset*

对列表，如果给定的 *value* 在列表中则返回 true，否则返回 false。

对字符串，判断 *string1* 是否是 *string2* 的子串。

对主体集合，判断给定的主体是否在主体集合之中。

```
show member? 2 [1 2 3]
=> true
show member? 4 [1 2 3]
=> false
show member? "bat" "abate"
=> true
show member? turtle 0 turtles
=> true
show member? turtle 0 patches
=> false
```

另见 [position](#)。

min

min *list*

返回列表中的最小数值，忽略其他类型的项。

```
show min [xcor] of turtles
;; prints the lowest x-coordinate of all the turtles
```

min-n-of

min-n-of *number agentset [reporter]*

返回一个包含 *number* 个主体的主体集合，该集合由 *agentset* 中具有最小 *reporter* 值的主体组成。主体集合的构造方法是：找出具有最小 *reporter* 值的主体，如果不足 *number* 个，

则继续寻找次小值，依次进行。最后，如果具有同一数值的主体有多个，如果都选入导致总数超过 *number*，则随机选出所需个数的主体。

```
; assume the world is 11 x 11
show min-n-of 5 patches [pxcor]
;; shows 5 patches with pxc当地 = min-pxcor
show min-n-of 5 patches with [pycor = 0] [pxcor]
;; shows an agentset containing:
;; (patch -5 0) (patch -4 0) (patch -3 0) (patch -2 0) (patch -1 0)
```

另见 [min-one-of](#), [with-min](#)。

min-one-of

min-one-of *agentset* [*reporter*]

返回主体集合中具有最小 reporter 返回值的主体。如果具有最小值的主体有多个，则随机选出一个。如果想得到所有具有最小值的主体，使用 [with-min](#)。

```
show min-one-of turtles [xcor + ycor]
;; reports the first turtle with the smallest sum of
;; coordinates
```

另见 [with-min](#), [min-n-of](#)。

min-pxcor

min-pycor

min-pxcor

min-pycor

返回瓦片的最小 x 坐标和最小 y 坐标，它们决定世界的大小。

与老版本 NetLogo 不同，当前版本中原点不一定位于世界中心。然而，最小 x 坐标和最小 y 坐标必须小于等于 0。

注意：仅通过编辑视图就能设置世界大小— these are reporters which cannot be set.

```
crt 100 [ setxy random-float min-pxcor
            random-float min-pycor ]
;; distributes 100 turtles randomly in the
```

```
;; third quadrant
```

另见 [max-pxcor](#), [max-pycor](#), [world-width](#), [world-height](#)

mod

number1 mod number2

返回 *number1* 模除 *number2*: 即 $number1 \pmod{number2}$ 的余数。与下面的代码等价:

```
number1 - (floor (number1 / number2)) * number2
```

注意 *mod* 是中缀运算, 在两个输入参数之间。

```
show 62 mod 5
=> 2
show -8 mod 3
=> 1
```

另见 [remainder](#)。*mod* 和 *remainder*对正数一样, 但对负数不一样。

modes

modes list

返回 *list* 中出现次数最多的项⁶ (一项或多项) 所组成的列表。

输入参数列表可以包括任何类型 NetLogo 值。

如果输入列表是空列表, 则返回空列表。

```
show modes [1 2 2 3 4]
=> [2]
show modes [1 2 2 3 3 4]
=> [2 3]
show modes [[1 2 [3]] [1 2 [3]] [2 3 4] ]
=> [[1 2 [3]]]
show modes [pxcor] of turtles
;; shows which columns of patches have the most
;; turtles on them
```

⁶ 译者注: 统计学上的众数。

mouse-down?**mouse-down?**

如果鼠标按钮按下则返回 true，否则返回 false。

注意：如果鼠标指针在当前视图之外，`mouse-down?` 总返回 false。**mouse-inside?****mouse-inside?**

如果鼠标指针在当前视图之内则返回 true，否则返回 false。

mouse-patch**mouse-patch**如果鼠标指针指向一个瓦片则返回该瓦片，否则返回 [nobody](#)。**mouse-xcor****mouse-ycor****mouse-xcor****mouse-ycor**返回 2 维视图中鼠标的 x 或 y 坐标。坐标值采用海龟坐标，因此不必是整数。如果想得到瓦片坐标，使用 `round mouse-xcor` 和 `round mouse-ycor`。

注意：如果鼠标不在 2 维视图中，返回它最后一刻在视图中的坐标值。

```
;; to make the mouse "draw" in red:  
if mouse-down?  
[ ask patch mouse-xcor mouse-ycor [ set pcolor red ] ]
```

move-to**move-to agent**

海龟将其 x 和 y 坐标设置为与给定主体 *agent* 的相同。

(如果给定主体是瓦片，则效果就是海龟移动到瓦片中心)

```
move-to turtle 5
;; turtle moves to same point as turtle 5
move-to one-of patches
;; turtle moves to the center of a random patch
move-to max-one-of turtles [size]
;; turtle moves to same point as biggest turtle
```

注意海龟的方向未变。也许需要先使用[face](#) 命令将海龟的方向调整为运动方向。

另见 [setxy](#) 。

movie-cancel

movie-cancel

取消当前电影。

movie-close

movie-close

停止录制当前电影。

movie-grab-view

movie-grab-interface

movie-grab-view
movie-grab-interface

将一幅当前视图或界面面板的图像添加到当前电影。

```
;; make a 20-step movie of the current view
setup
movie-start "out.mov"
repeat 20 [
  movie-grab-view
  go
```

```
]  
movie-close
```

movie-set-frame-rate

movie-set-frame-rate *frame-rate*

设置当前电影的帧频，帧频单位是每秒多少帧。（如果不明确设置帧频，默认为 15 帧/秒）

该命令必须在[movie-start](#) 之后，[movie-grab-view](#) 或 [movie-grab-interface](#) 之前调用。

另见 [movie-status](#) 。

movie-start

movie-start *filename*

创建一个新电影。电影保存到*filename*指定的QuickTime文件，该文件的后缀应为".mov"。

另见[movie-grab-view](#), [movie-grab-interface](#), [movie-cancel](#), [movie-status](#),
[movie-set-frame-rate](#), [movie-close](#) 。

movie-status

movie-status

返回一个描述当前电影的字符串。

```
print movie-status  
=> No movie.  
movie-start  
print movie-status  
=> 0 frames; frame rate = 15.  
movie-grab-view  
print movie-status  
1 frames; frame rate = 15; size = 315x315.
```

my-<breeds>**my-links****my-<breeds>****my-links**

返回与调用者连接的所有无向链组成的主体集合。

```
crt 5
ask turtle 0
[
  create-links-with other turtles
  show my-links ;; prints the agentset containing all links
    ;; (since all the links we created were with turtle 0 )
]
ask turtle 1
[
  show my-links ;; shows an agentset containing the link 0 1
]
end
```

my-in-<breeds>**my-in-links****my-in-<breeds>****my-in-links**

返回所有从其他节点出发到达调用者的有向链组成的主体集合。

```
crt 5
ask turtle 0
[
  create-links-to other turtles
  show my-in-links ;; shows an empty agentset
]
ask turtle 1
[
```

```
show my-in-links ;; shows an agentset containing the link 0 1
]
```

my-out-<breeds>

my-out-links



返回所有从调用者出发到达其他节点的有向链组成的主体集合。

```
crt 5
ask turtle 0
[
  create-links-to other turtles
  show my-out-links ;; shows agentset containing all the links
]
ask turtle 1
[
  show my-out-links ;; shows an empty agentset
]
```

myself

myself



“self”与“myself”大不相同。“self”很简单，就是指我(“me”)。“myself”是指“请求我做目前正在做的事情的海龟或瓦片”。

当主体被请求运行代码时，在代码中使用 `myself` 返回发出请求的主体（海龟或瓦片）。

`myself` 经常与 `of` 一起使用，用来读取或设置请求发出主体的变量。

`myself` 不仅可以用在 `ask` 命令的代码块里，还可用在 `hatch`, `sprout`, `of`, `with`, `all?`, `with-min`, `with-max`, `min-one-of`, `max-one-of`, `min-n-of`, `max-n-of`。

```
ask turtles
[ ask patches in-radius 3
  [ set pcolor [color] of myself ] ]
```

```
;; each turtle makes a colored "splotch" around itself
```

在代码例子“*Myself Example*”里有许多实例。

另见 [self](#)。

N

n-of

```
n-of size agentset
n-of size list
```

对主体集合，从输入主体集合中随机选取（不重复）*size*个主体组成一个主体集合，返回该主体集合。

对列表，从输入列表中随机选取（不重复）*size*项组成一个列表，返回该列表。结果列表中各项的顺序与原列表中的顺序一致。（如果需要随机顺序，对结果使用 `shuffle`）

如果 *size* 大于输入集合（列表）的项数，则出错。

```
ask n-of 50 patches [ set pcolor green ]
;; 50 randomly chosen patches turn green
```

另见 [one-of](#)。

n-values

```
n-values size [reporter]
```

返回一个项数为 *size* 的列表，其中的各项是重复运行 *reporter* 得到的。

在 *reporter* 中使用 [?](#) 引用当前被计算过的项数，项数从 0 开始。

```
show n-values 5 [1]
=> [1 1 1 1 1]
show n-values 5 [?]
=> [0 1 2 3 4]
show n-values 3 [turtle ?]
=> [(turtle 0) (turtle 1) (turtle 2)]
show n-values 5 [? * ?]
=> [0 1 4 9 16]
```

另见 [reduce](#), [filter](#), [?](#)。

neighbors

neighbors4

neighbors

neighbors4



返回由 8 个相邻瓦片（邻元）或 4 个相邻瓦片（邻元）组成的主体集合。

```
show sum [count turtles-here] of neighbors
;; prints the total number of turtles on the eight
;; patches around the calling turtle or patch
show count turtles-on neighbors
;; a shorter way to say the same thing
ask neighbors4 [ set pcolor red ]
;; turns the four neighboring patches red
```

<breed>-neighbors

link-neighbors

<breed>-neighbors

link-neighbors



返回与调用海龟相连的所有无向链上另一端海龟组成的主体集合。

```
crt 3
ask turtle 0
[
  create-links-with other turtles
  ask link-neighbors [ set color red ] ;; turtles 1 and 2 turn red
]
ask turtle 1
[
  ask link-neighbors [ set color blue ] ;; turtle 0 turns blue
]
end
```

<breed>-neighbor?**link-neighbor?**

`<breed>-neighbor? turtle
link-neighbor? turtle`


如果在 *turtle* 和调用者之间有无向链，则返回 true。

```
crt 2
ask turtle 0
[
  create-link-with turtle 1
  show link-neighbor? turtle 1    ;;= prints true
]
ask turtle 1
[
  show link-neighbor? turtle 0    ;;= prints true
]
```

netlogo-applet?**netlogo-applet?**

如果模型以 applet 方式运行则返回 true 。

netlogo-version**netlogo-version**

返回包含当前运行的 NetLogo 版本号的字符串。

```
show netlogo-version
=> "4.0.2"
```

new-seed**new-seed**

返回一个适合作为随机数发生器种子的数值。

`new-seed` 产生的数值基于当前日期和时间的毫秒数，并且在 NetLogo 允许的整数范围 -9007199254740992 到 9007199254740992 内。

`new-seed` 相连的两个返回值不会相同。（实现方法是如果当前毫秒数产生的种子已经用过，则再等一个毫秒）

另见 [random-seed.](#)

no-display

no-display

关闭当前视图的更新，直到发出 `display` 命令。有两个主要用途：

一、控制用户在什么时刻看到更新。你可能在视图后面改变许多东西，不想让用户看到，然后一下子全显示出来。

二、如果关闭视图更新，则模型运行更快。如果你很忙的话，该命令能让你更快的得到结果。
(注意一般不需使用 `no-display` 命令，你可以使用视图控制条上的 on/off 开关)

注意 `display` 与 `no-display` 和视图控制条上的视图冻结开关相互独立。

另见 [display.](#)

nobody

nobody

这是一个特殊值，有些原语如 `turtle`, `one-of`, `max-one-of` 等用来说明没有找到主体。另外，当主体死亡后，它也等于 `nobody`。

注意：空主体集合不等于 `nobody`，如果要测试主体集合是否为空，使用 [any?](#)。只有预期得到单一主体的地方才可能得到 `nobody`。

```
set other one-of other turtles-here
if other != nobody
  [ ask other [ set color red ] ]
```

no-links

no-links

返回一个空的链主体集合。

no-patches

no-patches

返回一个空的瓦片主体集合。

not

not *boolean*

如果 *boolean* 为 false 返回 true，否则返回 false。

```
if not any? turtles [ crt 10 ]
```

no-turtles

no-turtles

返回一个空的海龟主体集合。

O**of**

[reporter] of agent
[reporter] of agentset

对主体，返回 *agent*（海龟或瓦片）的 *reporter* 值。

```
show [pxcor] of patch 3 5
;; prints 3
show [pxcor] of one-of patches
;; prints the value of a random patch's pxcor variable
show [who * who] of turtle 5
=> 25
show [count turtles in-radius 3] of patch 0 0
;; prints the number of turtles located within a
;; three-patch radius of the origin
```

对主体集合，返回一个列表，该列表包括主体集合中所有主体的 *reporter* 值（随机顺序）。

```
crt 4
show sort [who] of turtles
=> [0 1 2 3]
show sort [who * who] of turtles
=> [0 1 4 9]
```

one-of

one-of agentset
one-of list

对主体集合，返回随机选择的一个主体。如果主体集合为空，返回[nobody](#)。

对列表，返回随机选择的一个列表项。如果列表为空则出错。

```
ask one-of patches [ set pcolor green ]
;; a random patch turns green
ask patches with [any? turtles-here]
  [ show one-of turtles-here ]
;; for each patch containing turtles, prints one of
;; those turtles

;; suppose mylist is [1 2 3 4 5 6]
show one-of mylist
;; prints a value randomly chosen from the list
```

另见 [n-of](#)。

or

boolean1 or boolean2

如果 *boolean1* 或 *boolean2*之一或全部为 true，则返回 true。

注意如果 *condition1* 为 true，*condition2*不再执行（因为对结果无影响）

```
if (pxcor > 0) or (pycor > 0) [ set pcolor red ]
;; patches turn red except in lower-left quadrant
```

other**other *agentset***

返回一个主体集合，该主体集合与输入主体集合相同，只是不包括调用主体。

```
show count turtles-here
=> 10
show count other turtles-here
=> 9
```

other-end**other-end**

如果由海龟执行，返回请求链另一端的海龟。

如果由链执行，返回链上不是请求海龟的另一个海龟。

这些定义很难抽象理解，下面的例子对你有所帮助：

```
ask turtle 0 [ create-link-with turtle 1 ]
ask turtle 0 [ ask link 0 1 [ show other-end ] ] ; prints turtle 1
ask turtle 1 [ ask link 0 1 [ show other-end ] ] ; prints turtle 0
ask link 0 1 [ ask turtle 0 [ show other-end ] ] ; prints turtle 1
```

正如例子希望说明白的，另一端（“other” end）是既不是请求者也不是被请求者的端点。

out-<breed>-neighbor?**out-link-neighbor?****out-<breed>-neighbor? *turtle*****out-link-neighbor? *turtle***如果从调用者到 *turtle*有一条有向链，则返回 true 。

```
crt 2
ask turtle 0 [
```

```

create-link-to turtle 1
show in-link-neighbor? turtle 1 ;; prints false
show out-link-neighbor? turtle 1 ;; prints true
]
ask turtle 1 [
  show in-link-neighbor? turtle 0 ;; prints true
  show out-link-neighbor? turtle 0 ;; prints false
]

```

out-<breed>-neighbors**out-link-neighbors**

`out-<breed>-neighbors`
`out-link-neighbors`



返回一个海龟主体集合，这些海龟有从调用者出发到达它的有向链。

```

crt 4
ask turtle 0
[
  create-links-to other turtles
  ask out-link-neighbors [ set color pink ] ;; turtles 1-3 turn pink
]
ask turtle 1
[
  ask out-link-neighbors [ set color orange ] ;; no turtles change colors
  ;; since turtle 1 only has in-links
]
end

```

out-<breed>-to**out-link-to**

`out-<breed>-to turtle`
`out-link-to turtle`



返回从调用者到 *turtle* 的链。如果没有链则返回 nobody 。

```
crt 2
ask turtle 0 [
  create-link-to turtle 1
  show out-link-to turtle 1 ;; shows link 0 1
]
ask turtle 1
[
  show out-link-to turtle 0 ;; shows nobody
]
```

output-print**output-show****output-type****output-write****output-print** *value***output-show** *value***output-type** *value***output-write** *value*

这些命令与[print](#), [show](#), [type](#), [write](#)相同, 只不过*value*不是显示在命令中心, 而是在模型的输出区域。 (如果模型没有独立的输出区域, 则显示在命令中心)

P**patch****patch** *xcor* *ycor*

给出点的 x 和 y 坐标, 返回包含该点的瓦片。 (此处为绝对坐标, 而不是像 patch-at 那样基于调用主体的相对坐标)

如果 x 和 y 是整数, 该点就是瓦片的中心。如果不是整数, 四舍五入为整数, 确定瓦片。

如果世界拓扑允许回绕, 坐标会回绕到世界之内。如果不允许回绕而坐标超出世界范围, 则返回 nobody。

```
ask patch 3 -4 [ set pcolor green ]
;; patch with pxcor of 3 and pycor of -4 turns green
```

```
show patch 1.2 3.7
;; prints (patch 1 4); note rounding
show patch 18 19
;; supposing min-pxcor and min-pycor are -17
;; and max-pxcor and max-pycor are 17,
;; in a wrapping topology, prints (patch -17 -16);
;; in a non-wrapping topology, prints nobody
```

另见 [patch-at](#) 。

patch-ahead

patch-ahead *distance*



返回沿调用海龟当前方向前方给定距离处的单个瓦片。如果超出世界范围，瓦片不存在，则返回 nobody 。

```
ask patch-ahead 1 [ set pcolor green ]
;; turns the patch 1 in front of the calling turtle
;; green; note that this might be the same patch
;; the turtle is standing on
```

另见 [patch-at](#), [patch-left-and-ahead](#), [patch-right-and-ahead](#),
[patch-at-heading-and-distance](#)

patch-at

patch-at *dx dy*



返回相对调用者(dx, dy)处的瓦片，即距离调用主体东方 dx、北方 dy 的瓦片。

如果该点超出了非回绕世界的边界，因此没有这样的瓦片，则返回 nobody 。

```
ask patch-at 1 -1 [ set pcolor green ]
;; if caller is a turtle or patch, turns the
;; patch just southeast of the caller green
```

另见 [patch](#), [patch-ahead](#), [patch-left-and-ahead](#), [patch-right-and-ahead](#),
[patch-at-heading-and-distance](#)

patch-at-heading-and-distance

`patch-at-heading-and-distance heading distance`



`patch-at-heading-and-distance` 返回沿给定绝对方向离调用海龟或瓦片给定距离处的单个瓦片。（与 `patch-left-and-ahead` 和 `patch-right-and-ahead` 相反，此处不考虑调用主体的当前方向）。如果因超出世界范围而瓦片不存在，则返回 `nobody`。

```
ask patch-at-heading-and-distance -90 1 [ set pcolor green ]
;; turns the patch 1 to the west of the calling patch
;; green
```

另见 [patch](#), [patch-at](#), [patch-left-and-ahead](#), [patch-right-and-ahead](#)

patch-here

`patch-here`



返回海龟下方的瓦片。

注意瓦片不能用该报告器，因为瓦片只能说“self”。

patch-left-and-ahead

patch-right-and-ahead

`patch-left-and-ahead angle distance`

`patch-right-and-ahead angle distance`



返回沿调用海龟当前方左转或右转给定角度（度数），相距给定距离处的单个瓦片。如果因超出世界范围而瓦片不存在，则返回 `nobody`。

（如果想在绝对方向寻找瓦片，而不是与当前海龟方向的相对方向，则使用 `patch-at-heading-and-distance`）

```
ask patch-right-and-ahead 30 1 [ set pcolor green ]
;; the calling turtle "looks" 30 degrees right of its
;; current heading at the patch 1 unit away, and turns
;; that patch green; note that this might be the same
;; patch the turtle is standing on
```

另见 [patch](#), [patch-at](#), [patch-at-heading-and-distance](#)

patch-set

```
patch-set value1
(patch-set value1 value2 ...)
```

返回一个包含所有输入瓦片的主体集合。输入可以是单个瓦片、瓦片主体集合、nobody 以及包含以上任何项的列表(或嵌套列表)。

```
patch-set self
patch-set patch-here
(patch-set self neighbors)
(patch-set patch-here neighbors)
(patch-set patch 0 0 patch 1 3 patch 4 -2)
(patch-set patch-at -1 1 patch-at 0 1 patch-at 1 1)
patch-set [patch-here] of turtles
patch-set [neighbors] of turtles
```

另见 [turtle-set](#), [link-set](#)

patches

patches

返回包含所有瓦片的主体集合。

patches-own

patches-own [*var1* ...]

该关键词与 `globals`, `breed`, `<breed>-own`, `turtles-own` 一样, 只能用在程序首部, 在任何例程定义之前。它定义所有瓦片可用的变量。

所有瓦片将具有给定的变量, 能够使用该变量。

所有瓦片变量可以由瓦片上方的海龟访问。

另见 [globals](#), [turtles-own](#), [breed](#), [<breeds>-own](#)

pcolor**pcolor**

这是一个内置瓦片变量，保存瓦片的颜色。设置这个变量改变瓦片颜色。

所有瓦片变量可以由瓦片上方的海龟直接访问。颜色可以用NetLogo颜色（一个数值）或RGB颜色（3个数的列表）。细节见编程指南的颜色部分[Colors section](#)。

另见 [color](#)

pen-down**pd****pen-erase****pe****pen-up****pu****pen-down****pen-erase****pen-up**

海龟改变画线模式：画线、擦线、既不画也不擦。线总是显示在瓦片之上和海龟之下。要改变笔的颜色，则使用set color改变海龟的颜色。

注意：当海龟画笔放下时，所有的移动命令导致画线，包括 jump, setxy 和 move-to。

注意：这些命令等价于设置海龟变量“pen-mode”为“down”，“up”，“erase”。

注意：在 Windows 上画线和擦线可能不会擦除所有像素。

pen-mode

这是一个内置海龟变量，保存海龟画笔的状态。设置该变量进行画线、擦线或既不画也不擦，可能的值为“up”，“down”，“erase”。

pen-size



这是一个内置海龟变量，保存线宽的像素数。当画笔放下（或擦除）时用来画线（擦线）。

plabel

plabel



这是一个内置瓦片变量，可能保存任何类型的值。给定的值以文本形式与瓦片附着在一起，显示在视图中。设置该变量来增加、改变、移除瓦片的标签。

所有瓦片变量可以由瓦片上方的海龟直接访问。

另见 [plabel-color](#), [label](#), [label-color](#)

plabel-color

plabel-color



这是一个内置瓦片变量，保存一个大于等于 0 小于 140 的数值。该数值决定了瓦片标签的颜色（如果有标签的话）。设置该变量改变瓦片标签的颜色。

所有瓦片变量可以由瓦片上方的海龟直接访问。

另见 [plabel](#), [label](#), [label-color](#)

plot

plot *number*

画笔的 x 值增加 plot-pen-interval，在新的 x 值以及 y 值为 *number* 处画一个点。（在绘图上第一次使用该命令时，x 值是 0）

plot-name**plot-name**

返回当前绘图的名字（字符串）。

plot-pen-exists?**plot-pen-exists? *string***

如果当前绘图中存在给定名字的画笔则返回 true，否则返回 false。

plot-pen-down**plot-pen-up****plot-pen-down****plot-pen-up**

放下(抬起)当前画笔，实现画(不画)图。(默认所有画笔初始时是放下的)

plot-pen-reset**plot-pen-reset**

清除当前画笔画过的所有东西，移到(0, 0)处，放下。如果画笔是永久画笔，则根据绘图编辑对话框中的设置，将画笔的颜色和模式设为默认值。

plotxy**plotxy *number1* *number2***将当前画笔移动到坐标(*number1*, *number2*)。如果画笔是放下的，则绘制线、条形、点(取决于笔的模式)。

plot-x-min**plot-x-max****plot-y-min****plot-y-max**`plot-x-min``plot-x-max``plot-y-min``plot-y-max`

返回当前绘图的 x、y 轴最小、最大值。

这些值可以使用 `set-plot-x-range` 和 `set-plot-y-range` 命令设置。（它们的默认值在绘图编辑对话框中设置）

position`position item list``position string1 string2`

对列表，返回 *item* 在 *list* 第一次出现的位置，如果没出现则返回 `false`。

对字符串，返回 *string1* 作为子串第一次在 *string2* 中出现的位置，如果没出现则返回 `false`。

注意：位置编号从 0 开始。

```
;; suppose mylist is [2 7 4 7 "Bob"]
show position 7 mylist
=> 1
show position 10 mylist
=> false
show position "in" "string"
=> 3
```

另见 [member?](#)

precision

`precision number places`

返回将 *number* 四舍五入到小数点后 *places* 位的值。

如果 *places* 是负数，舍入发生在小数点前几位。

```
show precision 1.23456789 3
=> 1.235
show precision 3834 -3
=> 4000
```

print

`print value`

在命令中心显示 *value*, 后跟回车。

与 [show](#) 不同， *value* 前不显示调用主体。

另见 [show](#), [type](#), [write](#)

另见 [output-print](#)

pxcor

pycor

pxcor

pycor



这些是内置瓦片变量，保存瓦片的 x, y 坐标，它们总是整数。因为瓦片不会移动，所以不能设置这些变量。

pxcor 大于等于 min-pxcor，小于等于 max-pxcor； pycor 大于等于 min-pycor，小于等于 max-pycor 。

所有瓦片变量可以由瓦片上方的海龟直接访问。

另见 [xcor](#), [ycor](#)

R

random

random *number*

如果 *number* 为正，返回大于等于 0、小于 *number* 的一个随机整数。

如果 *number* 为负，返回小于等于 0、大于 *number* 的一个随机整数。

如果 *number* 为 0，返回 0。

注意：在NetLogo2.0 前，如果给定非整数输入，则返回浮点数。现在不是这样了。如果要得到浮点数，必须使用[random-float](#)。

```
show random 3
;; prints 0, 1, or 2
show random -3
;; prints 0, -1, or -2
show random 3.5
;; prints 0, 1, 2, or 3
```

另见 [random-float](#)

random-float

random-float *number*

如果 *number* 为正，返回大于等于 0、小于 *number* 的一个随机浮点数。

如果 *number* 为负，返回小于等于 0、大于 *number* 的一个随机浮点数。

如果 *number* 为 0，返回 0。

```
show random-float 3
;; prints a number at least 0 but less than 3,
;; for example 2.589444906014774
show random-float 2.5
;; prints a number at least 0 but less than 2.5,
;; for example 1.0897423196760796
```

random-exponential**random-gamma****random-normal****random-poisson**

`random-exponential mean`

`random-gamma alpha lambda`

`random-normal mean standard-deviation`

`random-poisson mean`

根据均值 *mean* 返回服从相应分布的随机数，对正态分布还要给出标准差 *standard-deviation*。

`random-exponential` 返回服从指数分布的随机浮点数。

`random-gamma` 返回服从伽马分布的随机浮点数，分布参数由浮点数 *alpha* 和 *lambda* 给出，二者必须大于 0。（如果已知均值和方差的话，输入形式为 *alpha* = *mean* * *mean* / *variance*；*lambda* = 1 / (*variance* / *mean*)）

`random-normal` 返回服从正态分布的随机浮点数。

`random-poisson` 返回服从 Poisson-distributed 的随机整数。

```
show random-exponential 2
;; prints an exponentially distributed random floating
;; point number with a mean of 2
show random-normal 10.1 5.2
;; prints a normally distributed random floating point
;; number with a mean of 10.1 and a standard deviation
;; of 5.2
show random-poisson 3.4
;; prints a Poisson-distributed random integer with a
;; mean of 3.4
```

random-pxcor**random-pycor**

`random-pxcor`

`random-pycor`

返回一个随机整数，处于 `min-pxcor` (或 `-y`) 到 `max-pxcor` 或 `-y` 的闭区间

```
ask turtles [
  ;; move each turtle to the center of a random patch
  setxy random-pxcor random-pycor
]
```

另见 [random-xcor](#), [random-ycor](#)

random-seed

`random-seed` *number*

将伪随机数发生器的种子设为 *number* 的整数部分。种子可以是 NetLogo 整数区间 (-9007199254740992 到 9007199254740992) 中的任何整数。

细节参见编程指南的随机数[Random Numbers](#)部分。

```
random-seed 47823
show random 100
=> 57
show random 100
=> 91
random-seed 47823
show random 100
=> 57
show random 100
=> 91
```

random-xcor

random-ycor

`random-xcor`
`random-ycor`

返回海龟 x 或 y 坐标允许范围内的随机浮点数。

海龟水平坐标从 `min-pxcor - 0.5` (含) 到 `max-pxcor + 0.5` (不含); 垂直坐标为 `min-pycor - 0.5` (含) 到 `max-pycor + 0.5` (不含)。

```
ask turtles [
  ;; move each turtle to a random point
  setxy random-xcor random-ycor
```

]

另见 [random-pxcor](#), [random-pycor](#)

read-from-string

read-from-string *string*

将给定的字符串解释为命令中心的输入，返回结果值。结果可能是数值、列表、字符串、布尔值，或特殊值“nobody”。

与[user-input](#)一起使用，可以将用户输入转化为可用形式。

```
show read-from-string "3" + read-from-string "5"
=> 8
show length read-from-string "[1 2 3]"
=> 3
crt read-from-string user-input "Make how many turtles?"
;; the number of turtles input by the user
;; are created
```

reduce

reduce [*reporter*] *list*

使用*reporter*从左到右缩减列表，得到一个单值。含义为，例如`reduce [?1 + ?2] [1 2 3 4]`等价于`((1 + 2) + 3) + 4`。如果列表只有 1 项，返回该项。如果对空列表缩减，则出错。

在*reporter*中使用`?1` 和 `?2` 引用要组合的两个对象。

很难直接说明**reduce**的含义，下面有一些例子，尽管不太实用，但可以让你理解该原语的含义。

```
show reduce [?1 + ?2] [1 2 3]
=> 6
show reduce [?1 - ?2] [1 2 3]
=> -4
show reduce [?2 - ?1] [1 2 3]
=> 2
show reduce [?1] [1 2 3]
=> 1
show reduce [?2] [1 2 3]
=> 3
show reduce [sentence ?1 ?2] [[1 2] [3 [4]] 5]
```

```
=> [1 2 3 [4] 5]
show reduce [fput ?2 ?1] (fput [] [1 2 3 4 5])
=> [5 4 3 2 1]
```

还有一些更实用的例子：

```
;; find the longest string in a list
to-report longest-string [strings]
  report reduce
    [ifelse-value (length ?1 >= length ?2) [?1] [?2]]
    strings
  end

  show longest-string ["hi" "there" "!"]
=> "there"

;; count the number of occurrences of an item in a list
to-report occurrences [x the-list]
  report reduce
    [ifelse-value (?2 = x) [?1 + 1] [?1]] (fput 0 the-list)
  end

  show occurrences 1 [1 2 1 3 1 2 3 1 1 4 5 1]
=> 6

;; evaluate the polynomial, with given coefficients, at x
to-report evaluate-polynomial [coefficients x]
  report reduce [(x * ?1) + ?2] coefficients
end

;; evaluate  $3x^2 + 2x + 1$  at  $x = 4$ 
show evaluate-polynomial [3 2 1] 4
=> 57
```

remainder

remainder *number1* *number2*

返回 *number1* 除以 *number2* 的余数。等价于下面的代码：

```
number1 - (int (number1 / number2)) * number2
show remainder 62 5
=> 2
show remainder -8 3
```

```
=> -2
```

另见 [mod](#)。对整数 mod 和 remainder 一样，但对负数不一样。

remove

```
remove item list
remove string1 string2
```

对列表，返回去除 *list* 中所有 *item* 实例的列表拷贝。

对字符串，返回从 *string2* 中去除了所有 *string1* 子串的字符串拷贝。

```
set mylist [2 7 4 7 "Bob"]
set mylist remove 7 mylist
;; mylist is now [2 4 "Bob"]
show remove "to" "phototonic"
=> "phonic"
```

remove-duplicates

```
remove-duplicates list
```

对列表去除所有重复项，但每项的第一个位置保留。

```
set mylist [2 7 4 7 "Bob" 7]
set mylist remove-duplicates mylist
;; mylist is now [2 7 4 "Bob"]
```

remove-item

```
remove-item index list
remove-item index string
```

对列表，去除 *list* 给定索引项，返回列表的拷贝。

对字符串，去除 *string2* 给定索引处的字符，返回字符串的拷贝。

注意索引从 0 开始。(第一项索引是 0)

```
set mylist [2 7 4 7 "Bob"]
set mylist remove-item 2 mylist
;; mylist is now [2 7 7 "Bob"]
```

```
show remove-item 2 "string"  
=> "sting"
```

repeat

repeat *number* [*commands*]

运行 *commands* 共 *number* 次。

```
pd repeat 36 [ fd 1 rt 10 ]  
;; the turtle draws a circle
```

replace-item

replace-item *index list value*
replace-item *index string1 string2*

对列表，替换索引指定的项。索引从 0 开始（列表的第 6 项索引为 5）。注意“replace-item”与“set”一起改变列表。

对字符串作用相似，只是 *string1* 给定索引处的字符被去除，而 *string2* 插进来。

```
show replace-item 2 [2 7 4 5] 15  
=> [2 7 15 5]  
show replace-item 1 "cat" "are"  
=> "caret"
```

report

report *value*

立即从当前 to-report 例程退出，返回 *value* 作为例程的结果。report 和 to-report 总是一起使用。在[to-report](#) 讨论了它们的使用。

reset-perspective

rp

reset-perspective

观察者停止观察、跟随、乘骑任何海龟（或瓦片）。（如果观察者没有观察、跟随、乘骑任何主体，则什么也不发生）。在 3 维视图，观察者还要返回默认位置（在原点上方，向下直视）

另见 [follow](#), [ride](#), [watch](#) 。

reset-ticks

reset-ticks



将时钟计数器重设为 0。

另见 [tick](#), [ticks](#), [tick-advance](#)

reset-timer

reset-timer

将计时器重设为 0。另见 [timer](#) 。

注意计时器与时钟计数器不同。计时器用秒计量逝去的真实时间，而时钟计数器用滴答计量逝去的模型时间。

reverse

reverse *list*

reverse *string*

返回颠倒过来的列表拷贝。

```
show mylist
;; mylist is [2 7 4 "Bob"]
set mylist reverse mylist
;; mylist now is ["Bob" 4 7 2]
show reverse "live"
=> "evil"
```

rgb

rgb *red green blue*

当三个数描述 RGB 颜色时，返回一个 RGB 列表。数值范围为 0–255。

另见 [hsb](#)

ride

`ride turtle`



将视角设到 *turtle*。

每次 *turtle* 移动，观察者也移动。这样在 2 维视图，海龟总是停留在视图的中心。在 3 维视图，好像是从海龟的眼睛看世界。如果海龟死亡，视角设为默认。

另见 [reset-perspective](#), [watch](#), [follow](#), [subject](#)

ride-me

`ride-me`



请求观察者乘骑调用海龟。

另见 [ride](#)

right

`right number`



海龟右转 *number* 度。（如果 *number* 为负，则左转）

round

`round number`

返回接近 *number* 的整数。

如果小数部分恰好是 0.5，则正向取整（rounded in the **positive** direction）。

注意正向取整与其他软件程序可能会不一致。（特别是与 StarLogoT 不一样，StarLogoT 总是舍入到最接近的偶数）。这样做与 NetLogo 里海龟坐标和瓦片坐标的关系匹配。例如，如果海龟 x 坐标是 -4.5，则它在 x 坐标为 -4 和 -5 的两个瓦片边界处，我们认为海龟在 -4 的瓦片上，因为我们正向取整。

```
show round 4.2  
=> 4  
show round 4.5  
=> 5  
show round -4.5  
=> -4
```

run

run *string*

将给定字符串解释为一个或多个命令序列，然后运行。

代码在主体的当前上下文中运行，即能访问局部变量、“myself”等。

代码必须先被编译，这要花费时间，编译后的二进制代码被 NetLogo 缓存，因此如果重复运行同样的字符串则会快得多。

另见 [runresult](#)

注意不能使用 run 定义或重定义例程。

注意用 run 或 runresult 运行代码要比直接运行这些代码慢很多倍。

runresult

runresult *string*

主体将给定字符串解释为一个报告器，运行，返回结果。

代码在主体的当前上下文中运行，即能访问局部变量、“myself”等。

代码必须先被编译，这要花费时间，编译后的二进制代码被 NetLogo 缓存，因此如果重复运行同样的字符串则会快得多。

另见 [run](#)

注意用 run 或 runresult 运行代码要比直接运行这些代码慢很多倍。

S**scale-color**

scale-color *color number range1 range2*

返回明暗与 *number* 成正比的 *color* 色。

如果 *range1 < range2*, *number* 越大, 颜色越亮。如果 *range1 > range2*, 则相反。

如果 *number < range1*, 则为最暗的 *color* 色。

如果 *number > range1*, 则为最亮的 *color* 色。

注意: 对明暗无关的颜色, 例如 green 和 green + 2 一样, 使用同样的色谱。

```
ask turtles [ set color scale-color red age 0 50 ]
;; colors each turtle a shade of red proportional
;; to its value for the age variable
```

self

self


返回本海龟或瓦片。

“self” 与 “myself” 大不相同。“self” 很简单, 就是指我 (“me”)。“myself” 是指“请求我做目前正在做的事情的海龟或瓦片”。

另见 [myself](#)

; (semicolon)

; *comments*

本行分号后的内容被忽略。分号用来为程序增加“注释” - 为人类读者解释代码。可以增加多余的分号增加美观性。

NetLogo 的 Edit 菜单有对整块代码增加、去除注释的菜单项。

sentence

se

```
sentence value1 value2
(sentence value1 ...)
```

根据输入值创建列表。如果某个输入值是列表，该列表的项直接包含在结果列表中，而不是作为嵌套列表。下面的例子清楚的说明了这一点：

```
show sentence 1 2
=> [1 2]
show sentence [1 2] 3
=> [1 2 3]
show sentence 1 [2 3]
=> [1 2 3]
show sentence [1 2] [3 4]
=> [1 2 3 4]
show sentence [[1 2]] [[3 4]]
=> [[1 2] [3 4]]
show (sentence [1 2] 3 [4 5] (3 + 3) 7)
=> [1 2 3 4 5 6 7]
```

set

```
set variable value
```

将变量 *variable* 设为给定值。

变量包括：

- 使用"globals" 声明的全局变量
- 与滑动条、开关、选择器、输入框关联的全局变量
- 属于调用主体的变量
- 如果调用主体是海龟，海龟下面的瓦片变量
- 用[let](#) 创建的局部变量
- 当前例程的输入
- 特殊局部变量 (? , ?1 , ?2...).

set-current-directory

```
set-current-directory string
```

设置[file-delete](#), [file-exists?](#), [file-open](#)使用的当前目录。

如果上述命令给出了绝对文件路径，则不使用当前目录。创建新模型时，当前目录默认为用户目录（user's home directory）。打开模型时当前目录为模型所在目录。

注意在 Windows 中，字符串中的反斜线需要加上另一个反斜线作为转义符，如“C:\\\\”。

当前目录的改变是临时的，不会存在模型里。

注意：在 applet 里，该命令没有效果。因为 applet 只能从服务器的模型所在目录读取文件。

```
set-current-directory "C:\\NetLogo"
;; Assume it is a Windows Machine
file-open "my-file.txt"
;; Opens file "C:\\NetLogo\\my-file.txt"
```

set-current-plot

`set-current-plot plotname`

将给定名字的绘图设置为当前绘图。后面的绘图命令将影响当前绘图。

set-current-plot-pen

`set-current-plot-pen penname`

将指定名字的画笔设为当前绘图的当前画笔。如果当前绘图没有这个画笔，则出现运行错误。

set-default-shape

`set-default-shape turtles string`

`set-default-shape breed string`



对所有海龟或特定种类的海龟设定默认初始图形。当海龟创建或改变种类时，海龟被设置为给定图形。

该命令不会影响已存在的海龟，只对以后创建的海龟有影响。

指定的种类必须是海龟或是由[breed](#)关键字定义的种类，指定的字符串必须是当前定义的图形的名字。

在新模型里，所有海龟的默认图形是“default”。

注意指定默认图形，不会妨碍你以后改变单个海龟的图形，海龟不必一直使用所属种类的默认图形。

```
create-turtles 1 ;; new turtle's shape is "default"
create-cats 1      ;; new turtle's shape is "default"

set-default-shape turtles "circle"
create-turtles 1 ;; new turtle's shape is "circle"
create-cats 1      ;; new turtle's shape is "circle"

set-default-shape cats "cat"
set-default-shape dogs "dog"
create-cats 1      ;; new turtle's shape is "cat"
ask cats [ set breed dogs ]
;; all cats become dogs, and automatically
;; change their shape to "dog"
```

另见 [shape](#)

set-histogram-num-bars

`set-histogram-num-bars number`

设置当前画笔的画图间隔，这样给定绘图的当前 x 范围后，如果调用直方图命令则会画出 *number* 个条形。

另见 [histogram](#)

set-line-thickness

`set-line-thickness number`



指定海龟图形的线宽和轮廓。

默认值是 0，总是产生 1 个像素宽的线。

非 0 值使用瓦片宽度单位。例如宽度为 1，则画出 1 个瓦片宽的线。（一般使用较小的值，如 0.5, 0.2）

线至少 1 个像素宽。

这是一个实验性命令，以后版本可能会修改。

set-plot-pen-color

set-plot-pen-color *number*

将当前绘图画笔的颜色设为 *number*。

set-plot-pen-interval

set-plot-pen-interval *number*

告诉当前画笔每次使用绘图命令时，在 x 方向移动 *number*。（画笔间隔也影响直方图命令的行为）

set-plot-pen-mode

set-plot-pen-mode *number*

设置当前画笔的绘制模式为 *number*。允许的画笔模式有：

- 0 (线)：画笔画线将两点相连。
- 1 (条形)：画笔画一个条形，宽度为 plot-pen-interval，点作为条形的左上角（如果是负数，则为左下角）
- 2 (点)：画笔只画点，点之间不连。

新画笔的默认模式是 0 (line mode)。

set-plot-x-range

set-plot-y-range

set-plot-x-range *min max*

set-plot-y-range *min max*

设置当前绘图的 x、y 轴最小最大值。

改变是临时的，不会存在模型中。当绘图清除时，坐标范围恢复到绘图编辑对话框中设置的默认值。

setxy**setxy** *x y*海龟将它的坐标设置为 *x, y*。该命令等价于 `set xcor x set ycor y`, 不过只使用一个时间步, 而不是两个时间步。如果 *x, y* 超出世界范围, NetLogo 抛出运行错误。

```
setxy 0 0
;; turtle moves to the middle of the center patch
setxy random-xcor random-ycor
;; turtle moves to a random point
setxy random-pxcor random-pycor
;; turtle moves to the center of a random patch
```

另见 [move-to](#)**shade-of?****shade-of?** *color1 color2*如果两个颜色属于同一色系 (shades of one another), 返回 `true`, 否则返回 `false`。

```
show shade-of? blue red
=> false
show shade-of? blue (blue + 1)
=> true
show shade-of? gray white
=> true
```

shape**shape**这是一个内置海龟变量或链变量, 保存海龟或链当前图形的名字。设置该变量改变他们的形状。除非使用 [set-default-shape](#) 指定不同的形状, 否则新海龟或链的形状为 "default"。

例子:

```
ask turtles [ set shape "wolf" ]
;; assumes you have made a "wolf"
;; shape in NetLogo's Turtle Shapes Editor
ask links [ set shape "link 1" ]
;; assumes you have made a "link 1" shape in
;; the Link Shapes Editor
```

另见 [set-default-shape](#), [shapes](#)

shapes

shapes

返回模型中所包含的所有图形名称字符串的列表。

使用[Shapes Editor](#)能够创建新图形，也能从图形库或其他模型导入。

```
show shapes
=> ["default" "airplane" "arrow" "box" "bug" ...
ask turtles [ set shape one-of shapes ]
```

show

show *value*

在命令中心显示*value*，前面加上调用主体，后跟回车。（调用主体用来帮你跟踪哪个主体产生了哪行输出）。另外，与[write](#)相似，所有字符串有引号。

另见 [print](#), [type](#), [write](#).

另见 [output-show](#)

show-turtle

st

show-turtle



海龟再次变得可见。

注意：该命令等价于设置海龟变量“hidden?”为 `false`。

另见[hide-turtle](#)

show-link

`show-link`



链再次变得可见。

注意：该命令等价于设置链变量“hidden?”为 `false`。

另见 [hide-link](#)

shuffle

`shuffle list`

返回一个包含原列表中的所有项，但各项随机排序的新列表。

```
show shuffle [1 2 3 4 5]
=> [5 2 4 1 3]
show shuffle [1 2 3 4 5]
=> [1 3 5 2 4]
```

sin

`sin number`

返回给定角的正弦值，角度单位为度。

```
show sin 270
=> -1
```

size

`size`



这是一个内置海龟变量，保存海龟的外观大小。默认是 1，意味着海龟大小与瓦片一样。通过设置它改变海龟大小。

sort

```
sort list-of-numbers
sort list-of-strings
sort agentset
```

如果输入是数值列表或字符串列表，返回包含原列表所有项，但按升序排列的新列表（数值或字母顺序）。

非数值或非字符型列表项被忽略。（如果输入列表不包括数值或字符串，结果是空列表）

如果输入是主体集合或主体列表，返回主体列表（不会是主体集合）。如果主体是海龟，则他们根据 who number 升序排列。如果是瓦片，则从左到右，从上到下排列。

```
show sort [3 1 4 2]
=> [1 2 3 4]
let n 0
foreach sort patches [
  ask ? [
    set plabel n
    set n n + 1
  ]
]
;; patches are labeled with numbers in left-to-right,
;; top-to-bottom order
```

sort-by

```
sort-by [reporter] list
sort-by [reporter] agentset
```

如果输入是列表，返回一个包含原列表所有项，但由布尔型 *reporter* 定义顺序的新列表。

在*reporter*，使用?1 和 ?2 引用所比较的两个对象，如果?1 严格的在 ?2 之前，则返回 true，否则返回false。

如果输入是主体集合或主体列表，返回主体列表（不会是主体集合）。

排序是稳定的，即相等项的顺序不会被干扰。（The sort is stable, that is, the order of items considered equal by the reporter is not disturbed）

```
show sort-by [?1 < ?2] [3 1 4 2]
=> [1 2 3 4]
show sort-by [?1 > ?2] [3 1 4 2]
```

```
=> [4 3 2 1]
show sort-by [length ?1 < length ?2] ["Grumpy" "Doc" "Happy"]
=> ["Doc" "Happy" "Grumpy"]
foreach sort-by [[size] of ?1 < [size] of ?2] turtles
  [ ask ? [ do-something ] ]
;; turtles run "do-something" one at a time, in
;; ascending order by size
```

sprout

sprout-<breeds>

```
sprout number [ commands ]
sprout-<breeds> number [ commands ]
```



在当前瓦片上创建 *number* 个新海龟。新海龟的方向是随机整数，颜色从 14 个主色中随机产生。海龟立即运行 *commands*，如果要给新海龟不同的颜色、方向等就比较有用。（新海龟是一次全部产生出来，然后以随机顺序每次运行 1 个）

如果使用 sprout-<*breeds*>形式，则新海龟属于给定的种类。

```
sprout 5
sprout-wolves 10
sprout 1 [ set color red ]
sprout-sheep 1 [ set color black ]
```

注意：当运行命令时，其他主体不允许运行任何代码（就像使用without-interruption命令一样）。这就保证如果使用了ask-concurrent，新海龟在完全初始化之前，不能与任何其他主体交互。

另见 [create-turtles](#), [hatch](#)

sqrt

```
sqrt number
```

返回 *number* 的方根。

stamp

```
stamp
```



调用海龟或链在画图层的当前位置留下一幅主体图形。

注意：stamp 留下的图像在不同的计算机上可能不是逐个像素完全对应。

stamp-erase

`stamp-erase`



调用海龟或链在画图层里将它的图形范围内所有像素清除。

注意：stamp 留下的图像在不同的计算机上可能不是逐个像素完全对应。

standard-deviation

`standard-deviation list`

返回列表 *list* 里所有数值的无偏统计量-标准差。非数值型项忽略。

```
show standard-deviation [1 2 3 4 5 6]
=> 1.8708286933869707
show standard-deviation [energy] of turtles
;; prints the standard deviation of the variable "energy"
;; from all the turtles
```

startup

`startup`



这是用户定义的例程。如果该例程存在的话，模型第一次加载时就调用该例程。

```
to startup
  setup
end
```

stop

`stop`

调用主体立即从闭合例程、ask、或类似 ask 的结构 (crt, hatch, sprout, without-interruption) 中退出。当前例程停止，而不是主体的所有运行都停止。

```
if not any? turtles [ stop ]
;; exits if there are no more turtles
```

注意：可以使用 `stop` 停止一个永久性按钮。如果永久性按钮直接调用例程，当例程停止时，按钮停止。（在海龟或瓦片型永久性按钮中，直到所有海龟或瓦片停止后，按钮才停止 – 单个海龟或瓦片没有能力停止整个按钮）

subject

`subject`

返回观察者正在查看、跟随、乘骑的海龟（或瓦片）。如果没有这样的海龟（或瓦片）则返回[nobody](#)。

另见 [watch](#), [follow](#), [ride](#)。

sublist

substring

```
sublist list position1 position2
substring string position1 position2
```

返回给定列表或字符串的一部分，范围从第一个位置（包括）到第二个位置（不包括）。

注意：位置编号从 0 开始，而不是从 1 开始。

```
show sublist [99 88 77 66] 1 3
=> [88 77]
show substring "apartment" 1 5
=> "part"
```

subtract-headings

`subtract-headings heading1 heading2`

计算给定的两个方向之差，即 `heading2` 旋转为 `heading1` 所需的最小角度。结果为正表示顺时针旋转，为负表示逆时针旋转。结果总在 -180 到 180，不会恰好为 -180。

注意简单的对两个方向使用减法不会奏效。减法总是对应顺时针旋转，有时逆时针旋转角度更小。例如，5 度和 355 度之间的角度差是 10 度，而不是 -350 度。

```
show subtract-headings 80 60
```

```
=> 20
show subtract-headings 60 80
=> -20
show subtract-headings 5 355
=> 10
show subtract-headings 355 5
=> -10
show subtract-headings 180 0
=> 180
show subtract-headings 0 180
=> 180
```

sum**sum** *list*

返回列表的各项之和。

```
show sum [energy] of turtles
;; prints the total of the variable "energy"
;; from all the turtles
```

T**tan****tan** *number*

返回给定角的正切值，角的单位为度。

thickness**thickness**

这是一个内置链变量，保存链的表现尺寸大小，该尺寸是相对于瓦片尺寸的一个数值。默认是 0，表示不管瓦片大小，链的宽度总是 1 像素。设置该变量改变链的粗细。

tick

`tick`



时钟计数器前进 1。

另见 [ticks](#), [tick-advance](#), [reset-ticks](#)

tick-advance

`tick-advance number`



时钟计数器前进 *number*。 输入可以是整数或浮点数（有些模型将时间分割的更细）。输入不能为负。

另见 [tick](#), [ticks](#), [reset-ticks](#)

ticks

`ticks`

返回时钟计数器的当前值。结果总是是一个非负数值。

多数模型使用 tick 命令推进时钟，这样的话 ticks 总是返回整数。如果使用了 tick-advance 命令，则可能返回浮点数。

另见 [tick](#), [tick-advance](#), [reset-ticks](#)

tie

`tie`



将链的 *end1* 和 *end2* 端点捆绑在一起⁷。如果有向链，则 *end1* 为根海龟， *end2* 为叶海龟。根海龟的运动影响叶海龟的位置和方向。如果是无向链，则捆绑是相互的，两个海龟都能作为根海龟或叶海龟，某个海龟的运动将影响另一个海龟的位置和方向。

⁷ 译者注：原文为Ties *end2* and *end2* of the link together，有误。

当根海龟移动时，叶海龟也沿相同的方向移动相同的距离，叶海龟的方向不受影响。根海龟 forward, jump, 以及设置 xcor 或 ycor 坐标都会产生这样的影响。

当根海龟左转或右转时，叶海龟围绕根海龟旋转相同的角度，叶海龟的方向也旋转相同的角度。

如果链死亡，捆绑关系消失。

```
crt 2 [ fd 3 ]
;; creates a link and ties turtle 1 to turtle 0
ask turtle 0 [ create-link-to turtle 1 [ tie ] ]
```

另见 [untie](#)

tie-mode

tie-mode



这是一个内置链变量，保存链当前使用的捆绑模式名称字符串。使用[tie](#) 和 [untie](#) 命令改变链的模式。也可以设置tie-mode 为“free”，实现两个海龟之间的非刚性连接（non-rigid joint）。（细节见编程指南的捆绑部分）。默认情况下，链不捆绑。

另见: [tie](#), [untie](#)

timer

timer

返回自从上次使用[reset-timer](#)命令（或NetLogo启动）以来逝去的秒数。时钟最大的精度是毫秒。（是否能得到这个精度根据系统不同而不同，取决于所用的Java虚拟机）

另见 [reset-timer](#)

注意 timer 与 tick counter 不同。Timer 计量逝去的真实秒数，而 tick counter 计量模型运行的滴答数。

to

```
to procedure-name
to procedure-name [input1 ...]
```

用来开始一个命令例程。

```

to setup
  clear-all
  crt 500
end

to circle [radius]
  crt 100 [ fd radius ]
end

```

to-report

```

to-report procedure-name
to-report procedure-name [input1 ...]

```

用来开始一个报告器例程。

例程体应使用report为例程返回一个值。见[report](#)。

```

to-report average [a b]
  report (a + b) / 2
end

to-report absolute-value [number]
  ifelse number >= 0
    [ report number ]
    [ report (- number) ]
end

to-report first-turtle?
  report who = 0 ;; reports true or false
end

```

towards

towards *agent*



返回从本主体到给定主体的方向。

如果拓扑允许回绕并且回绕距离（穿越世界边缘）更短，towards 将使用回绕路径。

注意：当一个主体使用 towards 计算到自身的方向，或计算与处在同一位置的主体的方向时，会引起运行时间错误。

```
set heading towards turtle 1
;; same as "face turtle 1"
```

另见 [face](#)

towardsxy

towardsxy *x y*



返回从海龟或瓦片到点(*x, y*)的方向。

如果拓扑允许回绕并且回绕距离（穿越世界边缘）更短，`towardsxy` 将使用回绕路径。

注意：计算主体到它所在点的方向将引起运行时间错误。

另见 [facexy](#)

turtle

turtle *number <breed> number*

返回给定*who number*的海龟，如果没有则返回[nobody](#)。对有种类海龟，也可使用单数种类形式引用他们。

```
ask turtle 5 [ set color red ]
;; turtle with who number 5 turns red
```

turtle-set

turtle-set *value1*
(**turtle-set** *value1 value2 ...*)

返回输入参数中所有海龟组成的主体集合。输入可以是单个海龟、海龟主体集合、[nobody](#)、或包含以上任何类型的列表（或嵌套列表）

```
turtle-set self
(turtle-set self turtles-on neighbors)
(turtle-set turtle 0 turtle 2 turtle 9)
(turtle-set frogs mice)
```

另见 [patch-set](#), [link-set](#)

turtles**turtles**

返回包含所有海龟的主体集合。

```
show count turtles
;; prints the number of turtles
```

turtles-at**<breeds>-at**

turtles-at dx dy
<breeds>-at dx dy



返回与调用者距离为(dx, dy)的瓦片上的海龟组成的主体集合。(如果调用者是海龟, 结果可能包含调用者自身)

```
create-turtles 5 [ setxy 2 3 ]
show count [turtles-at 1 1] of patch 1 2
=> 5
```

如果使用了种类名, 则只有该种类的海龟被收集。

turtles-here**<breed>-here**

turtles-here
<breeds>-here

返回位于调用者瓦片上的所有海龟组成的主体集合(如果调用者是海龟, 则也包括它)

```
crt 10
ask turtle 0 [ show count turtles-here ]
=> 10
```

如果使用了种类名, 则只有该种类的海龟被收集。

```
breed [cats cat]
breed [dogs dog]
```

```
create-cats 5
create-dogs 1
ask dogs [ show count cats-here ]
=> 5
```

turtles-on

<breed>-on

```
turtles-on agent
turtles-on agentset
<breed>-on agent
<breed>-on agentset
```

返回所给定的一个或多个瓦片上所有海龟组成的主体集合，或者与给定的海龟站在同一瓦片上的海龟主体集合。

```
ask turtles [
  if not any? turtles-on patch-ahead 1
    [ fd 1 ]
]
ask turtles [
  if not any? turtles-on neighbors [
    die-of-loneliness
  ]
]
```

如果使用了种类名，则只有该种类的海龟被收集。

turtles-own

<breed>-own

```
turtles-own [var1 ...]
<breed>-own [var1 ...]
```

该关键词与 globals, breed, <*breed*>-own, patches-own 一样，只能用在程序首部，位于任何例程定义之前。它定义属于每个海龟的变量。

如果指定了种类而不是海龟，则只有该种类的海龟拥有所列的变量。（多个种类可以有同一个变量）

```
breed [cats cat ]
```

```

breed [dogs dog]
breed [hamsters hamster]
turtles-own [eyes legs]    ;;= applies to all breeds
cats-own [fur kittens]
hamsters-own [fur cage]
dogs-own [hair puppies]

```

另见 [globals](#), [patches-own](#), [breed](#), [⟨breeds⟩-own](#)

type

type *value*

在命令中心显示 *value*, 不跟回车 (与 [print](#) 和 [show](#) 不同)。由于没有回车, 你可以在一行中显示多项。

与 [show](#) 不同, 在 *value* 前面不显示调用主体。

```

type 3 type " " print 4
=> 3 4

```

另见 [print](#), [show](#), [write](#)

另见 [output-type](#)

U

undirected-link-breed

undirected-link-breed [*<link-breeds>* *<link-breed>*]

该关键词与 [globals](#), [breed](#) 一样, 只能用在程序首部, 位于任何例程定义之前。它定义一个无向链种类。特定种类的链或者都是有向的, 或者都是无向的。第一个输入项定义与该类链相关的主体集合名称, 第二个输入项定义该种类成员的名称。

给定种类链的任何一个链:

- 是由种类链名命名的主体集合的一部分
- 有内置变量 [breed](#) 设为该主体集合
- 由关键词决定是有向的还是无向的

大多数情况下主体集合与 [ask](#) 一起使用, 向特定种类链发出命令。

```

undirected-link-breed [streets street]
undirected-link-breed [highways highway]
to setup
  clear-all
  crt 2
  ask turtle 0 [ create-street-with turtle 1 ]
  ask turtle 0 [ create-highway-with turtle 1 ]
end

ask turtle 0 [ show sort my-links ]
;; prints [(street 0 1) (highway 0 1)]

```

另见 [breed](#), [directed-link-breed](#)

untie

`untie`



如果已经被捆绑，则该命令将end2 从end1 松开（设置`tie-mode`为“none”）。如果是无向链，end1 也从end2 松开。它不会移除两个海龟之间的链。

另见 [tie](#)

细节参见编程指南的捆绑部分 ([Tie](#)) 。

uphill

uphill4

`uphill patch-variable`

`uphill14 patch-variable`



海龟移动到 `patch-variable` 最大的那个相邻瓦片上。如果没有哪个相邻瓦片变量比当前瓦片大，则保持不动。如果有几个瓦片有相同的最大值，则随机选择一个。非数值型值忽略。

`uphill` 考虑 8 个相邻瓦片，而 `uphill14` 考虑四个相邻瓦片。

等价于下面的代码（假设变量为数值型）

```

move-to patch-here ;; go to patch center
let p max-one-of neighbors [patch-variable] ;; or neighbors4
if [patch-variable] of p > patch-variable [

```

```
face p  
move-to p  
]
```

注意海龟总是停在瓦片中心，方向是 45 (uphill) 或 90 (uphill4) 的倍数。

另见[uphill](#), [uphill4](#)

user-directory

user-directory

打开一个对话框，让用户选择一个已存在的目录。

返回绝对路径字符串。如果用户取消选择，则返回 false。

```
set-current-directory user-directory  
;; Assumes the user will choose a directory
```

user-file

user-file

打开一个对话框，让用户选择一个已存在的文件。

返回绝对路径文件名字符串。如果用户取消选择，则返回 false。

```
file-open user-file  
;; Assumes the user will choose a file
```

user-new-file

user-new-file

打开一个对话框，让用户选择一个位置，并为新文件命名。返回绝对路径文件名字符串。如果用户取消选择，则返回 false。

```
file-open user-new-file  
;; Assumes the user will choose a file
```

注意该报告器不会实际创建文件，正常情况下你使用 file-open 创建文件，就像例子中那样。

如果用户选择了已有文件，会询问是否要替换，但该报告器不会实际替换文件，应使用 file-delete 实现替换。

user-input

user-input *value*

返回对话框中标题为 *value* 的输入域用户所输入的字符串。

value 可以是任何类型，不过一般是字符串。

```
show user-input "What is your name?"
```

user-message

user-message *value*

打开一个对话框，显示消息 *value*。

value 可以是任何类型，不过一般是字符串。

```
user-message (word "There are " count turtles " turtles.")
```

user-one-of

user-one-of *value* *list-of-choices*

打开一个对话框，显示消息 *value*, *list-of-choices* 显示为弹出菜单，让用户选择。

返回用户选择的 *list-of-choices* 的项。

value 可以是任何类型，不过一般是字符串。

```
if "yes" = user-one-of? "Set up the model?" ["yes" "no"]
  [ setup ]
```

user-yes-or-no?

user-yes-or-no? *value*

根据用户对 *value* 的响应，返回 true 或 false。

value 可以是任何类型，不过一般是字符串。

```
if user-yes-or-no? "Set up the model?"  
[ setup ]
```

V

variance

variance *list*

返回列表中所有数值的样本方差。忽略非数值型项。

样本方差是各个数值与均值之差的平方和，除以数值项数减 1。

```
show variance [2 7 4 3 5]  
=> 3.7
```

W

wait

wait *number*

等待给定的秒数。（不必是整数，可以指定分数秒）。注意不能期望完全准确，主体等待的时间不会小于给定值，但可能稍微大于给定值。

```
repeat 10 [ fd 1 wait 0.5 ]
```

另见[every](#)

watch

watch *agent*



给 *agent* 打上聚光灯。在 3 维视图，观察者自动旋转，始终面对该主体。

另见 [follow](#), [subject](#), [reset-perspective](#), [watch-me](#)

watch-me

`watch-me`



请求观察者查看调用主体。

另见 [watch](#)

while

`while [reporter] [commands]`

如果 *reporter* 返回 `false`, 退出循环, 否则重复运行 *commands*。

对不同的主体, *reporter* 可能返回不同的值, 因此不同主体运行 *commands* 的次数可能不同。

```
while [any? other turtles-here]
  [ fd 1 ]
;; turtle moves until it finds a patch that has
;; no other turtles on it
```

who

`who`



这是一个内置海龟变量, 保存海龟的“*who number*”或 ID 号, 这是一个大于等于 0 的整数。不能设置该变量, 海龟的“*who number*”不会改变。

Who numbers 从 0 开始。死亡海龟的号码不会分配给新海龟, 除非使用[clear-turtles](#) 或 [clear-all](#) 命令, 使得重新从 0 开始编号。

例子:

```
show [who] of turtles with [color = red]
;; prints a list of the who numbers of all red turtles
;; in the Command Center, in random order
crt 100
[ ifelse who < 50
  [ set color red ]
  [ set color blue ] ]
;; turtles 0 through 49 are red, turtles 50
```

```
;; through 99 are blue
```

可以使用海龟报告器返回给定 who number 的海龟。

另见 [turtle](#)

with

agentset with [reporter]

有两个输入参数，左边是一个主体集合（一般是“turtles”或“patches”），右边是一个布尔型报告器。返回一个新的主体集合，集合中仅包含那些使报告器返回 true 的主体，换句话说，主体满足给定的条件。

```
show count patches with [pcolor = red]
;; prints the number of red patches
```

<breed>-with

link-with

<breed>-with *turtle*
link-with *turtle*


返回 *turtle* 和调用者之间的链。如果没有链则返回 nobody。

```
crt 2
ask turtle 0 [
  create-link-with turtle 1
  show link-with turtle 1 ;; prints link 0 1
]
```

with-max

agentset with-max [reporter]

有两个输入参数，左边是一个主体集合（一般是“turtles”或“patches”），右边是一个报告器。返回一个新的主体集合，集合中仅包含那些使报告器返回最大值的主体。

```
show count patches with-max [pxcor]
;; prints the number of patches on the right edge
```

另见 [max-one-of](#), [max-n-of](#)

with-min

agentset **with-min** [*reporter*]

有两个输入参数，左边是一个主体集合（一般是“turtles”或“patches”），右边是一个报告器。返回一个新的主体集合，集合中仅包含那些使报告器返回最小值的主体。

```
show count patches with-min [pycor]
;; prints the number of patches on the bottom edge
```

另见 [min-one-of](#), [min-n-of](#)

with-local-randomness

with-local-randomness [*commands*]

该处命令的运行不影响后面的随机事件。当要执行额外的操作（例如输出）而不想对模型输出产生影响时，使用这个命令。

例子：

```
;; Run #1:
random-seed 50 setup repeat 10 [ go ]
;; Run #2:
random-seed 50 setup
with-local-randomness [ watch one-of turtles ]
repeat 10 [ go ]
```

因为在without-local-randomness使用one-of，两次运行是相同的。

该命令的工作原理是：在该命令之前记住随机数发生器的状态，运行后再恢复。（如果要用随机数发生器新的状态运行该命令，在命令开始处使用random-seed new-seed）

下面的例子演示了在命令运行前后随机数发生器的状态一样。

```
random-seed 10
with-local-randomness [ print n-values 10 [random 10] ]
;; prints [8 9 8 4 2 4 5 4 7 9]
print n-values 10 [random 10]
;; prints [8 9 8 4 2 4 5 4 7 9]
```

without-interruption

`without-interruption [commands]`

主体运行块中的命令时不允许其他主体使用ask-concurrent打断。也就是说，其他主体被挂起（“on hold”），直到块中的命令执行完。

注意：这个命令只有与ask-concurrent一起使用才有用。在以前的NetLogo版本中，经常需要该命令，但在NetLogo 4.0中，只有使用ask-concurrent时才可能需要它。

另见 [ask-concurrent](#)

word

`word value1 value2
(word value1 ...)`

将输入项连在一起，做为字符串返回。

```
show word "tur" "tle"
=> "turtle"
word "a" 6
=> "a6"
set directory "c:\\\\foo\\\\fish\\\\"
show word directory "bar.txt"
=> "c:\\foo\\fish\\bar.txt"
show word [1 54 8] "fishy"
=> "[1 54 8]fishy"
show (word 3)
=> "3"
show (word "a" "b" "c" 1 23)
=> "abc123"
```

world-width

world-height

`world-width
world-height`

返回 NetLogo 世界的总宽度和总高度。

宽度等于 `max-pxcor - min-pxcor + 1`，高度等于 `max-pycor - min-pycor + 1`。

另见[max-pxcor](#), [max-pycor](#), [min-pxcor](#), [min-pycor](#)

wrap-color

wrap-color *number*

`wrap-color` 检查 *number* 是否在 NetLogo 颜色范围 0–140(不包括 140)内，如果不在，将数值回绕到 0–140 之内。

回绕的方法是对 *number* 重复加上或减去 140，直到落在 0–140 范围之内。（当将超过范围的数值赋给海龟的 color 或瓦片的 pcolor 时，自动按这样的方式回绕）

```
show wrap-color 150
=> 10
show wrap-color -10
=> 130
```

write

write *value*

输出 *value* 到命令中心，该值可以是数值、字符串、列表、布尔型或 nobody，后面不加回车（与[print](#) 和 [show](#)不同）。

与[show](#)不同，前面不显示调用主体。输出的字符串包含引号，前面有空格。

```
write "hello world"
=> "hello world"
```

另见 [print](#), [show](#), [type](#)

另见 [output-write](#)

X

xcor

xcor


这是一个内置海龟变量，保存海龟当前的 x 坐标。设置该变量改变海龟的位置。

该变量总是大于等于($\text{min-pxcor} - 0.5$)，严格小于($\text{max-pxcor} + 0.5$)。

另见 [setxy](#), [ycor](#), [pxcor](#), [pycor](#)

xor

boolean1 xor boolean2

当 *boolean1* 或 *boolean2* 其中之一为 true，返回 true。二者同时为 true 不返回 true。

```
if (pxcor > 0) xor (pycor > 0)
  [ set pcolor blue ]
;; upper-left and lower-right quadrants turn blue
```

Y

ycor

ycor



这是一个内置海龟变量，保存海龟当前的 y 坐标。设置该变量改变海龟的位置。

该变量总是大于等于($\text{min-pycor} - 0.5$)，严格小于($\text{max-pycor} + 0.5$)。

另见 [setxy](#), [xcor](#), [pxcor](#), [pycor](#)

?

? , ?1, ?2, ?3, ...

? , ?1, ?2, ?3, ...

这些是特殊的局部变量，为某些原语保存报告器或命令块的当前输入。（例如，[foreach](#) 或 [map](#) 正在访问的列表当前项）

? 总是等价于 ?1。

不能设置这些变量，并且它们仅在某些原语中使用，目前这些原语有[foreach](#), [map](#), [reduce](#), [filter](#), [sort-by](#), [n-values](#)，可以查看这些原语条目中的例子。

Java 小程序 (Applets)

单个模型可以作为 Java applets 在网页浏览器中运行。

制作和显示 Applets

在文件菜单中选择 Save As Applet 可以创建 applets。如果模型的当前状态未保存，则先提示你保存，也会提示你有一个 html 文件要保存。

要运行 applets，模型文件、html 文件和 NetLogoLite.jar 必须在同一目录。(NetLogo 的安装目录里有 NetLogoLite.jar，你可以从这里复制)

Applets可以在web服务器上读写文件。如果applet需要额外的文件，从文件读数据、输入图像等，这些文件也需要上传到服务器。这些文件应该出现在相同的目录配置中，就像出现在你的计算机上的一样。Applets不能读写用户计算机上的文件，只能读取web服务器的文件。Applets不能浏览web服务器或用户计算机，意味着user-file 和 user-new-file 在 applet里没有作用。要运行模型，你的模型文件和NetLogoLite.jar必须对web服务器用户是可读的。

在一些系统里，在上传到服务器之前，可以在本地机上测试 applet。当然不是所有系统都能这样做，因此如果不能在你的硬盘上运行，试着上传到 web 服务器上。

不需要在页面上包含 html 文件中的所有事情。如果需要的话，可以只将 HTML 代码中的 <applet> 和 </applet> 部分粘贴到任何 HTML 文件中。在一个页面中放置多个<applet>标记也是可以的。

如果 NetLogoLite.jar 和你的模型在不同的目录，必须修改 HTML 中的 archive= 和 value= 行，指向他们的实际位置。（例如，如果在一个 web 服务器上多个 applets 在不同的目录里，可能希望只有一个 NetLogoLite.jar 拷贝，因此需要修改 HTML 中的 archive= 行指向该拷贝。这样节约磁盘空间，也节省用户的下载时间）

Java 需求

获得所需的版本

目前的 NetLogo 需要浏览器支持 Java 1.4.1 或更高版本。下面是获得所需版本的 Java 的方法：

- 如果是 Windows 98 或更新的 Windows，需要从 http://www.java.com/en/download/windows_manual.jsp 下载 Java 浏览器插件
- 如果是 Mac OS X，需要 OS X 10.2.6 或更高版本。如果使用 OS X 10.2，你还需要 Java 1.4.1 更新包 1，可以通过软件更新得到。OS X 10.3 或更高版本已经有了适当的 Java 版本。必须有支持 Java 1.4 的浏览器，Internet Explorer 不行，Safari 可以。

- 在 Windows 95, MacOS 8, or MacOS 9, 不再支持在 web 运行模型。必须下载 NetLogo 1.3.1 运行模型。
- 如果使用 Linux 或其他 Unix, 需要 1.4.1 以上的 Sun Java Runtime Environment。可以从 <http://www.java.com/> 下载。检查浏览器的主页获得安装 Java 插件的信息。

如果认为浏览器和插件都可以, 但仍然不能运行, 检查浏览器的设置, 看看 Java 是否激活。

要想了解你的 Java 版本和获得合适的版本, 查看 <http://www.javatest.org/>

增加可用内存

有些 NetLogo 需要的内存比正常运行浏览器时要多一些。如果有大量的主体, 可能就会这样。在 Windows, 在 Java 控制板的 applet runtime settings 增加可用内存 (“堆”)。

Mac 用户注意 Mac OS X 10.2 和 10.3 对 Java applets 有很低的内存限制, 64M。过去 Mac OS X 10.4 也是同样的限制, 但最近的 Java 更新增加到 96M。使用软件更新得到 Java 更新。Mac OS X 10.5 估计有更高的限制值。

如果浏览器使用随 Sun JDK 或 JRE 提供的插件, 启动 Java Plug-In Control Panel 的步骤见 [here](#)。在控制板的高级选项页为 Java Runtime Parameters 增加: “-Xmx1024M”。

扩展

只需将扩展包上传到模型所在目录, 许多扩展包就可以在 applets 里使用。

那些需要额外 jar 的扩展不能在 applets 使用, sound 和 gogo 扩展就属于这一类。

已知问题

- 需要额外 jar 的扩展包不能在 applets 使用
- applets 不能使用 3 维视图。
- applets 没有使用代码产生器(意味着模型运行较慢)
- 返回定制错误消息的 web 服务器可能引起 java 例外。见 [FAQ](#)。

图形编辑器指南（Shapes Editor Guide）

海龟和链图形编辑器用来创建和保存海龟和链图案。NetLogo 使用完全可缩放、可旋转的矢量图形，可以使用基本几何形状组合成图案，图案可以用任何大小和角度显示在屏幕上。

开始

在 Tools 菜单中选择 **Turtle Shapes Editor** 或 **Link Shapes Editor** 开始制作图形。在打开的新窗口里，当前模型中的所有图形都会列出来，第一项是默认图形 *default*。使用图形编辑器可以创建新图形、编辑图形、从其他模型中导入图形。也可以从由已有图形组成的图形库里导入图形。

导入图形（Importing Shapes）

新模型开始时只包含一小部分常用的核心图形。使用 **Import from library...** 按钮获得更多的海龟图形，出现对话框，选择一个或多个图形导入模型之中。选择所需图形，按下 **Import** 按钮。

类似的，也可使用 **Import from model...** 从其他模型导入图形。

默认图形（Default shapes）

下面是默认情况下每个新模型包含的海龟图形：



第一行: default, airplane, arrow, box, bug, butterfly

第二行: car, circle, circle 2, cow, cylinder, dot

第三行: face happy, face neutral, face sad, fish, flag, flower

第四行: house, leaf, line, line half, pentagon, person

第五行: plant, square, square 2, star, target, tree

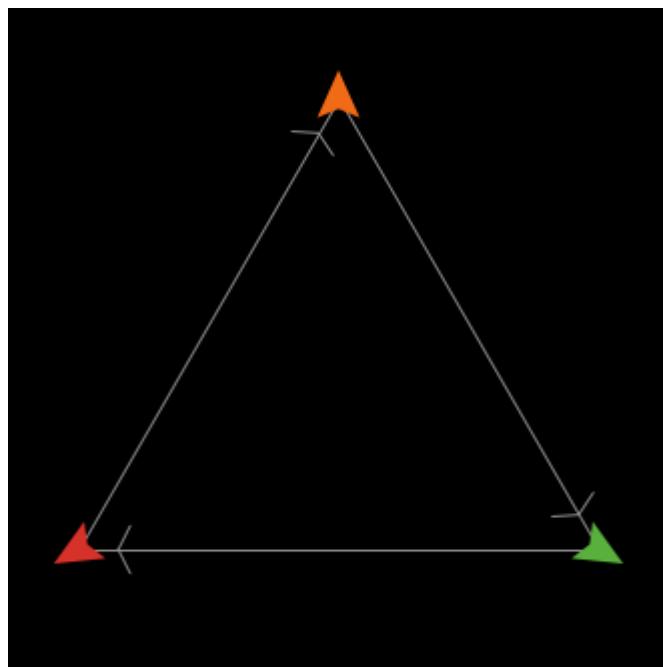
第六行: triangle, triangle 2, truck, turtle, wheel, x

图形库 (Shapes library)

下面是图形库中的所有图形 (包括所有的默认图形) :



默认情况下只有一个链图形，即 default”。这是一个简单的直线加一个简单的箭头（如果有向链的话）



创建和编辑海龟图形

按下 **New** 按钮创建新图形，或者选择已有图形后按下 **Edit**。

工具

在编辑窗口的左上角是一组画图工具。箭头是选择工具，用来选择已经画出的绘图元素。
使用其他七个画图工具绘制新元素。

- **line** 工具用来画线段
- **circle, square, polygon** 有两种模式：.

使用多边形 (polygon) 工具时，单击鼠标增加新线段，画完所有线段后，双击。
画出一个新元素后，它就是当前选择，因此可以移动、删除、修改。

- 要移动，使用鼠标拖拉
- 要删除，按下 **Delete** 按钮
- 要修改，拖动图形上的小句柄，当图形被选择后，句柄显示出来
- 要改变颜色，在新颜色上单击

预览

绘制图形时，在编辑窗口底部有 5 个预览区，显示 5 个小尺寸的图形。预览区显示当出现在模型中时，图形的样子，包括旋转后的样子。每个预览区下面的数字是预览的像素数。例如预览区下的 20，表示在瓦片尺寸为 20 个像素时，海龟（尺寸 1）的显示情况。

如果图形总是面对一个方向，不随海龟方向而改变，可以关闭可旋转特性。

图形叠加（Overlapping Shapes）

新元素在旧元素的上层。选择一个元素后，使用 **Bring to front** 和 **Send to back** 按钮改变他们的层序。

撤销（Undo）

任何时候可以使用 **Undo** 按钮，撤销刚才所做的操作。

颜色

与 *Color that changes* (从下拉式菜单选择 - 默认是灰色) 指定的颜色一致的元素会根据每个海龟的 *color* 变量而变化，而其他颜色的元素不变。例如，你可以创建车辆，他们的车灯总是黄色、车轮总是黑色，而车体有不同的颜色⁸。

其他按钮

“Rotate Left” 和 “Rotate Right” 按钮将元素旋转 90 度。“Flip Horizontal” 和 “Flip Vertical” 按钮根据轴线翻转元素。

这四个按钮一般用来旋转或翻转整个图形。但如果选定了一个元素时，只对该元素进行。

这些按钮与 “Duplicate” 一起，为构造对称图形提供了便利。例如，制作蝴蝶图形时，先用 polygon 工具绘制左翼，然后用 “Duplicate” 复制，在对复制品使用 “Flip Horizontal” 得到右翼。

图形设计

也许有人希望绘制复杂的、有趣的图形，但是要记住在多数模型里瓦片很小，看不到很多图形细节。简单的、粗体的、图标化的图形一般最好。

图形保存（Keeping a Shape）

图形绘制完成后，指定名称，按下编辑窗口底部的 **Done** 按钮。该图形及名字就与 “default” 图形一起，出现在图形列表中。

⁸ 译者注：使用这种方法实现海龟的某些部位颜色不变，而某些部位的颜色随海龟的 *color* 变量值而定。

创建和编辑链图形

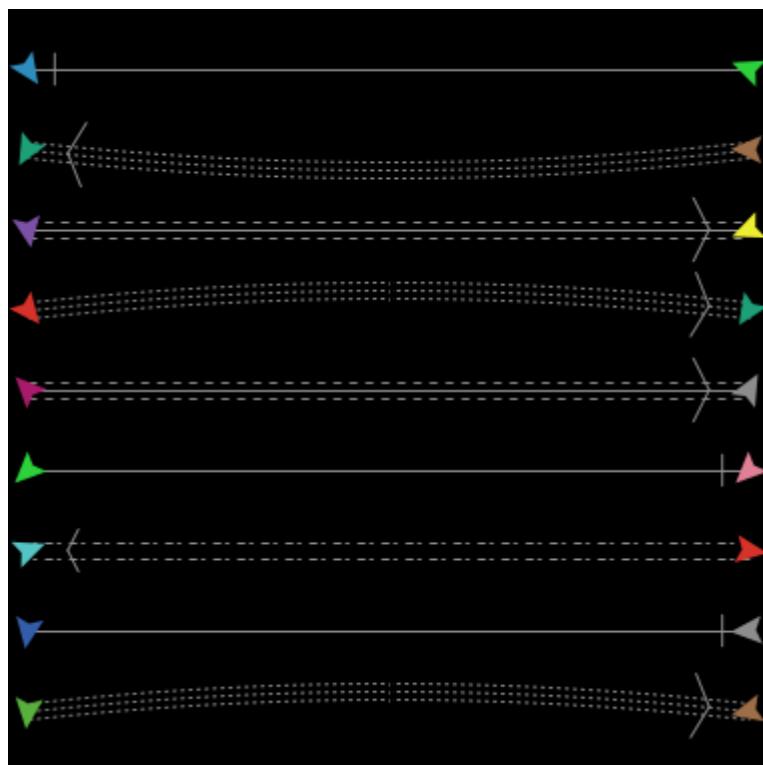
管理链图形与管理海龟图形很相似。使用 **New** 按钮创建新图形或编辑已有图形，编辑完成后按下 **Done** 按钮保存。

改变链图形的属性

每个链图形有几个不同的属性，这些属性可以改变：

- **Name** – 链图形可以与海龟图形同名，但每个链图形必须不同名。
- **Direction Indicator** – 方向标记（有向链上的小箭头）与海龟矢量图形一样，可以使用户同样的编辑器中 **Edit** 按钮编辑。
- **Curviness** – 这是用瓦片数量表示的链弯曲量（如果一对节点之间有两个反向有向链时很有用，可以用来分开这两条链）。
- **Number of lines**: 每个链图形可以有 1, 2, 3 条线。通过选择线模式"left line", "middle line", "right line"做出选择。
- **Dash pattern of lines**: 选择框里有几种点划线，因此不必都是实线。

下面是几条具有不同属性的链图形：



在模型里使用图形

在模型代码或命令中心可以使用模型中的任何图形（尽管海龟只能使用海龟图形，链只能使用链图形）。例如，创建 50 个使用“rabbit”图形的海龟，前提是模型里有名为 *rabbit* 的海龟图形，在命令中心为 observer 指定命令：

```
observer> crt 50
```

然后为海龟指定命令，让他们散开、改变图形：

```
turtles> fd random 15
```

```
turtles> set shape "rabbit"
```

瞧！兔子！注意图形名使用双引号，因为图形名是字符串。

类似的可以设置链的 shape 变量。假设有一个名为“road”的链图形：

```
observer> crt 5 [ create-links-with other turtles ]
```

```
turtles> fd 5
```

```
links> set shape "road"
```

命令 set-default-shape 也可用来为海龟或链分配图形。

行为空间指南（BehaviorSpace Guide）

本指南包括三部分：

- [BehaviorSpace是什么？](#): 对该工具的一般性描述，包括基本思想和原理。
- [怎么用](#): 带你过一遍工具的使用，重点介绍常用的功能。
- [高级用法](#): 怎样在命令行使用BehaviorSpace，或者怎样在Java代码里使用它。

什么是 BehaviorSpace？

BehaviorSpace 是与 NetLogo 集成的工具，用它对模型进行实验。它多次运行模型，系统改变模型的配置，记录每次运行结果。该过程有时称为参数扫描（“parameter sweeping”）。帮助你探索模型的可能行为空间（“space”），确定哪种参数组合导致感兴趣的行为。

为什么需要 BehaviorSpace？

下面的观察表明需要这样的[实验工具](#)。模型一般有多个参数，每个参数有一个取值范围，这些参数一起构成了数学上所谓的参数空间，该空间的维数是参数的数量，空间的一个点代表参数取值的一个组合。用不同的配置（有时甚至同一个配置）运行模型可能导致非常不同的系统行为。因此你怎么知道是哪个配置或哪类配置会产生你感兴趣的行为呢？等价于这样的问题：在这个巨大的多维参数空间里，哪个配置使模型有最佳性能？

例如，假设在 Fireflies 模型中你希望快速同步。模型有四个滑动条—— number, cycle-length, flash-length 和 number-flashes，它们分别大约有 2000, 100, 10 和 3 个可能值，这意味着有 $2000 * 100 * 10 * 3 = 6,000,000$ 个可能组合！要一个个的实验这些组合，了解哪个会激发最快的同步几乎是不可能的。

用 BehaviorSpace 能更好的解决这个问题。如果你指定每个滑动条取值范围的子集，它将对所有可能的组合进行实验，记录每次运行的结果。在这样做时，它对参数空间进行采样——不是穷尽，但足够让你看到滑动条取值与系统行为之间的关系形式。所有运行结束后产生一个数据集，可以用如电子表格、数据库、科学可视化应用程序等工具打开数据集，进行探索。

BehaviorSpace 可能是建模人员的得力助手，因为它能让你探索模型的整个行为“空间”。

历史注解

老版本的 NetLogo (2.0 之前) 包含一个早期版本的 BehaviorSpace 工具。那个版本与现在的版本有很大区别。在设置实验时不太灵活，但却有一些现在版本没有的显示和分析实验结果的工具。在当前版本里，假设你使用其他软件分析结果。我们希望在将来的

BehaviorSpace 版本再加上显示和分析数据的工具。

怎样使用

要开始使用 BehaviorSpace，打开你的模型，然后在 NetLogo 的 Tools 菜单里选择 BehaviorSpace 菜单项。

管理实验设置

现在打开的对话框让你创建、编辑、复制、删除、运行实验设置。所有实验名和模型运行次数一一列出。

实验设置被看作模型的一部分，与模型一起保存。

要创建一个新的实验设置，按下“New”按钮。

创建一个实验设置

在新出现的对话框里，可以指定以下信息。注意不一定要指定所有项，有些空着就可以，或使用默认值，这取决于你的需要。

Experiment name: 如果有多个实验设置，给每个实验指定一个名字，这样比较有条理。

Vary variables as follows: 此处指定哪些参数要变化，它们怎样取值。参数可以是滑动条、开关、选择器、模型中的全局变量。

参数也可以是 [max-pxcor](#), [min-pxcor](#), [max-pycor](#) 和 [min-pycor](#), [world-width](#), [world-height](#) 和 [random-seed](#)。严格来说它们并不是参数，然而BehaviorSpace允许变动它们。变动世界的大小可以用来探索世界规模对模型的影响。因为仅设置[world-width](#) 和 [world-height](#)无法确定世界的边界，世界如何变动取决于原点的位置。如果原点在中心，BehaviorSpace将保持原点的中心地位，因此[world-width](#) 和 [world-height](#)必须是奇数。如果某个边界是 0，则该边界保持为 0，另一侧的边界将移动。例如世界初始时是 `min-pxcor = 0, max-pxcor = 10`，这样变动 `world-width`:

```
[“world-width” [11 1 14]]
```

则 [min-pxcor](#) 将保持为 0，而每次运行时 [max-pxcor](#) 将设为 11, 12, 13。如果二者均不为真，即原点不在中心，也不在边界，则不能直接变动 [world-height](#) 或 [world-width](#)，你应该变动 [max-pxcor](#), [max-pycor](#), [min-pxcor](#), [min-pycor](#)。

变动 [random-seed](#) 使你为 NetLogo 随机数发生器指定已知的种子重复运行。注意在实验设置命令里也可使用 [random-seed](#) 命令。关于随机数种子的详细信息，参见编程指南的随机数部分 [Random Numbers](#)。

有两种方法用来指定要使用的参数值，一是将要使用的所有值列出来，二是指定一个范围尝试其中的每个值。例如，要指定名为 `number` 的滑动条数值从 100 到 1000，增量为 50，输入：

```
["number" [100 50 1000]]
```

或者只使用值 100, 200, 400, 800, 输入:

```
["number" 100 200 400 800]
```

小心使用方括号, 注意第二个例子方括号较少。是否包含这对多余的方括号是告诉 NetLogo 列出所有值还是指定一个范围。

还要注意变量名要用双引号。

可以变动一个参数或多个参数, 或者不变动任何参数。不变的参数保持当前值。当使用当前设置多次运行模型时, 就不变动任何参数。

变量列出的顺序决定了构造参数组合的顺序。后列出的变量的所有值要遍历一遍, 然后再遍历较早列出的变量。例如希望 x 和 y 都从 1 到 3 变动, x 先列出, 则模型运行的顺序为: x=1 y=1, x=1 y=2, x=1 y=3, x=2 y=1, ...。

Repetitions: 有时候即使模型的配置不变, 各次运行的行为也可能有很大变动。如果要对每个配置运行多次, 在这儿输入一个大于 1 的数。

Measure runs using these reporters: 在这指定每次运行要收集的数据。例如, 要记录每次运行海龟数量的起伏, 输入:

```
count turtles
```

可以输入一个或几个报告器, 或一个也没有。如果输入多个的话, 每个报告器要占一行, 例如:

```
count frogs
count mice
count birds
```

如果不输入任何报告器, 模型照样会运行。有时你自己记录结果, 例如使用[export-world](#) 命令等。

Measure runs at every step: 正常情况下, 每步都会使用上面给定的报告器对模型进行测量。如果模型运行时间很长, 不想收集所有各步数据, 反选该项则只在模型每次运行结束时进行测量。

Setup commands: 使用这些命令启动每次模型运行。通常在这输入模型设置例程名进行设置。但是也可以包含其他命令。

Go commands: 这些命令将不断执行, 将模型推进到下一步。通常在这输入模型运行例程名, 如 go。但是也可以包含其他命令。

Stop condition: 此处可实现模型运行时间可变, 当某个条件满足时模型本次运行结束。例如, 假设模型运行直到没有海龟为止, 输入:

```
not any? turtles
```

如果每次运行固定长度, 此处留空即可。

模型也可以因为go命令中的[stop](#) 而停止运行, 或者因为使用了永久性按钮的[stop.stop](#) 可以直接用在go例程, 也可以用在由go直接调用的其他例程。(意图是同一个go例程可以用于按钮也可以用于BehaviorSpace实验)。注意使用stop的该步被认为是失败, 不记录该步

的结果。因此停止测试应该用在 go 的开始部分而不是结束部分。

Final commands: 这些命令运行一次，当模型结束时运行这些额外命令。一般此处留空，但可用于调用 [export-world](#) 命令，或用其他方式记录运行结果。

Time limit: 设置每次运行的最大长度。如果不设置任何最大值，而是由停止条件控制运行长度，在这输入 0。

运行一个实验

实验设置完成后，按下“OK”按钮，然后按下“Run”按钮。

这时出现提示，让你选择实验数据的存储格式。根据在 **Measure runs at every step** 的设置，将会在每个间隔、每次运行或每步收集数据。

Table 格式每个间隔的数据占一行，每项数据占一列。每次运行结束后数据输出到数据文件。Table 格式适合自动处理数据，比如导入到一个数据库或统计包中。

Spreadsheet 格式为每项数据计算最小、平均、最大和最终值，然后列出数据，每个间隔的数据占一行，每项数据占一列。这种格式更适合人类阅读，特别是当导入到一个电子表格程序时。

（注意直到整个实验完成后，spreadsheet 格式数据才会写到数据文件。由于实验完成前数据都存在内存中，如果实验很大可能导致内存耗尽。如果因某种原因实验中断，可能的原因如出现运行错、内存耗尽、崩溃、断电等，则实验结果没有保存。对大型实验可能需要同时使用 spreadsheet 和 table 格式，这样即使出现问题，至少通过 Table 格式得到部分结果。）

选择了输出格式后，BehaviorSpace 提示你输入保存数据的文件名，文件默认后缀是 “.csv”。你可以随便给个文件名，但最好不要丢弃“.csv”部分，因为它表示这是一个 Comma Separated Values (CSV) 文件。这是一个纯文本数据格式，能被任何文本编辑器、多数流行的电子表格和数据库程序读取。

这时出现一个标题为“Running Experiment”的对话框。在对话框里，可以看到进度报告，显示模型运行了多少次，运行了多长时间。如果你输入了测量运行的任何报告器以及选择了“Measure runs at every step”，你将看到一个绘图，显示每次运行中该值如何变化。

你也可以在 NetLogo 主窗口中看到运行情况。（如果“Running Experiment”对话框挡住了主窗口，把它移动到其他位置）。模型运行时视图和绘图进行更新，如果不想要看到更新，则使用“Running Experiment”对话框中的选择框关闭更新，这样实验运行的更快一些。

如果在实验完成前需要停止，按下“Abort”按钮。但要注意这会失去至今为止的 spreadsheet 格式的数据。Table 格式的数据不会丢失，因为随着实验运行数据已经写到文件了。

当所有运行完成后，实验结束。

高级用法

从非图形界面运行

BehaviorSpace 实验可以用非图形界面（"headless"）方式运行，即在命令行中运行，没有任何图形界面。当需要在单机或机群中自动运行时这种方式可能有用。

这种方式不需要 Java 编程，可以先用 GUI 方式创建实验设置，然后在命令行中运行实验。或者如果你喜欢的话，可以直接用 XML 创建、编辑实验设置。

最容易的方式是先在 GUI 里创建实验设置，作为模型的一部分保存。下面是一个在命令行里运行已保存在模型中的实验设置的例子：

```
java -server -Xmx1024M -cp NetLogo.jar \
  org.nlogo.headless.HeadlessWorkspace \
  --model Fire.nlogo \
  --experiment experiment1
```

（要正常运行，NetLogo.jar 和包含必要库文件的 lib 子目录必须存在。
NetLogo.jar 和 lib 都包含在 NetLogo 里）

指定名称的实验运行完成后，结果以 spreadsheet 格式，即 CSV，送到标准输出。（要改变这一点，看下面）

当将 HeadlessWorkspace 类作为应用程序运行时，强迫系统特性 java.awt.headless 为真。这将告诉 Java 以无头（headless）模式运行，允许 NetLogo 在没有图形显示的机器上运行。

注意 -server 标志是告诉 Java 为 "server" 类型的应用做性能优化。在多数情况下我们推荐使用该标志，以获得最佳性能。

注意使用 -Xmx 指定最大 1G 的堆大小。如果没有指定大小则使用虚拟机的默认值，默认值可能太小。（1G 是武断给出的，可能比多数模型所需的都大，你可以自己指定所需的大小）

选项 --model 用来指定要打开的模型。

选项 --experiment 用来指定实验名。（在 GUI 里创建实验时指定了名称）

下面是另一个例子，显示了一些附加的、可选参数：

```
java -server -Xmx1024M -cp NetLogo.jar \
  org.nlogo.headless.HeadlessWorkspace \
  --model Fire.nlogo \
  --experiment experiment2 \
  --max-pxcor 100 \
  --min-pxcor -100 \
  --max-pycor 100 \
  --min-pycor -100 \
  --no-results
```

注意可选的 --max-pxcor, --max-pycor 等参数用来指定与模型保存的参数不同的世

界大小。（也可以在实验设置里指定世界大小，如果这样的话就不需要在命令行里指定了）

还要注意可选项--no-results参数指定不产生输出。如果你需要以其他方式产生输出，例如输出世界文件或写到文本文件等，这个选项就有用了。

还有另一个例子：

```
java -server -Xmx1024M -cp NetLogo.jar \
    org.nlogo.headless.HeadlessWorkspace \
    --model Fire.nlogo \
    --experiment experiment2 \
    --table table-output.csv \
    --spreadsheet spreadsheet-output.csv
```

可选项--table <filename>参数指定输出格式为table格式，并且以CSV形式写到给定文件。如果将以 - 作为文件名，则输出到标准输出流。Table数据每次运行完成就产生，然后写入。

可选项--spreadsheet <filename>参数指定输出格式为spreadsheet格式，并且以CSV形式写到给定文件。如果将 - 作为文件名，则输出到标准输出流。直到所有运行完成，实验结束后，spreadsheet数据才写入。

注意同时指定--table 和 - spreadsheet是允许的，如果同时指定的话，则同时产生两种类型的输出文件。

没有指定输出格式时，默认的输出行为是将 table 输出发送到系统标准输出流。

下面是最后一个例子，显示怎样运行一个以独立 XML 文件存储的实验设置，而不是在模型里保存的实验设置：

```
java -server -Xmx1024M -cp NetLogo.jar \
    org.nlogo.headless.HeadlessWorkspace \
    --model Fire.nlogo \
    --setup-file fire-setups.xml \
    --experiment experiment3
```

如果XML文件包含多个实验设置，必须使用--experiment参数指定要使用的实验名。

下面部分介绍怎样使用 XML 创建独立的实验设置文件。

用 XML 设置实验

目前我们还没有介绍用 XML 写实验设置的详细文档，但是如果你对 XML 比较熟悉的话，下面的几点介绍对你来说足够了。

在XML里，BehaviorSpace实验设置的结构是由一个文档类型定义文件（Document Type Definition，DTD）决定的。DTD存储在NetLogo.jar的system/behaviorspace.dtd里。

（JAR也是压缩文件，可以使用Java的“jar”工具或其他任何能够操作压缩文件的工具，从JAR文件里抽取DTD）

要了解实验设置在 XML 里的样子，最容易的学习方式是在 BehaviorSpace 的 GUI 里编写几个实验设置，保存模型，然后用文本编辑器打开产生的.nlogo 文件。实验设置在.nlogo

文件的尾部，该部分以实验标志开始与结束，例如：

```
<experiments>
  <experiment name="experiment" repetitions="10" runMetricsEveryStep="true">
    <setup>setup</setup>
    <go>go</go>
    <exitCondition>not any? fires</exitCondition>
    <metric>burned-trees</metric>
    <enumeratedValueSet variable="density">
      <value value="40"/>
      <value value="0.1"/>
      <value value="70"/>
    </enumeratedValueSet>
  </experiment>
</experiments>
```

该例子中只有一个实验设置，但是在实验开始标志和结束标志之间可以给定任意多个实验设置。

看看 DTD，再看看 GUI 中创建的例子，你能了解如何使用标志 (tag) 指定不同的实验。DTD 指定哪些标志是必须的，哪些是可选的，哪些可重复哪些不能重复，等等。

当实验设置 XML 包含在模型文件中时，没有以 XML 标题 (header) 开始，因为整个文件不是 XML。如果你将实验设置独立保存，而不是随模型文件一起保存的话，扩展名应是. xml 而不是. nlogo，并且文件应以正确的 XML 标题开始，如下：

```
<?xml version="1.0" encoding="us-ascii"?>
<!DOCTYPE experiments SYSTEM "behaviorspace.dtd">
```

第二行必须与上面的完全一样。第一行可以指定与us-ascii不同的编码，例如 UTF-8，但NetLogo多数情况下不支持非ASCII字符，所以指定不同的编码没什么意义。

控制 API (Controlling API)

如果BehaviorSpace不能满足你的需要的话，另一个可能的方法是使用Controlling API，让你写Java代码控制NetLogo。这些API让你在Java代码里运行BehaviorSpace实验，或者自己写代码直接控制NetLogo做类似BehaviorSpace的事情。详情见用户手册的[Controlling](#)部分，了解这两种可能方法。

结论

BehaviorSpace仍在开发之中。我们希望听到你的反映，告诉我们需要哪些功能。请发邮件到feedback@ccl.northwestern.edu。

系统动力学指南（System Dynamics Guide）

本指南包括三个部分：

- [系统动力学建模工具（Modeler）是什么？](#): 对工具做一般性描述，包括其基本思想和内部原理。
- [怎样使用](#): 描述界面和如何使用。
- [教学：狼吃羊（集计）](#): 带领你使用系统动力学建模工具创建一个模型。

NetLogo 系统动力学建模工具是什么？

系统动力学是用来理解事物是如何相关的一种模型。它与我们在 NetLogo 里经常使用的基于主体的方法有所区别。

在 NetLogo 里我们常用基于主体的方法，对单个主体的行为进行编程，观察它们的相互作用涌现出的现象。例如在狼吃羊模型里，确定狼、羊、草之间的交互规则，当模型运行时，观察涌现出的总体层次上的行为：例如狼和羊的数量如何随时间变化。

使用系统动力学建模工具时不是对个体行为编程，而是对一群主体的整体行为进行编程。例如在建立狼吃羊的系统动力学模型时，你指定羊群的数量如何随狼群的数量变化而变化，反之亦然，然后运行模型，观察两类群体的数量如何随时间变化。

使用系统动力学建模工具画出流图（diagram），定义这些总体数量或存量（"stocks"）如何相互影响。建模工具读取这些图，生成合适的 NetLogo 代码 - 全局变量、例程和报告器 - 在 NetLogo 里运行你的系统动力学模型。

基本概念

系统动力学流图由四类元素构成：存量（Stock）、变量（Variable）、流量（Flow）和连接（Link）⁹。

存量是物品的集合，是一个积聚。例如存量可以表示羊群数量、湖中的水或工厂里零件数量等。

流将物品送入或流出存量。流看起来像一个带阀门的管道，阀门控制多少物品流过管道。

变量是在流图中使用的数值，可以是依赖于其他变量的方程，也可以是个常数。

连接使图中某部分的值在其他部分可用，连接把来自变量或存量的数值传送到一个存量或流。

通过不断估计，系统动力学建模工具计算出存量如何随时间变化。估计并不是总准确，但可以通过改变 `dt` 值影响精度。随着 `dt` 减小，估计次数增加，精度增加。然而减小 `dt` 会使模型运行变慢。

⁹ 译者注：NetLogo的系统动力学模型采用的四个概念与系统动力学标准概念有一些差别。NetLogo存量对应水平变量，流量实际上是速率变量，变量对应辅助变量和常量，连接是流线。

模型实例

在 NetLogo 模型库的 Sample Models 部分有四个系统动力学模型，这四个模型都是研究种群数量增长的（在捕食模型里，数量减少）。

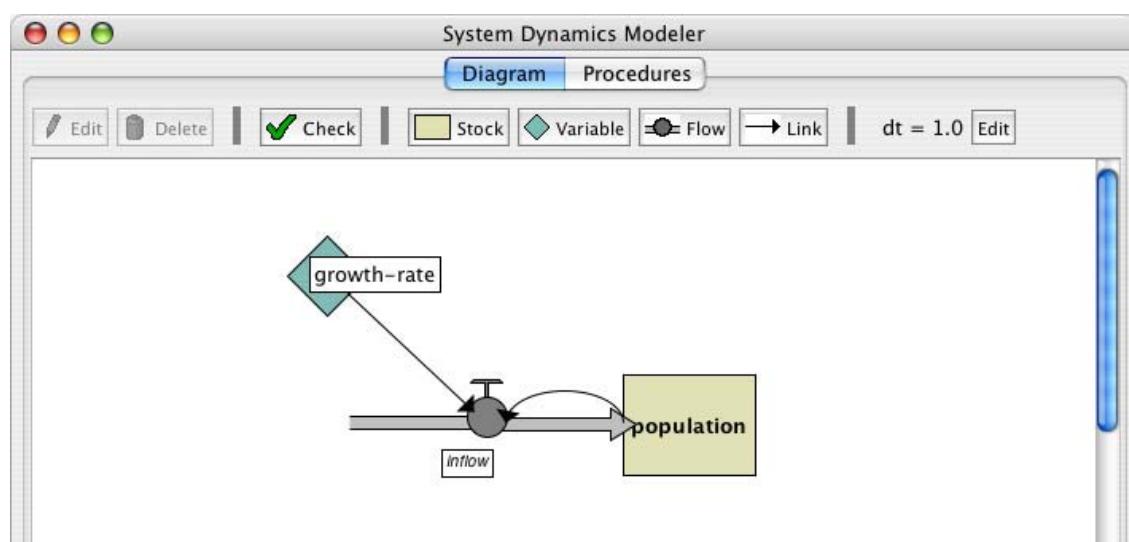
Exponential Growth 和 **Logistic Growth** 是只有一个存量的增长例子。

Wolf Sheep Predation (aggregate) 是有多个存量，存量相互影响的例子。这是一个使用系统动力学建立的捕食-食饵生态系统模型。

Wolf Sheep Predation (docked) 是一个同时运行系统动力学模型和基于主体的模型的例子。它运行狼吃羊系统动力学模型以及基于主体的狼吃羊模型，它位于 Sample Models 的 Biology 部分。

怎样使用

在 Tools 菜单选择 System Dynamics Modeler 菜单项，打开系统动力学建模工具，出现建模窗口。



流图页

在流图页绘制系统动力学流图。

工具条上有编辑、删除、创建流图元素的按钮。

创建流图元素



系统动力学流图有四类部件：存量、变量、流和连接。

存量

要创建存量，在工具条上按下 Stock 按钮，在下面的绘图区单击，就出现新的存量。每个存量要有唯一名，作为全局变量。存量还需要指定 **Initial value**（初始值），初始值可以是数值、变量、复杂的 NetLogo 表达式、或是对 NetLogo 报告器的调用。

变量

要创建变量，在工具条上按下 Variable 按钮，在下面的绘图区单击。每个变量要有唯一名，它作为例程名或全局变量名。变量还需要一个 **Expression**（表达式），可以是数值、变量、复杂的 NetLogo 表达式、或是对 NetLogo 报告器的调用。

流

要创建流，在工具条上按下 Flow 按钮，在你希望流开始的地方 — 存量或空白区 - 单击并保持，拖动鼠标到流结束之处 - 存量或空白区。每个流需要唯一名，它成为 NetLogo 报告器。流需要一个表达式，表示从输入到输出的流率，流率可以是数值、变量、复杂的 NetLogo 表达式、或是对 NetLogo 报告器的调用。如果值为负，则流是反向的。

当多个流连到一个存量上时，要重点考虑它们之间的交互作用。NetLogo 不强迫指定出自一个存量的流的顺序。NetLogo 也不确保存量的流出之和小于等于流入。这些行为可以在创建流的表达式时显式实现。

例如，如果流定义为常数 10，你可以使用 min 原语 `min (list stock 10)` 确保不会取出超过存量的值。如果需要流A在流B计算之前消耗存量，可以将流A连到流B，修改流B减去流A取自存量的值：`min (list (max (list 0 (stock - flow-a))) 10)`。

连接

要创建连接，在你希望连接开始的地方 — 变量、存量或流 - 单击并保持，拖动鼠标到目的变量或流。

操作流图元素

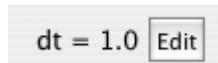
创建存量、变量、流之后，你会看到在这些流图元素上有一个红色问号。问号表示该元素还没有命名，红色表示存量不完全：缺少产生模型所需的一个或多个值。当流图要素设置完整后，名字变黑。

选择：要选择一个元素，在上面单击。要选择多个元素，按着 shift 键。也可以拖动一个选择框选择多个元素。

编辑：要编辑一个元素，选定后按下工具条上的“Edit”按钮，或双击该元素。（能够编辑存量、流和变量，但不能编辑连接）

移动：要移动流图元素，选定后用鼠标拖动。

编辑 dt



工具条右侧是默认 `dt`，它是系统动力学模型用来计算结果的时间间隔。要改变它的默认值，按下 `dt` 附近的 **Edit** 按钮，输入新值。

错误

当单击“check”按钮或编辑存量、流或变量后，建模工具将自动产生与流图对应的 NetLogo 代码，进行编译。如果有错，例程页变红，显示一条消息，有错的代码部分会高亮显示。

Nothing named SHEP has been defined

Diagram **Procedures**

```

;; use temporary variables so order of computation doesn't affect result.
let new-sheep max( list 0 ( sheep + local-sheep-births ) )
set sheep new-sheep

tick-advance dt
end

;; Report value of flow
to-report sheep-births
  report sheep-birth-rate * shep * dt
end

;; Plot the current state of the system dynamics model's stocks
;; Call this procedure in your model's GO procedure.
to system-dynamics-do-plot
  if plot-pen-exists? "sheep" [
    set-current-plot-pen "sheep"
    plotxy ticks sheep
  ]
end

```

这让你更好了解模型的错误之处。



例程页

系统动力学建模工具根据你画的流图产生 NetLogo 变量和例程，正是这些例程实际执行计算功能。系统动力学建模工具的例程页显示根据流图产生的 NetLogo 例程。

你不能修改例程页的内容，要修改系统动力学模型，只能编辑流图。

让我们仔细看看所产生的代码与流图的关系：

- 存量对应一个全局变量，它根据 **Initial value** 域初始化为给定的值或表达式。每个存量每个时间步根据进入和离开的流进行更新。
- 流对应一个例程，它包含 **Expression** 域给定的表达式。
- 变量可能是全局变量，也可能是一个例程。如果你提供的表达式是常数，则它就是一个全局变量，并且初始化为该值。如果使用一个复杂表达式定义变量，则像流那样创建一个例程。

在该页定义的变量和例程可以在 NetLogo 主窗口中访问，就像在 NetLogo 主窗口例程页中定义的变量和例程一样。可以在主例程页、命令中心、界面页的按钮处调用这些例程。可以在任何地方引用这些全局变量，包括主例程页和监视器。

有三个重要的例程需要注意：`system-dynamics-setup`, `system-dynamics-go` 和 `system-dynamics-do-plot`。

`system-dynamics-setup` 初始化集计模型。它设置 `dt`，调用 `reset-ticks`，初始化存量和转换器 (converter)，带有常数的转换器首先初始化，然后是带有常数的存量，其他的存量按字母顺序初始化。

`system-dynamics-go` 运行集计模型 `dt` 时间。它计算流和变量值，更新存量值。还调用以 `dt` 为参数的 `tick-advance`。该例程调用时，由非常数表达式定义的转换器和流只计算一次，然而他们的计算顺序是不确定的。

`system-dynamics-do-plot` 绘制集计模型的存量值。要使用它，先要在 NetLogo 主窗口创建一个绘图，然后为每个要绘制的存量定义一个画笔。该例程使用当前绘图，可以使用 `set-current-plot` 命令改变当前绘图。

系统动力学建模工具和 NetLogo

使用系统动力学建模工具绘制的流图以及根据流图产生的例程是 NetLogo 模型的一部分。当保存模型时，流图也保存在同一个文件里。

教学：狼吃羊(Wolf-Sheep Predation)

我们使用系统动力学建模工具创建狼吃羊模型。

第1步：羊群繁殖

- 在 NetLogo 里打开一个新模型。
- 在 Tools 菜单启动系统动力学建模工具（System Dynamics Modeler）。



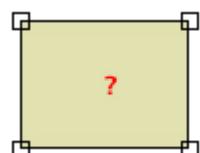
我们的模型包括狼群和羊群，从羊群开始建模。首先创建一个存量，保存羊的数量。

- 按下工具条中的 Stock 按钮。

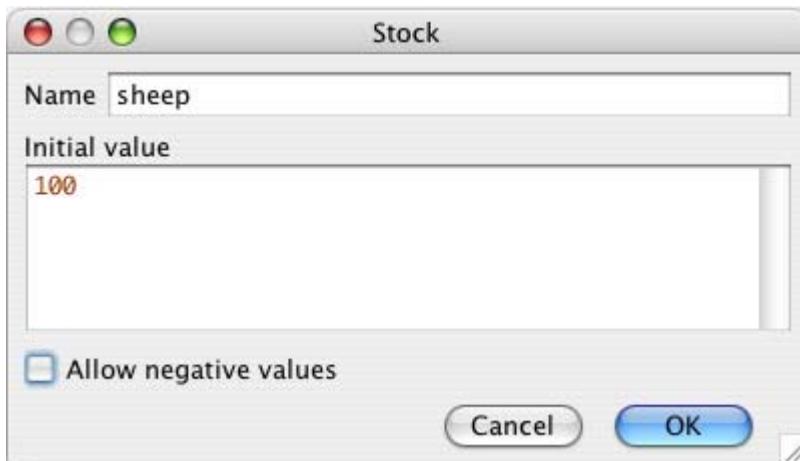


- 在绘图区单击。

你会看到一个存量，中间是红色问号。



- 双击编辑该存量。
- 将存量命名为sheep
- 设置初始值（initial value）为100。
- 令 Allow Negative Values 选择框为未选，因为羊群为负不合理！



如果有羊出生，则羊的数量增加。要实现这一点，创建一个进入羊群存量的流量。

- 单击工具条上的 Flow 按钮，在 Sheep 存量左边的空白区按下鼠标按钮，向右拖动流直到它的右侧连到 Sheep 存量。
- 编辑该流量，命名为 sheep-births

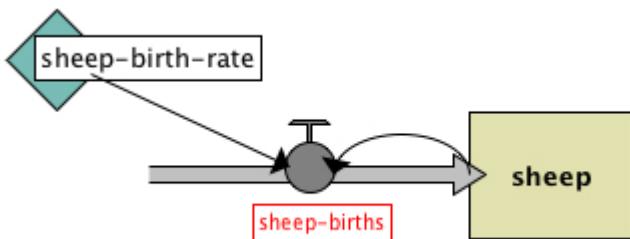
一段时间内羊的出生数量依赖于活羊的数量：羊越多则生的越多。

- 从 sheep 存量到 sheep-births 流量画一条连接（Link）

羊的出生率还取决于某些常数因子，如繁殖率等，这超出本模型讨论的范围。

- 创建一个变量，命名为 sheep-birth-rate，设置它的值是 0.04。¹⁰
- 从变量 sheep-birth-rate 画一条连线到 sheep-births

流图应该像下面这样：



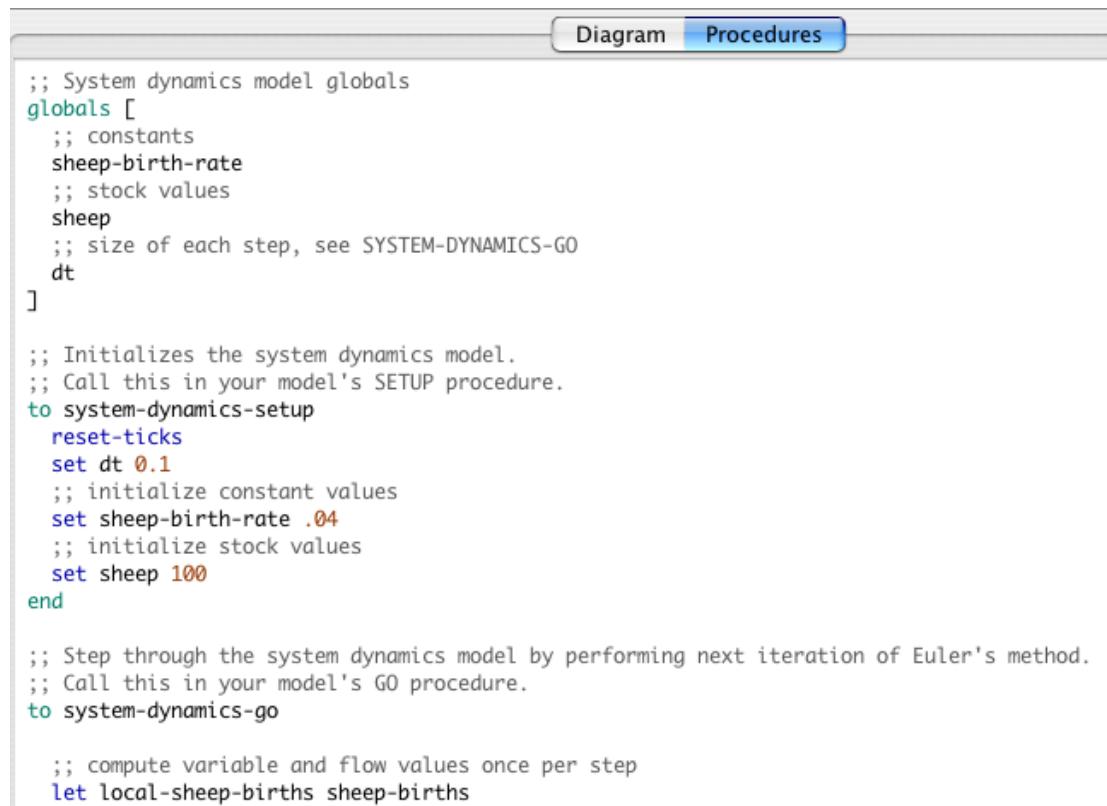
流量 sheep-births 有一个红色标签，因为还没有给它指定表达式。红色表示模型此处还缺少一些东西。

流入存量的羊数与羊群数量和羊的出生率成正比。

- 编辑流量 sheep-births，设置其表达式为 sheep-birth-rate * sheep.

¹⁰ 译者注：原文此处有误，将两条操作的次序放置颠倒了。

现在有了一个完整流图。单击系统动力学建模工具的例程页，查看由流图产生的代码。看起来像这样：



The screenshot shows the NetLogo interface with the 'Procedures' tab selected. The code area contains the following System Dynamics model code:

```

;; System dynamics model globals
globals [
  ;; constants
  sheep-birth-rate
  ;; stock values
  sheep
  ;; size of each step, see SYSTEM-DYNAMICS-GO
  dt
]

;; Initializes the system dynamics model.
;; Call this in your model's SETUP procedure.
to system-dynamics-setup
  reset-ticks
  set dt 0.1
  ;; initialize constant values
  set sheep-birth-rate .04
  ;; initialize stock values
  set sheep 100
end

;; Step through the system dynamics model by performing next iteration of Euler's method.
;; Call this in your model's GO procedure.
to system-dynamics-go
  let local-sheep-births sheep-births

```

第 2 步：NetLogo 集成

一旦使用系统动力学建模工具创建了集计模型，就可以在NetLogo主界面窗口中与模型交互。现在构建NetLogo模型来运行由流图产生的代码。我们需要一个setup和 go 按钮，分别调用由系统动力学建模工具产生的 system-dynamics-setup 和 system-dynamics-go 例程。我们还需要一个监视器和一个绘图，用于查看羊群数量的变化。

- 选择 NetLogo 主窗口
- 在例程页，写代码如下：

```

to setup
  ca
  system-dynamics-setup
end

to go
  system-dynamics-go

```

```
system-dynamics-do-plot
end
```

- 转移到界面页
- 创建setup按钮
- 创建go 按钮 (记住设为forever)
- 创建sheep监视器
- 创建绘图"populations" , 提供画笔"sheep".

现在准备运行模型。

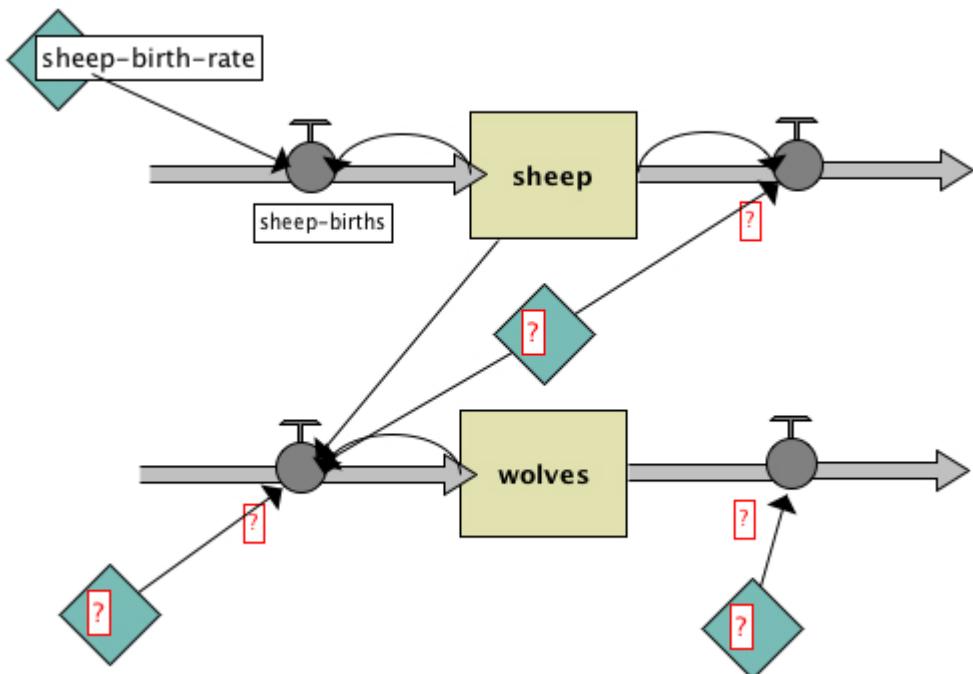
- 按下 setup 按钮。
- 现在不要按 "go" 按钮, 相反在命令中心里输入四五次go .

注意发生的事情。羊的数量指数增加, 经过四五次循环后, 羊的数量很大。这是因为羊只有出生, 没有死亡。

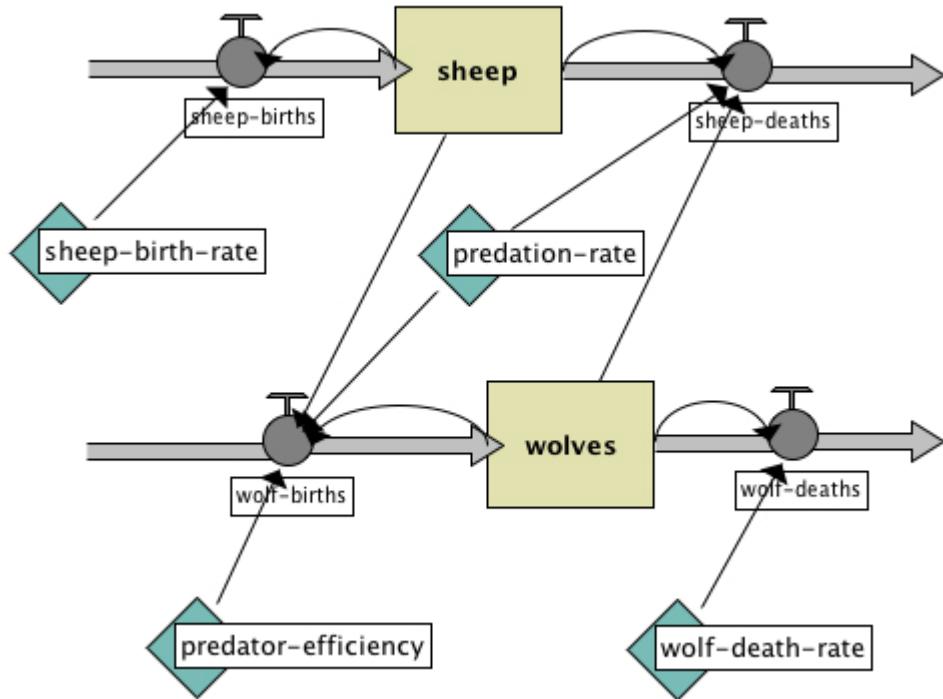
下面我们引入狼群, 狼吃羊, 来修正上面的模型。

第3步：狼捕食羊

- 返回系统动力学窗口
- 增加一个存量 wolves
- 增加流量、变量和连接, 如下图所示:



- 再增加一个连接¹¹，从存量wolves连接到流出Sheep的流量
- 为各个元素添上名字，如下图所示：



其中

wolves 的初值为30,

wolf-deaths的表达式为 $wolves * wolf-death-rate$,

wolf-death-rate 是 0.15,

predator-efficiency 是 .8,

wolf-births的表达式是 $wolves * predator-efficiency * predation-rate * sheep$,

predation-rate 是 3.0E-4,

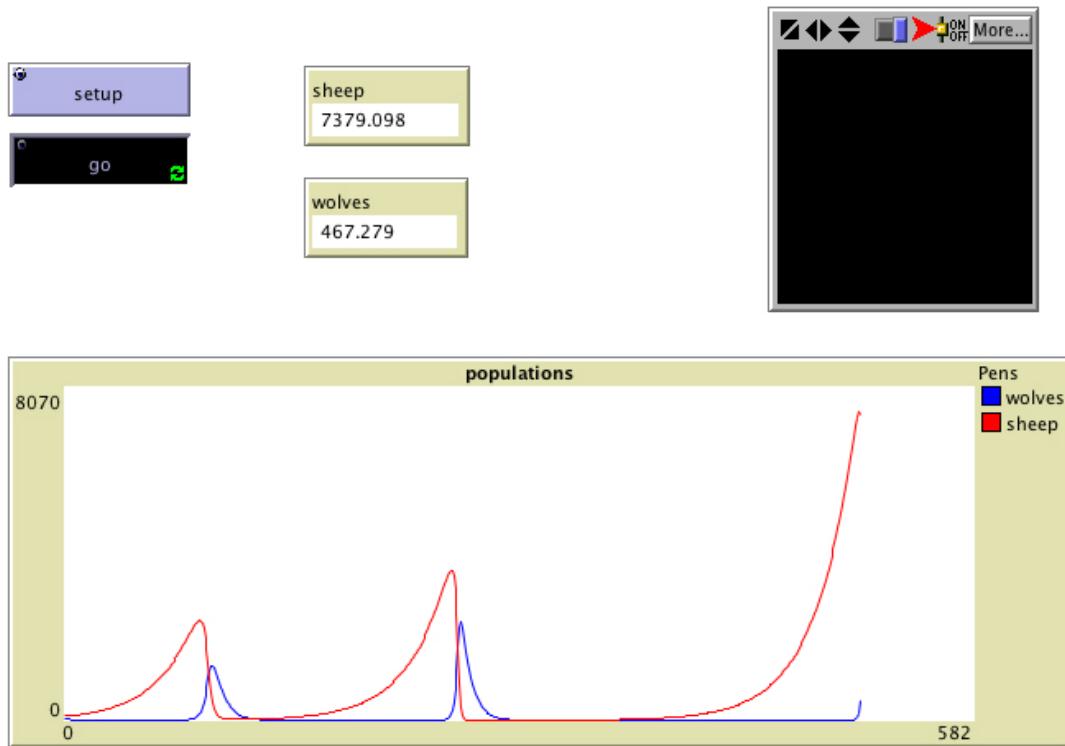
sheep-deaths 的表达式是 $sheep * predation-rate * wolves$.

现在真正完成了。

- 返回 NetLogo 主窗口
- 为羊群绘图增加一个名为 "wolves" 的画笔
- 按下setup和go，查看系统动力学建模工具的效果

种群数量图形如下所示：

¹¹ 译者注：原文为Flow，有误。



结论

系统动力学建模工具还在开发之中。

改进计划

我们想要添加以下功能：

- 增加复制、保存系统动力学流图的能力
- 当编辑存量或流量时，显示由连接提供的变量
- 使连接更容易选中

反馈

如 果 发 现 系 统 动 力 学 建 模 工 具 有 任 何 Bug ， 请 给 我 们 发 邮 件 bugs@ccl.northwestern.edu 。

我们很愿意听到你告诉我们工作中所需要的附加功能。如果有任何问题、评论或建议，请发邮件到feedback@ccl.northwestern.edu 。

HubNet 指南（HubNet Guide）

本部分介绍 HubNet 系统，还包括如何设置和使用 HubNet 活动（activity）的说明。

HubNet 是一种使用 NetLogo 在教室里进行参与式仿真（*participatory simulations*）的技术。在参与式仿真中，整个班级扮演系统行为，每个学生使用个人设备，如联网计算机或 TI 图形计算器，控制系统的一部分。

例如，在 Gridlock 仿真中，每个学生控制模拟城市中的一个交通信号灯，整个班级试图使城市交通高效运行。仿真运行时收集数据，以后这些数据可以在计算机或计算器上进行分析。

要了解关于参与式仿真的更多信息和学习潜力，请访问[Participatory Simulations Project web site](#)

理解 HubNet

NetLogo

NetLogo 是一个可编程建模环境，带有很大的一个模型库，既有参与式仿真模型，也有传统模型。模型库覆盖的领域有社会科学和经济学，生物学和医学，物理和化学，以及数学和计算机科学。你和你的学生可以使用它们建立自己的模型。要了解更多的 NetLogo 信息，参见 NetLogo 用户手册。

传统的 NetLogo 仿真根据建模人员给出的规则运行。HubNet 为 NetLogo 增加了一个维度，仿真不仅可以根据事先指定的规则运行，还可以有人直接参与。

HubNet 是基于 NetLogo 的，在你第一次尝试 HubNet 之前，最好对 NetLogo 的基本情况有所了解。要使用 NetLogo 模型，参见 NetLogo 用户手册的[教学 1：运行模型](#)。

HubNet 结构

HubNet 仿真基于客户-服务器结构。活动领导者使用 NetLogo 应用程序运行一个 HubNet 活动，当 NetLogo 运行一个 HubNet 活动时，称之为 HubNet 服务器。参与者使用一个客户程序登录到服务器，与 HubNet 服务器进行交互。

有两种 HubNet。一是计算机 HubNet，参与者在一台联网的计算机上运行 HubNet 客户程序。二是计算器 HubNet，参与者使用 TI 图形计算器，计算器通过 TI-Navigator 系统互联。

我们希望增加对其他类型客户的 support，比如移动电话或 PDA 等。

计算机 HubNet

活动 (Activities)

在模型库的Computer HubNet Activities部分有下面的一些活动。在[Participatory Simulations Project web site](#)能找到多数模型的教学目标, 以及如何在教室里进行参与式仿真的讨论。在每个模型的信息页也能找到一些信息。

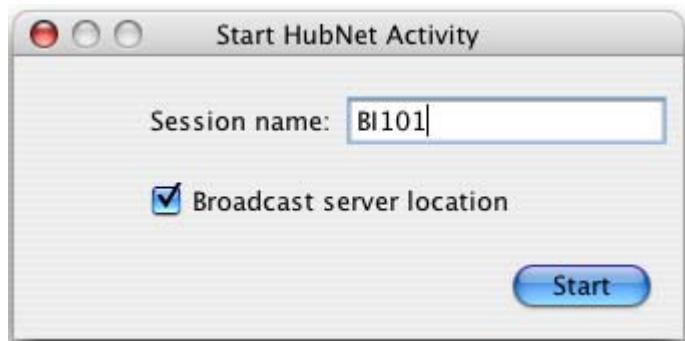
- Disease – 疾病在学生群中传播.
- Gridlock – 学生使用交通灯控制城市中的交通流.
- Polling – 问学生问题, 绘制他们的答案.
- Tragedy of the Commons – 学生扮演使用公共资源的农民

需求

要使用计算机 HubNet, 活动领导者需要一台安装了 NetLogo 的联网计算机, 每个参与者也是这样。推荐领导者使用投影将整个仿真显示给参与者。

启动一个活动

在 NetLogo 模型库的 HubNet Computer Activities 文件夹找一个 HubNet 活动。建议你在正式上课之前试着练习几次。



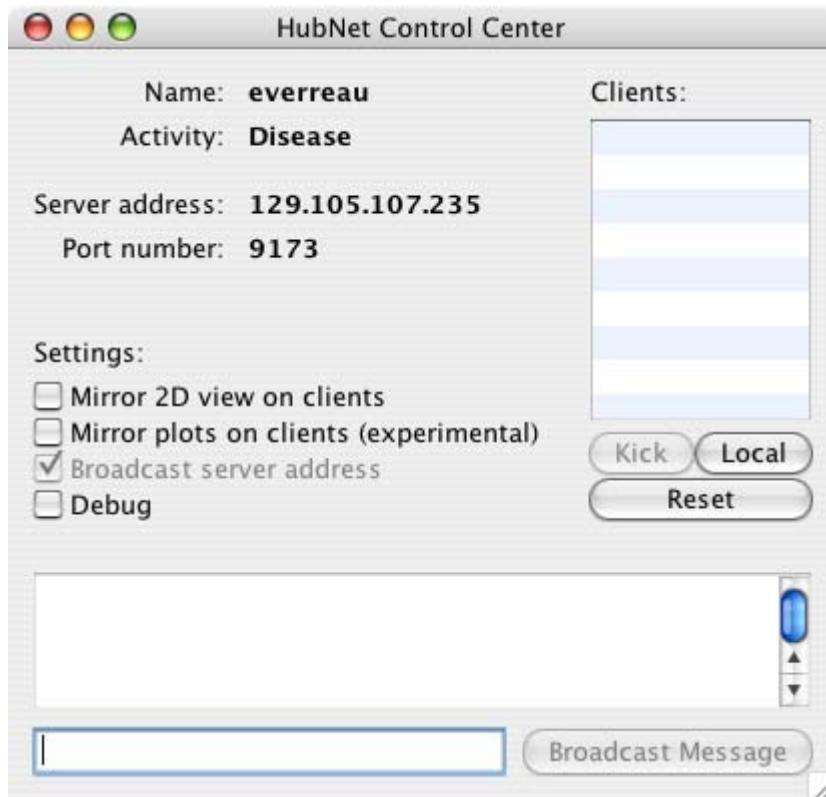
打开一个计算机 HubNet 模型。NetLogo 提示你输入新的 HubNet 会话的名字。参与者将使用这个名字来识别活动。输入一个名字, 按下 Start 。

NetLogo 打开 HubNet 控制中心 (Control Center), 使用控制中心与 HubNet 服务器交互。

做为领导者, 你应该通知每个人可以加入了。要加入活动, 参与者打开 HubNet 客户程序, 输入他们的名字。他们应该能看到你的活动已列出, 选择该活动后, 按下 Enter 加入活动。如果你的活动没有列出, 学生可以手工输入服务器地址, 该地址可以在 HubNet 控制中

心找到。

HubNet 控制中心



HubNet 控制中心用来和 HubNet 服务器进行交互。其中显示服务器的名字、活动、地址和端口号。“Mirror 2D View on clients”选择框控制参与者是否在他们的客户程序上看到视图，当然前提是客户设置了视图。“Mirror plots on clients”选择框控制参与者是否接收绘图信息。

右边的 client 列表显示目前连接到本活动的客户名。要去掉某参与者，在列表中选择它的名字，按下 Kick 按钮。要启动你自己的 HubNet 客户程序，按下 Local 按钮，当你调试活动时这个功能会很有用。“Reset”踢出所有登录客户，重新加载客户界面。

控制中心的底部显示参与者加入或离去的信息。要对所有参与者广播信息，单击最下面的编辑框，输入信息然后按下 Broadcast Message 。

问题及解决

我启动了一项 HubNet 活动，但当参与者打开 HubNet 客户程序时，我的活动没有列出。

在某些网络 HubNet 客户程序不能自动检测 HubNet 服务器。告诉参与者手工输入你的 HubNet 服务器的地址和端口号，这些信息在控制中心显示出来。

注意：技术细节如下。为了让客户能检测到服务器，他们之间必须有多播路由。不是所

有网络支持多播路由。特别是使用 IPSec 协议的网络一般不支持多播，很多虚拟专网 (VPNs) 使用 IPSec 协议。

当参与者试图连接到一个活动时，什么也没发生（客户程序好像挂起了，或者有条错误信息说没找到服务器）

如果计算机或网络有防火墙，可能会阻断 HubNet 服务器的通信。确保计算机和网络没有封闭 HubNet 服务器使用的端口（端口 9173–9180）

HubNet 客户视图是灰的

- 检查 HubNet 控制中心的"Mirror 2D view on clients" 选择框被选中
- 确保模型的 display 开关打开
- 如果在服务器上改变了视图大小，需要按下控制中心的"Reset"按钮，确保客户获得新的视图大小

HubNet 客户没有视图

有些活动在客户端没有视图。如果只想简单的加上视图，从 Tools 菜单选择"HubNet Client Editor"，加上类似的视图。确保客户登录之前按下"Reset"按钮。

无法退出 HubNet 客户

必须强迫客户程序退出。在 OS X 上选择 Apple 菜单的 Force Quit...，强行退出应用程序。在 Windows 按下 Ctrl-Alt-Delete 打开任务管理器，选择 HubNet Client 结束任务。

运行 HubNet 活动时计算机休眠了，唤醒后出错，HubNet 不再正常运行。

如果计算机休眠，HubNet 服务器停止工作。如果发生这种情况，退出 NetLogo 应用程序重新进入。改变计算机的设置使它不再休眠。

我的问题这儿没有

请发邮件到feedback@ccl.northwestern.edu.

已知问题

如果HubNet 出故障，请发邮件到bugs@ccl.northwestern.edu.

请注意：

- HubNet 没有在大量客户(多于 25)情况下进行全面测试，客户较多时可能会有一些未预料的结果
- 内存溢出问题未得到有效处理
- 发送大量绘图信息到客户需时较长.

- NetLogo 不能对付恶意客户攻击(对拒绝服务攻击比较脆弱)
- 当网络较慢或不可靠时，性能下降严重
- 如果使用无线网络或 LAN 子网，控制中心显示的 IP 地址可能不全
- 计算机 HubNet 仅在局域网进行了测试，没有在拨号连接或广域网进行测试.

计算器 HubNet

TI-Navigator 中的计算器 HubNet

TI-Navigator教室学习系统是为TI图形计算器设计的无线教室网络。TI-Navigator用户可以安装一个免费的NetLogo扩展包，实现与TI-Navigator的集成，使得计算器可以作为客户用于参与式仿真，就像计算机HubNet一样。该扩展包可以从Inquire Learning, LLC与TI的合作组织得到。Inquire Learning也提供计算器HubNet的支持、教学和专业开发材料。关于TI-Navigator的更多信息，访问TI网站<http://education.ti.com/navigator>。关于计算器 HubNet 扩 展 包 的 信 息 ， 请 联 系 Inquire Learning， 邮 件 地 址 是 calc-hubnet@inquirelearning.com ， 或 访 问 网 站 <http://www.inquirelearning.com/calc-hubnet.html>。

教师研讨会

要了解关于在教室中使用 NetLogo 和 HubNet 研讨会的信息，发邮件到 feedback@ccl.northwestern.edu.

HubNet 编程指南

要学习如何编写和修该HubNet活动，参见[HubNet Authoring Guide](#).

获得帮助

如果对计算机或计算器HubNet有什么问题，或者需要什么帮助，请发邮件到 feedback@ccl.northwestern.edu.

HubNet 编程指南（HubNet Authoring Guide）

本指南介绍如何修改已有 HubNet 活动代码以及如何编写自己的 HubNet 活动。本指南假设你熟悉如何运行 HubNet 活动，了解 NetLogo 代码及界面元素。关于 HubNet 更多的信息，参见 HubNet 指南。

- 一般信息
- HubNet 活动编程
 - 设置
 - 接收客户信息
 - 发送信息到客户
- 计算器 HubNet 信息
- 计算机 HubNet 信息
 - 如何制作客户界面
 - 客户上的视图更新
 - 客户视图上的点击
 - 客户上的绘图刷新

一般信息

本部分内容适用于计算机 HubNet，然而此处的许多代码经过小的修改后也可用于计算器 HubNet。

HubNet 活动编程

许多 HubNet 活动有一小部分代码完全相同，这部分代码是设置网络的代码和与客户发送、接收信息的代码。如果理解了这些代码，就能很容易的对已有活动做出修改，也是编写自己的活动的起点。我们提供了一个模板（Template）模型（在 HubNet Computer Activities → Code Examples），该模板包括大多数 HubNet 活动所需要的多数基本部件。该模板可以作为多数项目开发的起点。

设置（Setup）

要把一个NetLogo模型变成HubNet活动，首先需要初始化网络。在多数HubNet活动中，使用[startup](#)例程初始化网络。Startup是个特殊例程，打开任何模型时，NetLogo都试图运行该例程，因此它成为放置运行一次且仅一次代码的好地方（不管用户运行该模型多少次）。对HubNet，我们把初始化网络的命令放在startup中，因为一旦网络设置好了就不需再设置。首先用[hubnet-set-client-interface](#)指定客户类型，此处我们使用计算机客户：

```
hubnet-set-client-interface "COMPUTER" []
```

然后使用[hubnet-reset](#)初始化系统，它将请求客户输入用户名，并打开HubNet控制中心。NetLogo现在准备好开始监听客户消息。

现在网络完全设置好了，你不需担心再次调用[hubnet-set-client-interface](#)或[hubnet-reset](#)。看看模板模型的setup例程：

```
to setup
  cp
  cd
  clear-output
  ask turtles
  [
    set step-size 1
    hubnet-send user-id "step-size" step-size
  ]
end
```

该例程的多数部分看起来与其他setup一样，然而你应注意到它没有调用[clear-all](#)。在本模型以及模型库的大多数HubNet活动中，有一个表示登录客户的海龟种类，此处我们称之为students。当一个客户登录后，创建一个student，用海龟变量记录以后可能用到的所有信息。当设置活动时，不想要求每个客户退出、重新登录，因此不会杀死所有海龟，相反把所有变量设回初始值并通知客户我们所做的改变。

接收客户消息

在活动进行时需要在 HubNet 客户和服务器之间传输数据。多数 HubNet 活动在 go 循环中调用一个例程检查来自客户的新消息，此处称之为 listen-clients：

```
to listen-clients
  while [ hubnet-message-waiting? ]
  [
    hubnet-fetch-message
    ifelse hubnet-enter-message?
    [ create-new-student ]
    [
      ifelse hubnet-exit-message?
      [ remove-student ]
      [ execute-command hubnet-message-tag ]
    ]
  ]
end
```

只要队列中有消息，该循环就取出每一条消息。[hubnet-fetch-message](#)将队列中的下一条消息作为当前消息，设置报告器[hubnet-message-source](#), [hubnet-message-tag](#),

hubnet-message 为适当的值。当用户登录、退出，以及操作任一界面元素，如按下按钮、移动滑动条、在视图上单击等，客户程序就发送消息。我们依次处理每条消息，根据消息类型（进入、退出及其他）、hubnet-message-tag（界面元素的名字）、消息的hubnet-message-source（消息来源的客户名）决定采取的行动，

如果是enter(进入)消息，我们创建一个海龟，其user-id等于hubnet-message-source，就是进入活动的终端用户的名字，要保证它的唯一性。

```
to create-new-student
  create-students 1
  [
    set user-id hubnet-message-source
    set label user-id
    set step-size 1
    send-info-to-clients
  ]
end
```

此处我们将其他客户变量设为默认值，如果合适的话，将它们发送到客户。对客户程序界面上保存状态的每个元素，如滑动条、选择器、开关、输入框等，声明一个students-own变量，这些变量将成为服务器上的全局变量。重要的一点是要确保这些变量与客户程序可视值之间的同步。

当客户退出时，发送一条exit消息到服务器。这样就有机会清除为该客户保存的所有信息。此处仅仅请求对应的海龟死亡。

```
to remove-student
  ask students with [user-id = hubnet-message-source]
  [ die ]
end
```

所有其他消息都来自界面元素，通过hubnet-message-tag来识别元素，hubnet-message-tag 就是出现在客户界面上的元素名字。每当界面元素有所变化时就发送一条消息给服务器。除非保存界面状态当前值，否则在模型的其他部分无法访问界面，这就是我们为每个有状态的界面元素（滑动条、开关等）声明一个students-own变量的原因。当收到来自客户的消息时，将海龟变量设为消息内容：

```
if hubnet-message-tag = "step-size"
[
  ask students with [user-id = hubnet-message-source]
  [ set step-size hubnet-message ]
]
```

由于按钮没有关联数据，一般没有相应的海龟变量，相反它表示客户执行了一个动作。像常规按钮一样，一般每个按钮有一个相关例程，当接收到一条消息说明按下按钮时就调用该例程。一般该例程是一个海龟例程（尽管并非必须），即与消息源关联的student海龟能执行：

```
if command = "move left"
```

```
[ set heading 270
  fd 1 ]
```

向客户发送消息

前面已经提到，也可以向显示信息的任何界面元素发送值：监视器、滑动条、开关、选择器、输入框（注意绘图和视图是特例，有自己的部分）。有两个发送消息的原语[hubnet-send](#)和[hubnet-broadcast](#)。Broadcast向所有客户发送消息，send向指定的客户或选定的一组客户发送消息。前面说过，客户上的任何东西都不会自动更新。如果服务器上的某个值改变了，是你负责更新客户上的监视器。如果是你改变了与某个界面元素关联的服务器上的变量值，而该变化不是出自对界面元素消息的反应，你必须负责更新客户上的显示。例如假设客户上有一个名为step-size的滑动条和名为Step Size的监视器，需要像这样写代码：

```
if hubnet-message-tag = "step-size"
[
  ask student with [ user-id = hubnet-message-source ]
  [
    set step-size hubnet-message
    hubnet-send user-id "Step Size" step-size
  ]
]
```

可以发送任何类型的数据，包括数值型、字符串、列表、列表的列表，字符串列表。然而如果数据类型与接收数据的界面元素不匹配（例如发送字符串给滑动条），则消息被忽略。下面是一些不同类型数据的例子：

数据类型	hubnet-broadcast例子	hubnet-send 例子
number	hubnet-broadcast "A" 3.14	hubnet-send "jimmy" "A" 3.14
string	hubnet-broadcast "STR1" "HI THERE"	hubnet-send ["12" "15"] "STR1" "HI THERE"
list of numbers	hubnet-broadcast "L2" [1 2 3]	hubnet-send hubnet-message-source "L2" [1 2 3]
matrix of numbers	hubnet-broadcast "[A]" [[1 2] [3 4]]	hubnet-send "susie" "[A]" [[1 2] [3 4]]
list of strings (only for Computer HubNet)	hubnet-broadcast "user-names" [{"jimmy" "susie"} {"bob" "george"}]	hubnet-send "teacher" "user-names" [{"jimmy" "susie"} {"bob" "george"}]

例子

研究模型库“HubNet Computer Activities”和“HubNet Calculator Activities”部分的例子，看看这些原语是怎样在例程页使用的。Disease 是一个入门的好例子。

计算器 HubNet 信息

要获得为计算器HubNet客户编写hubNet活动的信息，与我们联系([contact us](#))

计算机 HubNet 信息

下面的信息只用于计算机 HubNet。

怎样制作客户界面

在Tools菜单打开HubNet客户编辑器(HubNet Client Editor)，可以添加任何按钮、滑动条、开关、监视器、绘图、选择器或注释，就像在界面页中一样。注意为每个部件输入的信息与界面面板中略有不同。客户上的部件与模型的交互方式有所不同，部件将消息发送回服务器，由模型决定它的影响，而不是直接与命令和报告器相连。所有部件有一个标志(tag)，唯一标志该部件的名字。当服务器接收到来自部件的消息时，通过[hubnet-message-tag](#)得到部件的标志。

例如，有按钮“move left”，滑动条“step-size”，开关“all-in-one-step？”，监视器“Location：”，这些界面元素的标志如下：

界面元素	tag
move left	move left
step-size	step-size
all-in-one-step？	all-in-one-step？
Location:	Location:

注意一个名字只能有一个部件。如果客户界面的多个部件有相同的标志，将会导致不可预料的结果，因为不知道向哪个元素发送消息。

客户上的视图更新

在客户视图中显示世界的最简单方式是使用视图镜像(view mirroring)。当view mirroring 打开时(通过 HubNet 控制中心激活)，客户视图自动更新，反映世界状态。所

有客户拥有与服务器一样的视图。视图大约每秒更新 5 次，这意味着要发送大量信息到客户。如果发生性能下降，可以试试使用 no-display，或减少更新频率。

要获得对视图更新更多的控制，可以使用 [hubnet-broadcast-view](#) 和 [hubnet-send-view](#)，显式地分别对所有客户或特定客户发送视图。然而注意视图镜像仅更新自上次更新以来有所变化的信息，而broadcast每次发送所有的世界信息，这意味消息较大(很大)，因此更新缓慢。

如果客户没有视图，或控制中心的 Mirror 2D View on Clients 选择框未选中，则不发送视图消息。

注意： hubnet-broadcast-view 和 hubnet-send-view are experimental 原语是实验性的，将来版本可能会变化。

注意：NetLogo 的一些视图功能尚未在 HubNet 客户中实现，例如视图回绕和观察者视角 (View Wrapping and Observer Perspectives)

客户视图上的点击

如果客户包含视图，则用户在视图上点击时都会向服务器发送消息。消息的标志是 "View"，消息包括由 x 和 y 坐标构成的一个两项列表。例如要把客户视图上点击的瓦片变成红色，使用下面的代码：

```
if hubnet-message-tag = "View"
[
  ask patches with [ pxcor = (round item 0 hubnet-message) and
                      pycor = (round item 1 hubnet-message) ]
  [ set pcolor red ]
]
```

客户上的绘图更新

如果绘图镜像 (plot mirroring) 已激活（在 HubNet 控制中心），并且客户上有名字完全相同的绘图，当 NetLogo 绘图变化时，关于该变化的消息就发送到客户，使得客户视图发生同样的变化。例如假设 HubNet 模型在 NetLogo 和客户上都有个名为 Milk Supply 的绘图，在 NetLogo 中 Milk Supply 是当前视图，在命令中心输入：

```
plot 5
```

引起将一条消息发送到所有客户，告诉他们在绘图的下个位置、y 为 5 处画一个点。注意，如果一次执行很多绘图动作，会有大量的绘图消息发送到客户。

日志 (Logging)

NetLogo 日志工具提供给研究人员，用来记录学生的行为，以备日后分析。

NetLogo 的日志功能启动后，对用户是不可见的。研究人员通过配置文件选择要记录的事件类型。

NetLogo 使用 Log4j 包进行记录。如果你有使用 Log4j 的经验，会发现 NetLogo 日志功能很相似。

开始记录

取决于所用的操作系统.

Mac OS X 或 Windows

在 NetLogo 目录有个专门的日志启动程序，叫做 NetLogo Logging，双击该图标。

在Windows，NetLogo 目录在C:\Program Files里，除非在安装时选择了其他目录。

Linux 及其他

要激活日志，调用netlogo.sh 脚本如下：

```
netlogo.sh --logging netlogo_logging.xml
```

也可以修改脚本包含这些标志，或复制脚本对拷贝进行修改。

可以用任何有效的 log4j xml 配置文件替换 netlogo_logging.xml，后面详细讨论。

使用日志工具

当 NetLogo 启动时，要求输入用户名，该用户名出现在本次会话所有记录中。

日志存在哪了

日志存储在操作系统指定的临时目录。在多数Unix类操作系统(包括MacOS)是指/tmp，在Windows XP是指c:\Documents and Settings\<user>\Local Settings\Temp，其中 <user> 是 登录 用户， 在 Windows Vista 是 c:\Users\<user>\AppData\Local\Temp 。

有两个命令帮你管理日志。`_zip-log-files filename`收集临时目录中所有日

志，打包到一个指定位置的zip文件。执行该命令后，原日志没有删除，要删除的话显式使用`_delete-log-files`。

下表列出所有记录器的名字，记录的事件类型、优先级，并提供了一个使用XMLLayout的样本输出。所有记录器都能在org.nlogo.api.Logger找到。当在配置文件中引用记录器时，要使用完整限定名。例如，记录器 GLOBALS 实际上是 org.nlogo.api.Logger.GLOBALS。

记录器	事件	优先级	例子
GLOBALS	a global variable changes	info, debug	<pre><event logger="org.nlogo.api.Logger.GLOBALS" timestamp="1177341065988" level="INFO" type="globals"> <name>F00</name> <value>51.0</value> </event></pre>
GREENS	sliders, switches, choosers, input boxes are changed through the interface	info	<pre><event logger="org.nlogo.api.Logger.GREENS" timestamp="1177341065988" level="INFO" type="slider"> <action>changed</action> <name>foo</name> <value>51.0</value> <parameters> <min>0.0</min> <max>100.0</max> <inc>1.0</inc> </parameters> </event></pre>
CODE	code is compiled, including: command center, procedures tab, slider bounds, and buttons	info	<pre><event logger="org.nlogo.api.Logger.CODE" timestamp="1177341072208" level="INFO" type="command center"> <action>compiled</action> <code>crt 1</code> <agentType>0</agentType> <errorMessage>success</errorMessage> </event></pre>

WIDGETS	a widget is added or removed from the interface	info	<pre><event logger="org.nlogo.api.Logger.WIDGETS" timestamp="1177341058351" level="INFO" type="slider"> <name></name> <action>added</action> </event></pre>
BUTTONS	a button is pressed or released	info	<pre><event logger="org.nlogo.api.Logger.BUTTONS" timestamp="1177341053679" level="INFO" type="button"> <name>show 1</name> <action>released</action> <releaseType>once</releaseType> </event></pre>
SPEED_SLIDER	the speed slider changes	info	<pre><event logger="org.nlogo.api.Logger.SPEED" timestamp="1177341042202" level="INFO" type="speed"> <value>0.0</value> </event></pre>
TURTLES	turtles die or are born	info	<pre><event logger="org.nlogo.api.Logger.TURTLES" timestamp="1177341094342" level="INFO" type="turtle"> <name>turtle 1</name> <action>born</action> <breed>TURTLES</breed> </event></pre>
LINKS	links die or are born	info	<pre><event logger="org.nlogo.api.Logger.LINKS" timestamp="1177341094347" level="INFO" type="link"> <name>link 0 1</name> <action>born</action> <breed>LINKS</breed> </event></pre>



如何配置日志输出

默认的日志输出配置(`netlogo_logging.xml`) 与下面的类似:

NetLogo 定义了 8 个记录器，它们都直接继承自根记录器 (root logger)，意味着除非你在配置文件里显式设置它们的特性(appender, layout, output level)，它们将从根记录器继承。默认配置中，根设置为 level 为 INFO，appender 为 `org.nlogo.api.XMLFileAppender`，layout 为 `org.nlogo.api.XMLLayout`。这些一起产生了一个由 `netlogo_logging.dtd` 定义的格式良好的 XML 文件，而 `netlogo_logging.dtd` 基于 `log4j.dtd`。如果 appender 是一个 `FileAppender` (包括 `XMLFileAppender`)，每次用户打开模型时启动一个新文件。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration debug="false"
xmlns:log4j='http://jakarta.apache.org/log4j/'>

    <appender name="A1" class="org.nlogo.api.XMLFileAppender">
        <layout class="org.nlogo.api.XMLLayout"/>
    </appender>

    <category name="org.nlogo.api.Logger.WIDGETS">
        <priority value="off" />
    </category>

    <category name="org.nlogo.api.Logger.TURTLES">
        <priority value="off" />
    </category>

    <category name="org.nlogo.api.Logger.LINKS">
        <priority value="off" />
    </category>

    <root>
        <priority value ="info" />
        <appender-ref ref="A1" />
    </root>

</log4j:configuration>
```

该配置首先定义一个名为“`A1`”的 appender，该 appender 的类型是使用 `XMLLayout` 的 `XMLFileAppender`。该 appender 定义日志数据的去向，此处数据的去向是一个文件。事实上

如果给了 NetLogo 一个 FileAppender，每次用户打开一个新模型时就启动一个新文件。 XMLFileAppender 还做一些格式化工作，将一些标题（header）写入文件。Layout 定义如何写每条信息。如果不是高级用户，不要改变（或担心）appender 或 layout。

在配置的尾部注意根记录器（root logger）的定义。其他所有记录器都源于根记录器，也继承了根的特性，除非显式设定。本例非常简单，设置 appender A1 为根（及所有其他）记录器的默认 appender，指定默认优先级为“INFO”。优先级高于或等于 INFO 的消息被记录，低于的不记录。注意 NetLogo 总是在 INFO 级记录，只有一个例外，即设置但不改变全局变量的值是在 DEBUG 级记录，意味着默认时这些消息不显示，因为 DEBUG 低于 INFO。配置文件的其他部分是一些记录器重载根记录器的特性（或配置文件知道的 categories，它们在本文档是同义词），具体内容是默认关闭 WIDGET, TURTLES, LINKS 记录器。要激活他们，你可以将优先级从 off 改为 info，如下：

```
<category name="org.nlogo.api.Logger.TURTLES">
  <priority value="info" />
</category>
```

或者简单的将这些对 category 的引用去掉，因为它们没有其他作用。

高级配置

这些仅是对NetLogo日志配置文件的基本介绍。使用log4j框架，有许多配置选项，参见[log4j documentation](#)。

控制指南（Controlling Guide）

Java 程序能够调用和控制 NetLogo，例如你可以从一个自动化模型运行的小程序里调用 NetLogo。

此部分用户手册为 Java 程序员介绍这一功能，假设你了解 Java 语言和相关工具。

注意：控制功能是“试验性”的，可能会改变和增强。使用这些功能的代码在使用以后的 NetLogo 版本时可能需要修改。

- 为 NetLogo 启动 Java VM
- 例子（带 GUI）
- 例子(headless)
- 行为空间
- 其他选项
- 结论

[NetLogo API Specification](#) 包含更深的细节。

为 NetLogo 启动 Java 虚拟机

NetLogo 对 Java 虚拟机有一些假定，因此启动时要给虚拟机一些参数。

对 GUI 和非 GUI 都适用的推荐选项

`-server`

为达到最高性能使用 server VM

`-Xmx1024m`

为 Java VM 堆最多分配 1G 内存，在运行一些模型时需要增加这个数值。

对 GUI 的一些附加推荐选项

`-XX:MaxPermSize=128m`

当重复编译长代码模型时，防止 Java 内存不够。

`-Djava.ext.dir=`

忽略任何系统内置库，避免与其他版本的 JOGL 冲突。可能需要令该项为空，或当使用 Java VM 扩展时指定你的内部库。

`-Djava.library.path=./lib`

Mac 或 Windows 不需要，其他操作系统如 Linux 可能需要。确保 NetLogo 能找到 JOGL 内部库和其他扩展。如何不是在 NetLogo 顶层目录启动 VM，需要改变 `./lib` 指向 NetLogo 安装时的 lib 子目录。

当前工作目录

NetLogo 应用程序假设启动时的当前工作目录为 NetLogo 安装所在的顶层目录。

例子（有 GUI）

下面是一个很小但完整的一个程序，它启动完全的 NetLogo 应用程序，打开一个模型，移动滑动条，设置随机数种子，运行模型 50 个滴答，然后打印结果：

```
import org.nlogo.app.App;
import java.awt.EventQueue;

public class Example1 {
    public static void main(String[] argv) {
        App.main(argv);
        try {
            EventQueue.invokeLater
                ( new Runnable()
                    { public void run() {
                        try {
                            App.app.open
                                ("models/Sample Models/Earth Science/"
                                 + "Fire.nlogo");
                        }
                        catch( java.io.IOException ex ) {
                            ex.printStackTrace();
                        }
                    } } );
            App.app.command("set density 62");
            App.app.command("random-seed 0");
            App.app.command("setup");
            App.app.command("repeat 50 [ go ]");
            System.out.println
                (App.app.report("burned-trees"));
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

为了程序能编译和运行，NetLogo.jar（来自NetLogo发行包）必须在classpath，另外lib目录（也来自NetLogo发行包）也要在同一位置，lib目录包括NetLogo.jar要用到的其他库。

注意使用EventQueue. invokeAndWait确保在正确的线程中调用方法，因为App类中的多数方法只能在某些特定的线程中调用。这些方法多数只能从AWT event queue线程中调用，但少数方法，如command()，只能从不是AWT event queue线程的其他线程（例如此例中的main线程）中调用。

此处不再继续深入讨论这个例子，相反请参考[NetLogo API Specification](#)，该规范说明了例子中涉及到的类的定义和方法，这些类还有一些其他方法可用。

例子 (headless)

下面的代码与前面的例子很相似，只是用HeadlessWorkspace类实例代替了App 的静态方法。

```
import org.nlogo.headless.HeadlessWorkspace;

public class Example2 {
    public static void main(String[] argv) {
        HeadlessWorkspace workspace =
            new HeadlessWorkspace();
        try {
            workspace.open(
                ("models/Sample Models/Earth Science/" +
                 + "Fire.nlogo");
            workspace.command("set density 62");
            workspace.command("random-seed 0");
            workspace.command("setup");
            workspace.command("repeat 50 [ go ]");
            System.out.println(
                (workspace.report("burned-trees")));
            workspace.dispose();
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

为编译和运行该程序，NetLogo. jar或 NetLogoLite. jar（来自NetLogo发行包）必须在classpath。（后面的jar较小，但只能进行headless操作，没有完全的GUI操作）。Lib目录包含其他库，也必须在同一目录。在没有图形显示的环境下运行时，系统特性java.awt.headless 必须为 true，强迫 Java 以 headless 模式运行，HeadlessWorkspace自动设置该特性。

由于没有GUI，则向命令中心或输出区发送输出的原语转到标准输出。仍然可以用[export-world](#)保存模型状态，[export-view](#)将当前（否则看不到）2 维视图的快照写到图像文件。report()方法从模型得到结果送到Java代码。

由[export-world](#)产生的文件包含所有绘图内容，也可以使用[export-plot](#)输出单个绘图的内容。

可以生成多个HeadlessWorkspace实例，他们在单独的线程中独立运行，互不干扰。

当以 headless 运行时，有一些限制：

- [movie-*](#)原语不可用；试图使用他们造成Java例外
- 请求用户输入的原语[user-*](#)，例如user-yes-or-no，造成Java例外.

[NetLogo API Specification](#) 包含更多细节。

行为空间 (BehaviorSpace)

控制 API 支持以 headless 模式运行 BehaviorSpace 实验。（它不支持 GUI 模式运行 BehaviorSpace，尽管你可以写出类似 BehaviorSpace 的 Java 代码，运行类似 BehaviorSpace 的实验）。

注意不需要使用API以headless运行BehaviorSpace。可以使用命令行直接运行Headless BehaviorSpace，根本不需编程。有关介绍见[BehaviorSpace Guide](#)。

多数情况下，有命令行支持就足够了，不需使用 API。然而可以使用 API 增加灵活性。 HeadlessWorkspace 有四个方法来运行实验：三个 runExperiment 变种，加一个 runExperimentFromModel。

当实验设置已经存储在模型文件中时，使用 runExperimentFromModel .

当实验设置存储在单独的XML文件中，而不是与模型共同存储时，可以使用两个带有File 参数的runExperiment。如果文件只包含一个实验设置，只需要传送File对象，如果包含多个实验设置，则还必须传送一个字符串对象，该字符串保存试验名。

只有一个String参数（还有一个指定输出格式的参数）的runExperiment，用来直接将 XML文件作为实验设置。

所有这些方法将PrintWriter作为结果输出地。如果想输出到标准输出，可以传入新的 java. io. PrintWriter(System.out) 。

行为空间指南[BehaviorSpace Guide](#) 解释了如何用XML指定实验设置。

[NetLogo API Specification](#) 包含关于 HeadlessWorkspace 类和方法的更多细节。

其他选项

当使用App类控制NetLogo时，整个NetLogo应用程序都出现，包括标签页、菜单等。这种方式适合控制或“执行”（“scripting”）一个NetLogo模型。不适合将NetLogo模型嵌入到一个大的应用程序。

我们还有一套类似的API将NetLogo部分嵌入，例如只有标签页（而不是整个窗口），或只有界面页的内容。目前这些API还没有文档，如果你感兴趣，联系

feedback@ccl.northwestern.edu。

结论

记着参考[NetLogo API Specification](#)，获得类和方法的所有细节。

前面说过控制功能是实验性的，最初的API不会包含你所希望的所有东西。有些功能已实现，但还没有文档。因此如果没有发现所需的功能，请与我们联系，我们能帮你做你想做的事。别犹豫，联系我们feedback@ccl.northwestern.edu，我们可能找到所需的东西，提供那些编写不太详细文档的附加信息。

Mathematica 连接 (Mathematica Link)

它是什么？

NetLogo-Mathematica 连接 (link) 给用户提供了 NetLogo 和 Mathematica 之间易用、实时的连接。将这两个软件连在一起后，可以给用户提供高度交互性、文档自动化的工作流，这一点仅靠其中一个软件是不能提供的。

Mathematica 有许多工具是基于主体的建模人员需要的，包括高级输入能力、统计功能、数据可视化、文档创建等。使用 NetLogo-Mathematica 连接，就可以与 NetLogo 一起使用这些工具。

因为所有的 Mathematica 文档或 notebook 包含注释、代码、图像、注解 (annotation) 和交互对象，NetLogo 和 Mathematica 的集成成为学生和研究人员探索复杂模型提供了更完全的解决方案。

该连接的基本功能与 NetLogo 控制 API 很像，可以加载模型、执行命令、从 NetLogo 返回数据。但不同的是，控制 API 是基于 Java 的，而与该连接的交互是解释的，这使得不仅可以快速设计自定制的类似 BehaviorSpace 的实验，还可以在调试模型时作为 NetLogo 的伴侣。

关于 Mathematica 的更多信息，请访问 [Wolfram Research web site](#).

我能用它做什么？

下面是使用 Mathematica-NetLogo 连接所能做的事情的一些例子。

- 以双向无缝数据转换方式实时分析你的模型
- 开发高质量的模型数据自定制可视化
- 在巨大的多维参数空间中收集详细的仿真数据
- 快速开发交互式界面，探索模型行为
- 使用内置函数直接访问瓦片和网络数据

使用 NetLogo-Mathematica 连接

本部分简单介绍如何使用 NetLogo-Mathematica 连接，告诉你怎样加载 NetLogo-Mathematica 连接包、启动 NetLogo、执行命令、从 NetLogo 获取数据。

加载包：一旦安装了 NetLogo-Mathematica 连接，可以在 Mathematica notebook 输入下面的命令加载该包：

```
<<NetLogo`
```

从 Mathematica 启动 NetLogo：要在 Mathematica 启动 NetLogo 会话，需要在 notebook

里输入下面的命令

```
NLStart["your netlogo path"];
```

其中 “*your netlogo path*” 是 NetLogo 所在目录，一般在 Macintosh 计算机是 “/Applications/NetLogo 4.0/”

加载模型：要加载模型，必须指定模型的全路径。例如要加载 Forest Fire 模型，并且使用典型的 Macintosh 安装位置给出路径。

```
NLLoadModel["/Applications/NetLogo 4.0/models/Sample Models/Earth  
Science/Fire.nlogo"];
```

执行NetLogo命令：给NLCommand[]传入一个命令字符串执行命令。NLCommand[]函数自动拼接Mathematica常用数据类型，成为NetLogo可用的字符串。下面的命令使用单个字符串设置density，或使用一个Mathematica定义的变量myDensity设置density。

```
NLCommand["set density 50"];  
myDensity = 60;  
NLCommand["set density", myDensity];
```

从NetLogo返回信息：NetLogo数据可以使用NLReport[]返回Mathematica。数据类型包括数值、字符串、布尔值、列表。

```
NLReport["count turtles"];  
NLReport["[(list pxcor pycor)] of n-of 10 patches"]
```

更多信息参加NetLogo 所带的NetLogo–Mathematica Tutorial notebook。该notebook 带你体会使用该连接的整个过程，其中有许多例子。如果没有Mathematica，但却考虑使用此连接，可以下载教学pdf文档 ([download a PDF](#))。

安装

NetLogo–Mathematica 连接需要 NetLogo 4.0 和 Mathematica 6.0 以上版本。安装 NetLogo–Mathematica 连接的步骤如下：

- 到 Mathematica 的菜单
- 单击 File，选择 Install...
- 在 Install Mathematica 对话框
- 选择 Package for Type of item to install
- 单击 Source, 选择 From file...
- 在文件浏览器，找到 NetLogo 安装目录，
- 单击 Mathematica Link 子文件夹，选择 NetLogo.m.
- 在 Install Name 输入 NetLogo.

可以将 NetLogo 连接安装到用户级目录或系统级目录。如果安装在用户级目录，其他用户也必须自己安装才能使用。如果你无权修改用户目录之外的目录就选择这种方式，否则将连接安装到系统级目录。

已知问题

- 不退出 J/Link (Java-Mathematica link) , NetLogo 会话不能完全退出。这可能干扰其他使用 J/Link 的包。这个问题在以后的版本中解决。
- 如果 NetLogo-Mathematica 连接所加载的模型使用了 NetLogo 扩展包，则该扩展必须在扩展自身的目录，如果扩展在应用程序级目录则找不到。这个问题在以后的版本中解决
- 对NetLogo的调用,如NLCommand[]和 NLReport[]不能中途退出 (abort)。

致谢

NetLogo-Mathematica 连接的主要开发人员是 Eytan Bakshy。

在学术出版物上引用此包的格式：Bakshy, E., Wilensky, U. (2007). NetLogo-Mathematica Link. <http://ccl.northwestern.edu/netlogo/mathematica.html>. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

扩展指南（Extensions Guide）

NetLogo 允许用户用 Java 编写新的命令和报告器，然后在模型中使用。用户手册本部分介绍这一功能。

第一部分介绍当你写完了扩展或别人给你提供了扩展后，如何在模型中使用它。

第二部分提供给Java程序员，介绍如何使用NetLogo 扩展API（[the NetLogo Extension API.](#)）编写自己的扩展包

- 使用扩展包
- 编写扩展包

[NetLogo Extension API Specification](#) 包括更多的细节。

使用扩展

要在模型里使用扩展包，需要在声明任何种类或变量之前，在例程页的首部加上关键词[extensions](#)。

在[extensions](#)后面的方括号里列出扩展包的名字，例如：

`extensions [sound speech]`

关键词[extensions](#)告诉NetLogo去寻找、打开指定的扩展包，使得扩展包中的命令和报告器可用于当前模型，就像使用内置NetLogo原语一样使用这些命令和报告器。

扩展包的存储位置

NetLogo 在几个地方寻找扩展包：

1. 当前模型目录
2. NetLogo应用程序的extensions 文件夹

每个NetLogo扩展包由同名的一个文件夹组成，文件夹名字全小写。该文件夹必须包括与文件夹同名的一个JAR文件。例如扩展包sound就存储在名为sound的文件夹，其中有个文件是sound.jar。关于扩展包文件夹所包括内容的更多信息，参见手册的编写扩展包部分（[Writing Extensions](#)）。

要编写一个对所有模型都可用的扩展包，要把这个扩展包的文件夹放在NetLogo的extensions目录。或者将扩展包文件夹放在模型同一目录。

有些扩展包需要其他文件，这些文件要与 JAR 文件一起放在扩展包目录。该文件夹还可能有其他文件，如文档和模型例子等。

Applets

保存为 applets 的模型（使用 NetLogo 的 File 菜单的“Save as Applet”）可以使用扩展包。扩展包必须与模型放在同一目录下。然而 applets 仍然不能使用需要附加外部 jar 的扩展。

编写扩展

假设你有 Java 编程经验。

总结

NetLogo 扩展由一个文件夹组成，该文件夹包括以下内容：

必须的：

- 与扩展包同名的 JAR 文件，包括以下内容：
 - 一个或多个实现[org.nlogo.api.Primitive](#)的类，
 - 实现[org.nlogo.api.ClassManager](#)的一个主类，和
 - 一个 NetLogo 扩展清单文件（manifest file），有下面四个标志(tag):
 - Manifest-Version, 总是 1.0
 - Extension-Name, 扩展包的名字
 - Class-Manager, 实现org.nlogo.api.ClassManager的一个完全(fully-qualified) 的类名
 - NetLogo-Extension-API-Version, JAR预期的NetLogo Extension API 版本。如果NetLogo的Extension API 版本不同，则打开扩展包时出现警告信息。要得知NetLogo所支持的Extension API 版本,在 "Help"菜单选择"About NetLogo" ,然后单击System页。或者以参数--extension-api-version启动NetLogo. jar。

可选的：

- 演示扩展包如何使用的一个或多个 NetLogo 模型.
- 扩展包所需的一个或多个 JAR 文件
- 存储所需的内部库（native librarie）的lib目录
- 包括模型源代码的 src 目录
- 文档.

要创建自己的扩展包，必须在 class 路径中包括 NetLogo. jar.

例子

NetLogo带有几个扩展包的例子，有全部Java源代码。还有一些可以从[here](#)下载。

教学

让我们写一个扩展包，它只提供一个报告器first-n-integers 。
first-n-integers 有一个输入数值型参数n, 返回一个包含整数 0 到n-1 的列表。（当然在 NetLogo很容易做到这一点，此处只是一个例子）

1. 创建扩展文件夹

因为扩展是一个包含几项的文件夹，首先需要创建一个文件夹。本例中称为example 。我们在该文件夹中做所有的工作。还要创建一个src子文件夹保存Java代码，一个classes 子文件夹保存编译类。

2. 编写原语

原语是作为一个或多个Java类实现的。这些类的. java文件放在src子文件夹。

命令执行一个动作，报告器返回一个值。要创建一个新的命令或报告器，需要创建一个实现接口[org.nlogo.api.Command](#) 或 [org.nlogo.api.Reporter](#)的类，这两个接口是[org.nlogo.api.Primitive](#) 的 继承 。多数情况需要继承抽象类[org.nlogo.api.DefaultReporter](#)或[org.nlogo.api.DefaultCommand](#) 。

DefaultReporter 需要实现：

```
Object report (Argument args[], Context context)
    throws ExtensionException;
```

因为我们的报告器有一个参数，还要实现：

```
Syntax getSyntax();
```

下面是我们报告器的实现，它在一个src/IntegerList. java 文件中：

```
import org.nlogo.api.*;

public class IntegerList extends DefaultReporter
{
    // take one number as input, report a list
    public Syntax getSyntax() {
        return Syntax.reporterSyntax(
            new int[] {Syntax.TYPE_NUMBER}, Syntax.TYPE_LIST
        );
    }
}
```

```

}

public Object report(Argument args[], Context context)
    throws ExtensionException
{
    // create a NetLogo list for the result
    LogoList list = new LogoList();

    int n ;
    // use typesafe helper method from
    // org.nlogo.api.Argument to access argument
    try
    {
        n = args[0].getIntValue();
    }
    catch( LogoException e )
    {
        throw new ExtensionException( e.getMessage() );
    }

    if (n < 0) {
        // signals a NetLogo runtime error to the modeler
        throw new ExtensionException
            ("input must be positive");
    }

    // populate the list
    // note that we use Double objects; NetLogo numbers
    // are always doubles
    for (int i = 0; i < n; i++) {
        list.add(new Double(i));
    }
    return list;
}
}

```

注意：

- 注意列表中的数值对象是 Double，不是 Integer。作为 NetLogo 数值使用的必须是 Double 类型，即使数值没有分数部分。
- 要访问参数，使用[org.nlogo.api.Argument](#)的类型安全帮助方法，比如 `getDoubleValue()`
- 抛出一个 [org.nlogo.api.ExtensionException](#) 表示 NetLogo 运行时错误

命令与报告器类似，只是报告器实现 [Object report\(...\)](#) 而命令实现 [void](#)

[perform\(...\).](#)

2. 编写 ClassManager

每个扩展除了包括任意数量的命令或报告器类外，还必须包括一个实现 [org.nlogo.api.ClassManager](#) 接口的类。ClassManager告诉NetLogo扩展包有哪些原语。简单情况下，继承抽象类[org.nlogo.api.DefaultClassManager](#)，它提供了ClassManager方法的空实现。

下面是我们的扩展例子的类管理器src/SampleExtension.java:

```
import org.nlogo.api.*;

public class SampleExtension extends DefaultClassManager {
    public void load(PrimitiveManager primitiveManager) {
        primitiveManager.addPrimitive
            ("first-n-integers", new IntegerList());
    }
}
```

`addPrimitive()` 告诉NetLogo我们的报告器存在，它的名字是什么。

3. 编写清单文件 (Manifest)

扩展必须包括一个清单文件。它是一个文本文件，告诉NetLogo扩展的名字以及 ClassManager的位置。

`manifest` 必须包括三个标志：

- Extension-Name，扩展的名字。
- Class-Manager，实现org.nlogo.api.ClassManager类的全名
- NetLogo-Extension-API-Version, JAR预期的NetLogo Extension API 版本。如果 NetLogo的Extension API 版本不同，则打开扩展包时出现警告信息。要得知NetLogo 支持的Extension API 版本，在"Help"菜单选择"About NetLogo"，然后单击System页。或者以参数--extension-api-version启动NetLogo.jar。

下面是我们扩展例子的清单文件manifest.txt:

```
Manifest-Version: 1.0
Extension-Name: example
Class-Manager: SampleExtension
NetLogo-Extension-API-Version: 4.0
```

`NetLogo-Extension-API-Version` 行应该与你使用的 NetLogo Extension API 实际版本对应。

确保最后一行以新行字符结束。

4. 创建 JAR

要创建扩展的 JAR 文件，首先使用命令行或 IDE 像一般情况一样编译你的类。

重要：编译时必须将 NetLogo.jar (NetLogo 分发包中有) 加到 classpath。

下面是一个在命令行编译扩展的例子：

```
$ mkdir -p classes      # create the classes subfolder if it does not exist
$ javac -classpath NetLogo.jar -d classes src/IntegerList.java
src/SampleExtension.java
```

你需要改变 classpath 参数指向 NetLogo 安装处的 NetLogo.jar 文件。这个命令行将编译.java 文件，将.class 文件存到 classes 子文件夹。

然后创建包含生成的 class 文件和清单文件的 JAR，例如：

```
$ jar cvfm example.jar manifest.txt -C classes .
```

关于清单文件、JAR文件和Java工具的信息，访问 java.sun.com。

5. 在模型中使用你的扩展

要使用例子扩展，把 example 文件夹放到 NetLogo 的 extensions 目录，或者放到使用该扩展的模型命令。在例程页的顶部写：

extensions [example]

现在可以像使用内置报告器一样使用 example:first-n-integers。例如选择界面页，在命令行输入：

```
observer> show example:first-n-integers 5
observer: [0 1 2 3 4]
```

扩展开发提示

实例化

当使用扩展的模型加载时，类管理器实例化。

当使用扩展命令或报告器的 NetLogo 代码编译时，这些命令和报告器对象实例化。

Classpath

编译时别忘记在类路径中包括 NetLogo.jar，这是编写扩展的新手最常犯的错误。（如果编译器找不到 NetLogo.jar，会出现 classes in the org.nlogo.api package not being found 的错误）

调试扩展

NetLogo 有一些特殊原语帮你开发、调试扩展。它们是实验性的，以后可能有变化。（这

是它们名字有下划线的原因)

- `print __dump-extensions` 打印所加载的扩展包的信息
- `print __dump-extension-prims` 打印所加载的扩展包原语信息
- `__reload-extensions` 强制 NetLogo 下次编译模型时重新加载所有扩展。如果没有这条命令, 当你改变扩展包JAR后, 只有再次打开模型或重启NetLogo才会起作用。

第三方 JAR

如果你的扩展需要另外一个 JAR 中的代码, 将 JAR 复制到你的扩展目录。当输入一个扩展时, NetLogo 使该文件夹下的所有 JAR 对扩展可用。

如果计划将扩展分发给其他 NetLogo 用户, 确保提供安装说明。

支持 Java 旧版本

NetLogo 需要 Java 1.4.1 以上。如果希望你的扩展包所有 NetLogo 用户都能用, 你的扩展包应该支持 Java 1.4.1。

实现这一点的最容易方式是使用 1.4.1 JDK 进行开发。

使用 Java 1.5 或 1.6 为 1.4 做开发也是可能的, 但需要做以下两件事情:

- Javac (或IDE) 使用 `-target 1.4` 选项, 告诉新编译器忽略与旧版本不兼容的类文件。这保证你的代码不使用任何 1.5 或 1.6 独具的特点
- Javac (或IDE) 使用 `-bootclasspath` 选项, 使用 1.4 Java类库编译 (注意无论如何需要安装JDK 1.4)。这保证你的代码不使用任何 1.5 或 1.6 独具的Java API调用。

结论

别忘了参考[NetLogo API Specification](#), 获得关于这些类、接口、方法的详细信息。

注意建模人员没有任何方法获得扩展包所提供的命令和报告器的列表清单, 因此提供合适的文档很重要。

扩展功能还没有完成。API不能满足你所有希望, 有些工具还没有文档。如果你没有发现所需的功能, 请告诉我们。别犹豫, 把问题告诉我们feedback@ccl.northwestern.edu, 我们可能会发现工作方向或为你提供文档欠缺处的附加信息。

API 用户的意见也帮我们为未来版本正确定位。我们致力于让 NetLogo 具有灵活性和扩展性, 非常欢迎你的反馈。

数 组 与 表 扩 展 (Array and Table Extensions)

这些扩展为 NetLogo 增加两个数据结构：数组和哈希表。

何时使用

原则上凡是使用数组和哈希表的地方只用列表也能实现。但出于速度考虑，有时可以使用数组或哈希表来代替列表。这三种数据结构（列表、数组和哈希表）具有不同的性能特点，如果选择的数据结构合适，模型运行的更快一些。

当数值集合的规模固定时，使用数组较好。如果知道数值项所在的位置，可以快速的直接存取或修改该项。

当需要将一个值与另一个值关联时，使用哈希表较好。例如，可以创建单词和释义的哈希表，然后就可以在表中查找任一单词的释义。此处单词是键（key），可以容易获取任何键对应的值（value），但反之不行。

如何使用

两个扩展包都预安装了。

要在模型中使用数组扩展，在例程页的首部增加一行：

`extensions [array]`

要在模型中使用哈希表扩展，在例程页的首部增加一行：

`extensions [table]`

要在同一模型中使用这两个扩展，在例程页的首部增加一行：

`extensions [array table]`

如果模型已经使用了其他扩展，则已有`extensions`行，因此只需将`array` 或/和`table`增加到该行的列表中。

关于使用NetLogo扩展的更多信息，参见扩展指南 ([Extensions Guide](#))

哈希表键的限制

哈希表的键只能是字符串、数值、布尔值或列表。（列表可以是嵌套列表，只要具体项是字符串、数值或布尔值即可）

数组例子

```

let a array:from-list n-values 5 [0]
print a
=> {{array: 0 0 0 0 0}}
print array:length a
=> 5
foreach n-values 5 [?] [ array:set a ? ? * ? ]
print a
=> {{array: 0 1 4 9 16}}
print array:item a 0
=> 0
print array:item a 3
=> 9
array:set a 3 50
print a
=> {{array: 0 1 4 50 16}}

```

哈希表例子

```

let dict table:make
table:put dict "turtle" "cute"
table:put dict "bunny" "cutest"
print dict
=> {{table: "turtle" -> "cute", "bunny" -> "cutest" }}
print table:length dict
=> 2
print table:get dict "turtle"
=> "cute"
print table:get dict "leopard"
=> 0
print table:keys dict
=> ["turtle" "bunny"]

```

已知问题

当输出NetLogo世界时（使用[export-world](#)命令或Export World菜单项），数组和哈希表是按值（"by value"）输出的。这意味着如果同一个数组或哈希表存储在多个位置，当输出或重新输入时，不同的数组或哈希表出现在原来的各个位置上。初始时它们都包含同样的值，但每个拷贝可以单独改变，其他的不受影响。

数组原语

[array:from-list](#) [array:item](#) [array:set](#) [array:length](#) [array:to-list](#)

array:from-list

array:from-list *list*

根据输入列表返回一个数组，该数组以列表中相同的顺序包含列表中的各项。

array:item

array:item *array index*

返回给定数组给定索引处的项。（索引号从 0 到数组规模-1）

array:set

array:set *array index value*

设置给定数组给定索引处的项为给定值。（索引号从 0 到数组规模-1）

注意与列表的[replace-item](#) 不同，这里不创建新数组，给定的数组被修改。

array:length

array:length *array*

返回给定数组的长度，即数组的项数。

array:to-list

array:to-list *array*

根据给定的数组，按相同的顺序创建一个包含相同项的列表，返回该列表。

哈希表原语

[table:clear](#) [table:from-list](#) [table:get](#) [table:has-key?](#) [table:keys](#) [table:length](#)
[table:make](#) [table:put](#) [table:remove](#) [table:to-list](#)

table:clear

`table:clear table`

删除 *table* 的所有键-值对。

table:from-list

`table:from-list list`

根据 *list* 的内容创建一个新哈希表。*List* 必须是一个两项列表的列表 (a list of two element lists)，子列表的第一项是键，第二项是值。

table:get

`table:get table key`

返回表中 *key* 映射的值。如果表中没有该键，则引发错误。

table:has-key?

`table:has-key? table key`

如果 *table* 有 *key* 条目返回 true。

table:keys

`table:keys table`

返回 *table* 所有键构成的一个列表。

table:length

`table:length table`

返回 *table* 中的条目数。

table:make

`table:make`

返回一个新建的空哈希表。

table:put

`table:put table key value`

在 *table* 中建立 *key* 和 *value* 的映射。如果给定的键在表中已有条目，则替换。

table:remove

`table:remove table key`

删除 *table* 中 *key* 的映射。

table:to-list

`table:to-list table`

根据 *table* 内容创建列表返回。该列表是由多个两项列表构成的，两项列表的第一项是键，第二项是值。

声音扩展（Sound Extension）¹²

NetLogo 的 sound 扩展提供了一些原语，为模型增加声音。它提供了两种产生声音的方式：MIDI 声音和回放已录制好的声音文件。

使用 Sound 扩展

Sound 扩展已经预安装。要在模型中使用该扩展，在例程页的首部增加一行：

`extensions [sound]`

如果模型已经使用了其他扩展，则已经有一行[extensions](#)，只需把sound加到列表中。

关于使用NetLogo扩展的更多信息，参见扩展指南（[Extensions Guide](#)）。

在 NetLogo 模型库 Code Examples 的 Sound 部分，提供了使用 sound 扩展的例子。

注意 sound 扩展只能在 NetLogo 应用程序里工作，在保存的 applet 里无效。

MIDI 支持

MIDI 部分模拟 128 键电子键盘，有 47 种鼓乐（drums）和 128 种旋律乐器（melodic instruments），与通用 MIDI 一级规范一致。

它支持 15 个复音乐器通道和一个打击乐器通道。如果在模型里同时使用 15 种以上的旋律乐器，会导致某些声音消失或切断。

旋律乐器的音调由一个键号指定。键盘上的键从 0-127 连续编号，0 是最左的键，中音 C 是 60 号键。

乐器的音量由一个速度值指定，代表弹奏键盘的力度。速度值 0-127，64 是标准速度，速度值越高，音量越大。

原语

[sound:drums](#) [sound:instruments](#) [sound:play-drum](#) [sound:play-note](#)
[sound:play-note-later](#) [sound:play-sound](#) [sound:play-sound-and-wait](#)
[sound:play-sound-later](#) [sound:start-note](#) [sound:stop-note](#) [sound:stop-instrument](#)
[sound:stop-music](#)

¹² 译者说明：本部分涉及一些音乐方面的术语，本人的音乐知识非常贫乏，恐怕有不少误译之处，请通晓音乐且英文水平高的读者批评指正。

sound:drums

sound:drums

返回包括 47 种鼓乐名称的列表，用于 “sound:play-drum”。

sound:instruments

sound:instruments

返回包括 128 种乐器名称的列表，用于“sound:play-note”，“sound:play-note-later”，“sound:start-note” 和“sound:stop-note”。

sound:play-drum

sound:play-drum *drum velocity*

演奏鼓乐

`sound:play-drum "ACOUSTIC SNARE" 64`

sound:play-note

sound:play-note *instrument keynumber velocity duration*

演奏某音符给定时长（秒）。主体不会等待音符演奏完再执行下一条命令。

`; play a trumpet at middle C for two seconds`

`sound:play-note "TRUMPET" 60 64 2`

sound:play-note-later

sound:play-note-later *delay instrument keynumber velocity duration*

等待给定的延迟，然后演奏音符给定时长（秒）。主体不会等待音符演奏完再执行下一条命令。

`; in one second, play a trumpet at middle C for two seconds`

`sound:play-note-later 1 "TRUMPET" 60 64 2`

sound:play-sound

sound:play-sound *filename*

播放声音文件。不会等待文件演奏完再执行下一条命令。支持 WAV, AIFF, AU 文件。

```
;; plays the beep.wav sample file
sound:play-sound "beep.wav"
```

sound:play-sound-and-wait

sound:play-sound-and-wait *filename*

播放声音文件。等待文件演奏完再执行下一条命令，支持 WAV, AIFF, AU 文件。

```
;; plays the beep.wav sample file, waiting for it to finish before
;; playing boop.wav
sound:play-sound-and-wait "beep.wav"
sound:play-sound-and-wait "boop.wav"
```

sound:play-sound-later

sound:play-sound-later *filename delay*

等待给定秒数的延迟后，播放声音文件。不会等待文件演奏完再执行下一条命令，支持 WAV, AIFF, AU 文件。

```
;; plays the beep.wav sample file one second from now
sound:play-sound-later "beep.wav" 1
```

sound:start-note

sound:start-note *instrument keynumber velocity*

开始演奏一个音符。

演奏持续进行，直到遇到“sound:stop-note”，“sound:stop-instrument”或“sound:stop-music”。

```
;; play a violin at middle C
sound:start-note "VIOLIN" 60 64
;; play a C-major scale on a xylophone
foreach [60 62 64 65 67 69 71 72] [
  sound:start-note "XYLOPHONE" ? 65
  wait 0.2
```

```
sound:stop-note "XYLOPHONE" ?  
]
```

sound:stop-note

sound:stop-note *instrument keynumber*

停止演奏一个音符。

```
;; stop a violin note at middle C  
sound:stop-note "VIOLIN" 60
```

sound:stop-instrument

sound:stop-instrument *instrument*

停止一个乐器的所有音符

```
;; stop all cello notes  
sound:stop-instrument "CELLO"
```

sound:stop-music

sound:stop-music

停止所有音符。

声音名称

Drums

- | | |
|------------------------|-------------------|
| 35. Acoustic Bass Drum | 59. Ride Cymbal 2 |
| 36. Bass Drum 1 | 60. Hi Bongo |
| 37. Side Stick | 61. Low Bongo |
| 38. Acoustic Snare | 62. Mute Hi Conga |
| 39. Hand Clap | 63. Open Hi Conga |
| 40. Electric Snare | 64. Low Conga |
| 41. Low Floor Tom | 65. Hi Timbale |
| 42. Closed Hi Hat | 66. Low Timbale |
| 43. Hi Floor Tom | 67. Hi Agogo |
| 44. Pedal Hi Hat | 68. Low Agogo |
| 45. Low Tom | 69. Cabasa |
| 47. Open Hi Hat | 70. Maracas |

- | | |
|--------------------|--------------------|
| 47. Low Mid Tom | 71. Short Whistle |
| 48. Hi Mid Tom | 72. Long Whistle |
| 49. Crash Cymbal 1 | 73. Short Guiro |
| 50. Hi Tom | 74. Long Guiro |
| 51. Ride Cymbal 1 | 75. Claves |
| 52. Chinese Cymbal | 76. Hi Wood Block |
| 53. Ride Bell | 77. Low Wood Block |
| 54. Tambourine | 78. Mute Cuica |
| 55. Splash Cymbal | 79. Open Cuica |
| 56. Cowbell | 80. Mute Triangle |
| 57. Crash Cymbal 2 | 81. Open Triangle |
| 58. Vibraslap | |

Instruments

Piano

1. Acoustic Grand Piano
2. Bright Acoustic Piano
3. Electric Grand Piano
4. Honky-tonk Piano
5. Electric Piano 1
6. Electric Piano 2
7. Harpsichord
8. Clavi

Reed

65. Soprano Sax
66. Alto Sax
67. Tenor Sax
68. Baritone Sax
69. Oboe
70. English Horn
71. Bassoon
72. Clarinet

Chromatic Percussion

9. Celesta
10. Glockenspiel
11. Music Box
12. Vibraphone
13. Marimba
14. Xylophone
15. Tubular Bells
16. Dulcimer

Pipe

73. Piccolo
74. Flute
75. Recorder
76. Pan Flute
77. Blown Bottle
78. Shakuhachi
79. Whistle
80. Ocarina

Organ

17. Drawbar Organ
18. Percussive Organ
19. Rock Organ
20. Church Organ
21. Reed Organ
22. Accordion
23. Harmonica
24. Tango Accordion

Synth Lead

81. Square Wave
82. Sawtooth Wave
83. Calliope
84. Chiff
85. Charang
86. Voice
87. Fifths
88. Bass and Lead

Guitar	Synth Pad
25. Nylon String Guitar	89. New Age
26. Steel Acoustic Guitar	90. Warm
27. Jazz Electric Guitar	91. Polysynth
28. Clean Electric Guitar	92. Choir
29. Muted Electric Guitar	93. Bowed
30. Overdriven Guitar	94. Metal
31. Distortion Guitar	95. Halo
32. Guitar harmonics	96. Sweep
Bass	Synth Effects
33. Acoustic Bass	97. Rain
34. Fingered Electric Bass	98. Soundtrack
35. Picked Electric Bass	99. Crystal
36. Fretless Bass	100. Atmosphere
37. Slap Bass 1	101. Brightness
38. Slap Bass 2	102. Goblins
39. Synth Bass 1	103. Echoes
40. Synth Bass 2	104. Sci-fi
Strings	Ethnic
41. Violin	105. Sitar
42. Viola	106. Banjo
43. Cello	107. Shamisen
44. Contrabass	108. Koto
45. Tremolo Strings	109. Kalimba
47. Pizzicato Strings	110. Bag pipe
47. Orchestral Harp	111. Fiddle
48. Timpani	112. Shanai
Ensemble	Percussive
49. String Ensemble 1	113. Tinkle Bell
50. String Ensemble 2	114. Agogo
51. Synth Strings 1	115. Steel Drums
52. Synth Strings 2	116. Woodblock
53. Choir Aahs	117. Taiko Drum
54. Voice Oohs	118. Melodic Tom
55. Synth Voice	119. Synth Drum
56. Orchestra Hit	120. Reverse Cymbal
Brass	Sound Effects
57. Trumpet	121. Guitar Fret Noise
58. Trombone	122. Breath Noise

- | | |
|-------------------|---------------------|
| 59. Tuba | 123. Seashore |
| 60. Muted Trumpet | 124. Bird Tweet |
| 61. French Horn | 125. Telephone Ring |
| 62. Brass Section | 126. Helicopter |
| 63. Synth Brass 1 | 127. Applause |
| 64. Synth Brass 2 | 128. Gunshot |

GoGo 扩展 (GoGo Extension)

GoGo 板 (GoGo Board) 是什么?

GoGo 板扩展用来将 NetLogo 与物理世界连接起来，使用传感器、电机、灯泡、LED、继电器和其他设备。NetLogo 的 GoGo 扩展提供了与 GoGo 板通过串口进行通信的原语。

GoGo板是一个开源、易做、便宜、通用的板卡，用于教育项目。它是MIT媒体实验室的[Arnan Sipitakiat](#)创造的。GoGo板有 8 个传感器端口和 4 个输出端口，还有一个用于连接附加卡(如显示器、无限通信模块等)的连接器。使用GoGo板扩展，NetLogo可以用两种方式与物理世界交互，第一，可以收集环境数据，如温度、环境光或用户输入，这些信息可为模型使用，用于改变或校准它的行为。第二，它可以控制输出设备 - NetLogo可以控制电机、玩具、遥控车、电器、灯泡和自动化实验设备。

怎样得到 GoGo 板?

GoGo板不是商品，在商店里买不到。你必须自己做或请人帮着做一个。该板的设计追求便宜和易做，即使你没有电子方面的技能也能做。关于GoGo板的主要资源在网站www.gogoboard.org，教你一步步怎么做，包括购买元件、设计印刷电路板、组装。GoGo板的邮件列表是gogoboard@yahoogroups.com。

安装 GoGo 扩展

GoGo 板使用串口以某种方式与计算机通信。之所以选择串口而不是 USB 口是出于成本考虑，因为用于构建 USB 兼容板卡的元件较贵。如果计算机没有串口，需要买一个 USB-串口转换器，一般计算机商店都有，价格约 US\$15–30。（如果是 Mac 或 Linux，确认转换器是否与你的操作系统兼容）。

Mac OS X

为了使用串口，登录用户必须对目录/var/spool/uucp有写权限。可以运行脚本 gogo/lib/Mac OS X/fix-permissions.command 设置适当的权限，它使得uucp用户组对这个目录有写权限，并将当前帐户加入该组。在该文件上双击运行。

Linux

需要有写入串行设备的权限，该设备正常情况下是/dev/ttyS*。多数Linux发行包

可以用User Manager来设置。

使用 GoGo 扩展

GoGo 扩展预安装了。要使用它，在例程页首部增加一行：

```
extension [gogo]
```

如果模型已经使用了其他扩展，则已有[extensions](#)行，只需将gogo加到列表中。

扩展加载后，在命令中心输入以下命令，查看哪些端口可用：

```
print gogo:ports
```

使用gogo:open命令打开GoGo所连的端口，使用ping报告器查看该端口是否有反应。

在 Windows：

```
gogo:open "COM1"
```

```
print gogo:ping
```

在 Linux：

```
gogo:open "/dev/ttyS01"
```

```
print gogo:ping
```

关于NetLogo扩展的更多信息，参见扩展指南 ([Extensions Guide](#))

作为 applet 保存的模型（使用 NetLogo 的 File 菜单“Save as Applet”）不能使用 GoGo 扩展，因为 applet 不能使用需要额外 jar 的扩展。（另外未授权的 applet 不能访问外部设备）。在 NetLogo 的 Code Examples 的 GoGo 部分，有一些使用 GoGo 扩展的例子。

原语

```
gogo:close gogo:open gogo:open? gogo:ports gogo:output-port-coast  

gogo:output-port-off gogo:output-port-reverse gogo:output-port-[that|this]way  

gogo:ping gogo:sensor gogo:set-output-port-power gogo:talk-to-output-ports
```

gogo:close

`gogo:close`

关闭与 GoGo 板的连接

另见 [gogo:open](#) 和 [gogo:open?](#) 。

gogo:open

`gogo:open port-name`

通过端口名*port-name*打开与GoGo板的连接。关于端口名，参见[gogo:ports](#)。

如果 GoGo 板没有反应，或者企图打开没有 GoGo 板连接的端口，则产生错误。

例子：

```
gogo:open "COM1"
```

另见 [gogo:open?](#) 和 [gogo:close.](#)

gogo:open?

`gogo:open?`

如果与 GoGo 板的一个连接打开了，返回 true，否则返回 false。

gogo:ports

`gogo:ports`

返回 GoGo 可以连接的端口名列表。在一些计算机上列表有 2-3 个不同的串口。如果这样的话，一一实验直到连接成功。

gogo:output-port-coast

`gogo:output-port-coast`

关闭活动端口的电源。当连接电机时，不像[gogo:output-port-off](#)那样使用制动力。因此电机逐渐减速最终停下来。除电机外，该命令与[gogo:output-port-off](#)有相同的效果。该命令所影响的端口由[gogo:talk-to-output-ports](#)决定。

下面的代码将打开 a 输出端口 1 秒，然后逐渐停下电机：

```
gogo:talk-to-output-ports ["a"]
gogo:output-port-on
wait 1
gogo:output-port-coast
```

gogo:output-port-off

gogo:output-port-off

关闭输出端口的电源。如果使用电机，则使用制动力。该命令所影响的端口由[gogo:talk-to-output-ports](#)决定。

gogo:output-port-reverse

gogo:output-port-reverse

反转输出端口的方向。该命令所影响的端口由[gogo:talk-to-output-ports](#)决定。

gogo:output-port-[that/this]way

gogo:output-port-thatway

gogo:output-port-thisway

对端口以给定的方向提供动力。输出端口可以有两种动力方向，任意命名为*thisway* 和 *thatway*。该命令所影响的端口由[gogo:talk-to-output-ports](#)决定。注意这与[gogo:output-port-reverse](#) 不同，因为如果连接器的极性相同的话，*thisway* 和 *thatway* 总是同一方向。

gogo:talk-to-output-ports

gogo:talk-to-output-ports *output-portlist*

该命令将对应的输出端口设为活动。这是如[gogo:output-port-on](#) 和 [gogo:output-port-off](#) 命令影响的端口。用户可以同时与一个或多个端口交谈（“talk”）。输出端口一般连到电机，但也可连到灯泡、LED和继电器。输出端口通过单字符名字来区分：“a”，“b”，“c”，和“d”。

例子：

```
; talk to all output-ports  
gogo:talk-to-output-ports [ "a" "b" "c" "d" ]  
;; will give power to all output-ports  
gogo:output-port-on  
  
;; talk to output-ports A and D  
gogo:talk-to-output-ports [ "a" "d" ]
```

```

;; will turn off output-ports A and D.
;; The other output-ports will keep
;; their current state
gogo:output-port-off

gogo:talk-to-output-ports [ "c" "b" ]
;; turn off remaining output-ports
gogo:output-port-off

```

gogo:ping

gogo:ping

检查 GoGo 板的状态。常用于确认 GoGo 板连接到正确的串口。如果 GoGo 板对诊断信息有反应则返回 true，否则返回 false。

例子：

```
print gogo:ping
```

gogo:sensor

gogo:sensor *sensor*

将名为 *sensor* 的传感器的值作为数值返回。传感器由 1-8 的数字命名。数值范围是 0-1023。当没有传感器与端口相连（最高阻抗）或传感器是“open”状态，返回 1023。当传感器短路（阻抗为 0）时返回 0。

例子：

```

print gogo:sensor 1
;; prints the value of sensor 1

foreach [ 1 2 3 4 5 6 7 8 ]
  [print (word "Sensor " ? " = " gogo:sensor ?)]
;; prints the value of all sensors

if gogo:sensor 1 < 500 [ ask turtles [ fd 10 ] ]
;; will move all turtles 10 steps forward if sensor 1's value is less than 500.

loop [if gogo:sensor 1 < 500 [ ask turtles [ fd 10 ] ] ]
;; will continuously check sensor 1's value and
;; move all turtles 10 steps forward every time

```

```
;; that the sensor value is less than 500.
```

gogo:set-output-port-power

gogo:set-output-port-power *power-level*

设置活动输出端口的动力水平。*power-level*是0-7的一个数值，0为关，7为满动力。该命令所影响的端口由[gogo:talk-to-output-ports](#)决定。注意在实际应用时，更有效的方式是使用机械装置，如齿轮、滑轮等，控制电机的扭矩。

例子：

```
gogo:talk-to-motors ["a" "b" "c" "d"]
gogo:set-motor-power 4
;; will lower the power of all output ports by half of the full power .
```

性能剖析扩展 (Profiler Extension)

Profiler 扩展提供信息，帮你使模型运行更快。它包括一组原语，用来计量模型中每个例程调用多少次，每次调用花了多长时间。

告诫

注意兼容性：该扩展需要运行 NetLogo 时使用 Java 1.5 以上，它用到了 Java 1.4 没有的功能。在 Windows 没有问题，因为下载的 NetLogo 包含了 Java 1.5。Mac 用户必须使用 Mac OS X 10.4 以上，因为在 10.2 或 10.3 上 Apple 没有提供 Java 1.5。

小心！ Profiler 扩展是新的、实验性的。没有经过全面测试，用户友好性不够。虽然如此，某些用户可能觉得它有用。

使用

profiler 扩展预安装了。要使用它，在例程页首部增加一行：

`extensions [profiler]`

如果模型已经使用了其他扩展，则已有 [extensions](#) 行，只需将 profiler 加到列表中。

关于 NetLogo 扩展的更多信息，参见扩展指南 ([Extensions Guide](#))

例子

```
setup          ;; set up the model
profiler:start    ;; start profiling
repeat 20 [ go ]      ;; run something you want to measure
profiler:stop      ;; stop profiling
print profiler:report  ;; view the results
profiler:reset      ;; clear the data
```

代码例子：Profiler Example

原语

[profiler:calls](#) [profiler:exclusive-time](#) [profiler:inclusive-time](#) [profiler:start](#)
[profiler:stop](#) [profiler:reset](#) [profiler:report](#)

profiler:calls

`profiler:calls procedure-name`

返回例程 *procedure-name* 被调用的次数。如果没有定义 *procedure-name* 例程，返回 0。

profiler:exclusive-time

`profiler:exclusive-time procedure-name`

返回 *procedure-name* 的独占运行时间，单位毫秒。独占时间从进入例程开始，直到它结束为止，但不包括它所调用的用户定义例程的时间。

如果没有定义 *procedure-name* 例程，返回 0。

profiler:inclusive-time

`profiler:inclusive-time procedure-name`

返回 *procedure-name* 的非独占运行时间，单位毫秒。非独占时间从进入例程开始，直到它结束为止。

如果没有定义 *procedure-name* 例程，返回 0。

profiler:start

`profiler:start`

命令 profiler 开始记录用户例程调用。

profiler:stop

`profiler:stop`

命令 profiler 停止记录用户例程调用。

profiler:reset

`profiler:reset`

命令 profiler 删除所有已收集的数据。

profiler:report

profiler:report

返回一个字符串，包含所有用户定义例程的调用细目。Calls列是用户定义例程调用的次数，Incl T(ms)列是独占时间，Excl T(ms)是非独占时间，Excl/calls列是每次调用用户例程所花时间的估计值。

下面是一个输出例子：

Sorted by Exclusive Time

Name	Calls	Incl T(ms)	Excl T(ms)	Excl/calls
CALLTHEM	13	26.066	19.476	1.498
CALLME	13	6.413	6.413	0.493
REPORTME	13	0.177	0.177	0.014

Sorted by Inclusive Time

Name	Calls	Incl T(ms)	Excl T(ms)	Excl/calls
CALLTHEM	13	26.066	19.476	1.498
CALLME	13	6.413	6.413	0.493
REPORTME	13	0.177	0.177	0.014

Sorted by Number of Calls

Name	Calls	Incl T(ms)	Excl T(ms)	Excl/calls
CALLTHEM	13	26.066	19.476	1.498

常见问题解答(Frequently Asked Questions)

用户的反馈对我们设计和改进NetLogo非常有价值，很高兴听取你的反馈。请将意见、建议和问题发送到 feedback@ccl.northwestern.edu，bug报告发送到 bugs@ccl.northwestern.edu。

问题

一般问题

- 它为什么叫 NetLogo?
- 在学术出版物上如何引用 NetLogo?
- 在出版物上如何引用模型库中的模型?
- NetLogo 是在何时、何地开发的?
- NetLogo 是用哪种编程语言写的?
- StarLogo, MacStarLogo, StarLogoT, NetLogo 有何区别?
- NetLogo 采用哪种许可? 使用、分发等是否受到任何法律限制?
- 能得到 NetLogo 源代码吗?
- 你们提供 NetLogo 研讨会或训练机会吗?
- 有 NetLogo 教科书吗?
- NetLogo 有西班牙版,德语版,(你的语言) 版吗?
- NetLogo 是编译的还是解释的?
- 有人建过某某模型吗?
- NetLogo 模型具有科学意义上的可重现性吗?
- NetLogo 和 NetLogo 3D 还会保持分离吗?
- 还支持 NetLogo 旧版本吗?

下载问题

- 下载表单不能正常使用，有到软件的直接连接吗?
- 下载 NetLogo 太费时间，有其他的方式吗，比如 CD?
- 我下载和安装了 NetLogo ,但模型库的模型很少或没有。这个问题怎么解决?
- 我能同时安装多个版本的 NetLogo 吗?
- 我使用 UNIX ，不能解包下载文件，为什么?
- 怎样以无人看管方式安装 NetLogo?
- 我下载了无 Java 的 Windows 安装文件,为什么 NetLogo 不能启动?

Java 小程序（Applet）问题

- 我想运行网站上的一个 applet，但它不运行。我该怎么办？
- 我能将模型作为 applet，但不公开源代码吗？
- 保存为applet 的模型能通过import-world, file-open, 和其他命令读文件吗？
- 当以 applet 方式运行我的模型时，出现错误: java.lang.OutOfMemoryError: Java heap space。
- 当我以 Applet 加载我的模型时，出错: java.lang.ClassFormatError: Incompatible magic value。

运行问题

- 能在 CD 上运行 NetLogo 吗？
- 当拔掉 Windows 笔记本的电源时，为什么 NetLogo 这么慢？
- 在 Linux 机器上 NetLogo 怎么不能运行？
- 在 Windows 启动 NetLogo 时，出现错误信息"could not create Java virtual machine"，帮忙啊！
- 能使用命令行而不需 GUI 运行 NetLogo 吗？
- NetLogo 能利用多个处理器/内核的优势吗？
- NetLogo 模型能分布运行在计算机集群上吗？
- 我想试试 HubNet，可以吗？

使用问题

- 当我把速度滑动条一下子拖到最右边时，为什么模型看起来像停止一样？
- 怎么改变瓦片的数量？
- 我能使用鼠标在视图上画图 ("paint") 吗？
- 模型可以多大？模型能包含多少海龟、瓦片、例程、按钮等？
- 可以将 GIS 数据导入 NetLogo 吗？
- 模型运行很慢，怎么能快一些？
- 能同时打开多个模型吗？
- 能在运行时改变选择器的选项吗？
- 能将模型代码分到几个文件吗？

编程问题

- NetLogo 语言和 StarLogo 和 StarLogoT 语言有什么区别？怎么把 StarLogo 或 StarLogoT 模型转换为 NetLogo 模型？
- NetLogo 语言与其他 Logo 语言有什么区别？
- 用以前 NetLogo 建立的模型怎么不能正常工作？
- 为什么代码里有奇怪的字符？

- 怎样取一个数的负值?
- 海龟前进 1, 但还在原来的瓦片上, 为什么?
- 怎样保持海龟在瓦片中心?
- patch-ahead 1 返回海龟当前所在的瓦片, 为什么?
- 怎样给海龟视场("vision")?
- 海龟能感知画图层 (drawing layer) 的东西吗?
- 我得到的是像 0.10000000004 和 0.799999999999 这样的数, 而不是 0.1 和 0.8。为什么?
- 文档说random-float 1 可能返回 0 但不可能返回 1, 如果我想包括 1 怎么办?
- 怎样防止两个海龟占据同一个瓦片?
- 怎样判断海龟是否死亡?
- NetLogo 有数组吗?
- NetLogo 有哈希表或关联数组吗?
- 怎样使用不同的瓦片邻域 ("neighborhoods") (圆形, Von Neumann, Moore, 等)?
- 怎样将一个主体集合转换为主体的列表, 或相反?
- 如何停止 foreach?

行为空间问题

- 怎样每n个滴答收集一次运行数据?
- 我变动在例程页声明的一个全局变量, 但没作用。为什么?
- 为什么在 Excel 里某些结果被截掉了?

扩展问题

- 我编写了一个扩展, 为什么编译器说找不到 org.nlogo.api?

一般问题

它为什么叫 NetLogo?

"Logo"是因为 NetLogo 是 Logo 语言的一个方言。

"Net"暗指用 NetLogo 建模的现象（包括网络现象）具有分散化、互联性。也指 HubNet，这个 NetLogo 所带的多用户参与式仿真环境。

在学术出版物上如何引用 NetLogo?

引用NetLogo: Wilensky, U. 1999. NetLogo. <http://ccl.northwestern.edu/netlogo/>. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.

引用 HubNet: Wilensky, U. & Stroup, W., 1999. HubNet. <http://ccl.northwestern.edu/netlogo/hubnet.html>. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.

在出版物上如何引用模型库中的模型?

在每个模型的信息页的 CREDITS AND REFERENCES 部分告诉了正确的引用方式。

NetLogo 是在何时、何地开发的?

NetLogo 最早是 1999 年由 Uri Wilensky 在 Center for Connected Learning and Computer-Based Modeling 创建的，那时是在 Boston 的 Tufts University。NetLogo 出自 StarLogoT，StarLogoT 是 Wilensky 在 1997 年开发的。2000 年，CCL 迁入 Chicago 地区的 Northwestern University。NetLogo 1.0 在 2002 年发行，2.0 是 2003 年，3.0 是 2005 年，4.0 是 2007 年。

NetLogo 是用哪种编程语言写的?

NetLogo 完全用 Java 语言(1.4.1 版)写的。

StarLogo, MacStarLogo, StarLogoT, NetLogo 有何区别?

最早的StarLogo是MIT Media Lab在 1989–1990 年开发的，运行在名为Connection Machine的大型并行超级计算机上。几年后（1994 年），为Macintosh计算机开发了伪并行

版，该版本最终演变成MacStarLogo 。[StarLogoT](#) (1997)是在连接学习与计算机建模中心开发(CCL)的，基本是对MacStarLogo的扩展，还增加了许多功能和特性。

自那以后开发了两个跨平台的、基于 Java 的多主体 Logo: NetLogo(CCL 开发) 和 StarLogo 的 Java 版 (MIT 开发) 。

NetLogo 和 MIT 的 StarLogo 在语言和环境方面有许多不同。二者都受原始 StarLogo 的启发，但许多方面有所区别。NetLogo 的设计目的是对语言进行修改和扩展，使之容易使用和更加强大，并支持 HubNet 结构。NetLogo 包含了早期 StarLogoT 的所有扩展功能，还有很多新特征。

NetLogo 采用哪种许可 License? 使用、分发等是否受到任何法律限制?

在 NetLogo 用户手册的“Copyright”部分给出了 License，在应用程序的 about 对话框，以及下载包的 README 文件也带了 License 文本。

粗略而言，使用(包括商业使用)不受任何限制，但重新分发和/或修改受一些限制(除非你与 Uri Wilensky 联系，协商不同的条款)

我们还在根据用户的反馈重新评估 License 文本，将来可能做出修改。

能得到 NetLogo 源代码吗?

目前不行。我们还在工作，最后会遵照开源 license 公开源代码。

不过 NetLogo 不是封闭平台，我们提供了 API 从外部 Java 程序控制 NetLogo，以及使用 Java 编写新的命令和报告器。(见用户手册的“控制”和“扩展”部分)。我们鼓励用户编写扩展，在 NetLogo 用户社区共享。

你们提供 NetLogo 研讨会或训练机会吗?

我们有时提供研讨会。如果安排了研讨会，我们会在NetLogo主页和用户组通知。如果对研讨会感兴趣，与我们联系feedback@ccl.northwestern.edu。

有 NetLogo 教科书吗?

在 CCL 我们曾希望写一些 NetLogo 的教程，面对不同层面的读者，如初中、高中、大学的建模或复杂性课程、为成人写的实用指南等。

非常遗憾我们一直没时间做这些。如果用户社区的某人对这项感兴趣，告诉我们，我们很欢迎。

NetLogo 有西班牙版,德语版,(你的语言) 版吗?

目前 NetLogo 仅有英语版。

我们计划最终使用户能够为 NetLogo 制造自己的语言包。要实现这一点, 我们需要将源代码中的文本分离出来, 还不知道什么时候能做到。

NetLogo 是编译的还是解释的?

简短回答: 部分编译, 我们正在工作使它成为完全编译的。

详细回答: NetLogo 包括一个产生 Java 字节码的编译器。然而现在还不支持整个语言, 因此部分代码是解释的。我们正在扩展编译器使它支持整个语言。注意我们的编译器产生字节码, Java 虚拟机有一个“即时”编译器接着将字节码编译成机器码, 因此用户程序最终转换为机器码。

有人建过某某模型吗?

最好到[NetLogo Users Group](#) 问这个问题。

也可以查看模型库网页的 Community Models 部分。

NetLogo 模型具有科学意义上的可重现性吗?

有。 NetLogo 的主体调度算法是确定性的, 并且 NetLogo 总是使用 Java 的“strict math”库, 不管平台如何, 总是给出按位 (bit-for-bit) 相同的结果。但是要记住以下几点:

- 如果模型使用随机数, 为了获得可重现行为, 必须先使用 random-seed 命令设置随机数种子, 这样模型每次都使用相同的随机数序列。记住主体集合总是采用随机顺序, 因此使用主体集合时使用了随机数。
- 如果以影响输出的形式使用 every 或 wait 命令, 在不同的计算机, 甚至在同一台计算机上可能得到不同的结果, 因为模型运行速度可能不同。(这样的模型很少, 这两个命令很常见, 但影响结果的使用方式不常见)
- 要准确重现结果, 必须使用同一个 NetLogo 版本。在不同的 NetLogo 版本, 主体调度机制和随机数发生器可能不同, 其他的改变 (引擎修复、语言改变等) 也可能影响模型的行为 (当然也可能不影响)。
- 我们付出很多努力保证 NetLogo 模型的可重现, 但这也不是铁的保证, 这不仅因为可能会硬件故障, 也可能因为设计中的人为差错: 你的模型, NetLogo, Java 虚拟机, 硬件等等。

NetLogo 和 NetLogo 3D 还会保持分离吗?

不会。分离是暂时的。最终是一个支持 2D 和 3D 的统一版 NetLogo。我们保证 3D 世界

的支持不会妨碍你建造 2D 模型。

使用 NetLogo 3D preview 版建造的模型要在最终统一版上运行，可能需要修改。

还支持 NetLogo 旧版本吗？

支持。我们还支持 NetLogo 1.3.1 (Mac OS 8 和 9, Windows 95), NetLogo 2.0.2, NetLogo 2.1, NetLogo 3.0.2, NetLogo 3.1.5。只要人们还使用，我们就继续支持。

如果用户报告 bug 或有兼容问题需要修复，我们还会发行 3.1.x 系列。

为避免用户头大，NetLogo 下载页只列出部分版本（上面所列）。但是如果你需要某个特定版本没有列出，与我们联系，我们很乐意提供。

下载问题

下载表单不能正常使用，有到软件的直接连接吗？

请发邮件到bugs@ccl.northwestern.edu，我们或者修改下载表单，或给你提供一种其他下载软件的方式。

下载 NetLogo 太费时间，有其他的方式吗，比如 CD？

目前没有。如果你有这样的问题，与我们联系feedback@ccl.northwestern.edu.

我下载和安装了 NetLogo，但模型库的模型很少或没有。这个问题怎么解决？

至今反映这个问题的用户使用的都是 Windows 平台下的“without VM”的下载选项。卸载 NetLogo，尝试“with VM”下载选项。

即使使用“with VM”下载解决了你的问题，请还是与我们联系bugs@ccl.northwestern.edu，这样我们能了解 setup 的更多细节，在以后的版本中解决这个问题，要解决的话需要用户的帮助。

我能同时安装多个版本的 NetLogo 吗？

可以。当安装 NetLogo 时，所创建的文件夹包含版本号，因此多个版本可以共存。

在 Windows，你最后安装的版本将在 Windows 浏览器中双击模型文件时打开。在 Mac，你可以在 Finder 的“Get Info”控制使用哪个版本。

我使用 UNIX , 不能解包下载文件, 为什么?

下载tar包的某些文件路径名太长, 超过标准tar格式所允许。必须使用GNU版的tar(或其他理解GNU tar扩展的程序)。在某些系统, GNU版的 tar程序名是"gnutar"。输入tar - version 看看输出是否是"tar (GNU tar)", 这样可以了解所用的tar 是否是GNU 版。

怎样以无人看管方式安装 NetLogo?

取决于所用操作系统.

- **Linux:** 将 NetLogo 解包到合适的地方。

Mac: 从磁盘镜像复制 NetLogo 目录到应用程序文件夹。

- **Windows:**

安装 NetLogo, 然后复制安装目录到其他机器。不幸的是, NetLogo 不会出现在开始菜单, 双击.nlogo 文件也不会自动启动 NetLogo。然而 Anders Martinusen 写到:

发现可以从 Windows 的注册表将几个键输出到.reg 文件, 这个文件可以与其他文件一起压入目标机器。这样能解决双击.nlogo 文件的问题, 也可以使以后从控制面板卸载程序成为可能。

Reg 文件内容:

```
[HKEY_CLASSES_ROOT\.nlogo]
@="NetLogoModelFile"

[HKEY_CLASSES_ROOT\NetLogoModelFile]
@="NetLogo model file"

[HKEY_CLASSES_ROOT\NetLogoModelFile\DefaultIcon]
@="%TARGET_PATH%\Model icon.ico,0"

[HKEY_CLASSES_ROOT\NetLogoModelFile\shell\open\command]
@="\%TARGET_PATH%\NetLogo 4.0.2.exe\" --launch \"%1\""

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\
NetLogo
4.0.2]
"DisplayName"="NetLogo 4.0.2"
"UninstallString"="\%TARGET_PATH%\UninstallerData\Uninstall
NetLogo.exe\""
```

我下载了无 Java 的 Windows 安装文件,为什么 NetLogo 不能启动?

推荐带有 Java 的 Windows 下载。如果使用无 Java 的下载遇到问题,参考需求部分的细节。

Java 小程序 (Applet) 问题

我想运行网站上的一个 applet, 但它不运行。我该怎么办?

当前版本的 NetLogo 要求网页浏览器支持 Java 1.4.1 以上。关于 java 需求的细节, 参考 [Java 小程序指南](#)。

某些 NetLogo applet 需要的内存超过浏览器正常设置。关于如何改变内存分配, 参考 [Java 小程序指南](#)的[内存](#)部分。

我能将模型作为 applet, 但不公开源代码吗?

不行。Applet 要正常运行, 模型文件必须可访问。

在 File 菜单使用“Save as applet”, 所产生的 HTML 页包含下载模型文件的链接。如果你想的话, 可以移去此处的链接。如果这样做, 用户不容易访问模型文件, 但并非不可能。

保存为applet 的模型能通过import-world, file-open, 和其他命令读文件吗?

可以, 但只能读取 Web 服务器上与模型文件和 HTML 同一目录下的文件。Applet 不能读取用户机器上的文件, 只能读取 Web 服务器上的文件。

当以 applet 方式运行我的模型时, 出现错误: **java.lang.OutOfMemoryError: Java heap space.**

Java插件没有分配足够的空间来运行模型。NetLogo的模型可以有多大, 参见[here](#)。需要增加Java插件的可用内存, 方法见[here](#)。

当我以 Applet 加载我的模型时，出错： **java.lang.ClassFormatError: Incompatible magic value.**

如果你的浏览器对请求不存在的页返回定制错误信息，它必须同时返回错误码 404 Not Found。否则 NetLogo 认为下面的数据正是所请求的，并且试着读取。即使运行 applet 所需的文件都在，也会发生这样的问题。

如果 web 服务器没有控制错误信息，可以使用下面的作为工作区 (you can use the following as a workaround)：

- 在applet文件所在目录创建目录META-INF
- 创建META-INF 的子目录services
- 在services子目录创建文件org.apache.commons.logging.LogFactory
- 将下面的行加到 org.apache.commons.logging.LogFactory:

```
org.apache.commons.logging.impl.LogFactoryImpl
```

注意所有目录和文件名都是大小写敏感，必须完全一致。

运行问题

能在 CD 上运行 NetLogo 吗？

能。NetLogo 在只读文件系统上运行的很好。

当拔掉 Windows 笔记本的电源时，为什么 NetLogo 这么慢？

当拔掉电源时，笔记本切换到节电模式。速度略降一点是正常的，但不幸的是 Java 有一个 bug 使得 Swing 应用程序大幅度变慢，而 NetLogo 也是这样的应用程序。

一种方法是改变计算机的电源设置，使得当拔掉电源时不切换到节电模式。（如果这样的话，电池不能支持太久）。

另一种方法是编辑 NetLogo 目录（在硬盘的 Program Files 目录，除非你安装到其他目录）的 NetLogo.lax 文件，使用 Sun 推荐的一个选项。编辑这一行：

```
lax.nl.java.option.additional=-Djava.ext.dirs=-server -Dsun.java2d.nodraw=true  
并在最后一行末增加 -Dsun.java2d.ddoffscreen=false
```

在这 ([here](#)) 查看Java bug的细节，投票促使Sun排除它。

在 Linux 机器上 NetLogo 怎么不能运行?

理论上任何 Java runtime 1.4.1 以上都能运行 NetLogo。然而，有些 Java 实现不支持 NetLogo 使用的某些功能，如 Java2D 和 Swing。例如 Linux 上基于 GNU libgcj 的 Java 运行库就不行，该运行库在某些 Linux 发行版上预安装。

当在 Linux 上使用 NetLogo 时，推荐使用 Sun 或 IBM 的 Java 运行库。

在 Windows 启动 NetLogo 时，出现错误信息 "could not create Java virtual machine"，帮忙啊！

我们不太清楚这条消息是否有唯一的原因，但我们知道一个可能的原因是 Windows XP 升级包 SP2(或其他 Windows 版本如 Windows Server 2003 等)有一个 bug，在某些机器上不允许分配大块的连续虚拟内存。

一个可能的办法是使用文本编辑器修改 NetLogo 4.0.2.lax 文件（在 NetLogo 目录，默认是在 C:\Program Files）

```
# LAX.NL.JAVA.OPTION.JAVA.HEAP.SIZE.MAX  
# -----  
# allow the heap to get huge  
  
lax.nl.java.option.java.heap.size.max=1073741824
```

试试将 1073741824 改为较小的数如 500000000。这样应该能让 NetLogo 运行起来，但因为堆的大小受限，有很多主体的模型可能无法运行。（参见模型可以有多大的 [How big can my model be?](#)）

另一个可能的方法，但我们不太确认是否有效，是升级到 Windows Vista 或 Windows XP Service Pack 3，当然前提是已经有了。（2007 年 11 月还没有）

能使用命令行而不需 GUI 运行 NetLogo 吗？

能。最容易的方式是将你的模型作为行为空间实验来运行。不需额外编程。参考用户手册的行为空间部分，获知细节。

另一种方法是使用控制 API，这需要编写一些 Java 程序。参考用户手册的控制部分，获知细节和代码例子。

NetLogo 能利用多个处理器/内核的优势吗？

对单个模型不能。NetLogo 引擎是单线程的，我们希望继续保持这样。我们没有任何计划将单个模型在多个处理器或多个计算机上并行运行。

利用多处理器或多核处理器优势的一种方法是同时运行 NetLogo 的多个实例，并行运行

多个模型，每个实例有自己的 Java 虚拟机。

- 要同时运行多个完全的NetLogo程序，方法见 [this answer](#) 。
- 也可以使用行为空间或控制 API 运行模型，同时启动多个无头 NetLogo 进程。

在以后的 NetLogo 版本，我们希望在以下方面改进多处理器/核心的支持：

- 允许同时打开多个模型，每个模型使用一个单独的线程，因此可以利用不同的处理器/核心。
- 修改行为空间，增加并行运行多个模型的选项，可以配置独立线程的数量，因此模型的运行就可以分散在可用的处理器/核心。

NetLogo 模型能分布运行在计算机集群上吗？

上个问题已经谈到了很多相同的事情。无法将单个模型分割到多个计算机上，但能使用行为空间或控制 API，使集群中的每台计算机运行单独的模型，

许多用户已经在集群上使用NetLogo，可以在[NetLogo Users Group](#)找到他们。

我想试试 HubNet，可以吗？

可以。有两种类型的HubNet。在计算机HubNet，参与者在通过常规网络连接的计算机上运行HubNet客户应用程序。在计算器HubNet（与TI联合研制），参与者使用TI的图形计算器和[TI-Navigator](#)教室学习系统。

查看用户手册的HubNet部分，了解[Calculator HubNet for TI-Navigator](#)的更多细节。
要了解 HubNet 的详细信息，参考 HubNet 指南。

使用问题

当我把速度滑动条一下子拖到最右边时，为什么模型看起来像停止一样？

NetLogo 让模型运行更快的唯一方法是减少视图更新。当你将速度滑动条向右拖动时，视图更新频率越来越小。由于视图更新花费时间，频率变小意味着速度更快。

然而更新少还意味着更新间隔变大，当间隔达到几秒时，看起来模型就像停止了。实际没有，它在全速运行，看看时钟计数器（tick counter）就知道了。（如果模型使用了时钟计算器就看它，否则看别的事物，如绘图等）

要感知发生的事情，最好慢慢的把滑动条拖到右边，而不是一下子拖到最右。如果发现在最右侧视图更新太少，那就别拖那么多。

怎么改变瓦片的数量?

在界面页的工具条按下 Settings... 按钮，出现对话框，在这改变世界的大小。

另一种较快的方式是使用 2D 视图左上角的三组黑色箭头。

我能使用鼠标在视图上画图 ("paint") 吗?

NetLogo 没有内建在视图上画图的工具。但只要在模型中加上几行代码，就可以有画图能力。要知道如何实现，查看模型库的 Code Examples 部分的 Mouse Example。同种技术也用来实现用户使用鼠标与模型交互。

另一种方法是使用一个特别的画图模型，例如James Steiner的Drawing Tool模型，可以在<http://ccl.northwestern.edu/netlogo/models/community/> 得到。

第三种方法是使用其他程序创建图像，然后导入模型。参见问题[Can I import a graphic into NetLogo?](#)的答案。

模型可以多大？模型能包含多少海龟、瓦片、例程、按钮等？

我们测试过使用几百兆 RAM 的模型，运行的很好。但还没有测试过使用上 G 字节 RAM 的模型，理论上应该没问题，但可能会遇到 Java 虚拟机和/或操作系统的某些限制（他们都有局限，或 bug）

NetLogo 引擎没有设定规模限制，然而默认有最大 1G 字节 RAM 的限制。

如果需要的话，可以增加这个上限：

Windows: 编辑"NetLogo.lax" 文件的这个部分，该文件在 NetLogo 文件夹

- # LAX.NL.JAVA.OPTION.JAVA.HEAP.SIZE.MAX
- # -----
- # allow the heap to get huge
- lax.nl.java.option.java.heap.size.max=1073741824

注意：在某些 Windows 98 或 Windows ME 上可能无效

Macintosh: 编辑 NetLogo 应用包的 Contents/Info.plist 文件。（在 Finder 中 Ctr+单击应用程序，在弹出菜单中选择"Show Package Contents"）。相关部分如下，第二个数值是上限：

- <key>VMOptions</key>
- <string>-XX:MaxPermSize=128m -Xmx1024M</string>

使用这种方法最大可以设为 2G。如果你的Mac是 64 位Intel处理器，运行Mac OS X 10.5，如果改变了某些附加选项，甚至可以大于 2G。给feedback@ccl.northwestern.edu 发信获

得帮助。

Other: 编辑脚本netlogo.sh(或它的拷贝), 将数值-Xmx 改为所需的数。

可以将 GIS 数据导入 NetLogo 吗?

可以。有些用户已经建立了使用光栅 GIS 数据的模型。(我们还不知道哪个用户使用了矢量 GIS 数据, 除非先转换为光栅数据)

一种简单方法是使用import-pcolors, 但这只能输入图像, 而不是其他格式的图。

NetLogo没有内置读取一般GIS格式的功能。然而有些用户使用文件输入/输出原语, 如file-open, 编写了NetLogo程序, 成功使用了GIS数据

还有一种可能是使用外部程序将GIS数据转换为NetLogo更易读的数据格式。这一点已经在[NetLogo Users Group](#) 讨论了几次了。我们鼓励使用NetLogo用于GIS的用户在用户组分享他们的问题和经验。

模型运行很慢, 怎么能快一些?

有一些方法不用修改代码结构, 可以使模型运行快一些:

- 使用基于滴答的视图更新, 而不使用连续更新.
- 将速度滑动条拖到最右 (通过减少视图更新频率, 加快模型)
- 如果你的模型占用了所有内存, 给计算机加点内存可能有用。如果模型运行时硬盘噪音很多, 可能模型需要更多内存。
- 海龟大小设为 1, 1.5, 或 2 , NetLogo 对这几个大小做缓存。

然而很多情况下要使模型运行更快, 需要修改程序。最常见的改进机会是有些计算涉及到所有海龟或所有瓦片, 经常模型此处可以修改, 使每步计算减少。如果需要帮助, 将模型或实现思路发给我们feedback@ccl.northwestern.edu, 我们可能提供帮助。[NetLogo Users Group](#)的成员也可能提供帮助。

注意使用run 和 runresult比直接运行慢得多。如果代码对性能敏感就不要使用这些原语。

能同时打开多个模型吗?

一个 NetLogo 实例只能同时打开一个模型。(我们计划在将来的版本对此做出修改)

然而可以同时打开多个 NetLogo 实例。在 Windows 和 Linux, 只要再次启动 NetLogo 即可。在 Mac, 需要在 Finder 中复制应用程序, 然后打开拷贝。(拷贝只占用一点硬盘空间)

能在运行时改变选择器的选项吗？

目前不能。我们计划在将来的 NetLogo 版本支持这个功能。

能将模型代码分到几个文件吗？

能。这是一个实验性功能，使用关键词 `_includes`。

编程问题

NetLogo 语言和 StarLogo 和 StarLogoT 语言有什么区别？

怎么把 StarLogo 或 StarLogoT 模型转换为 NetLogo 模型？

我们没有提供专门总结这些程序之间区别的文档。如果你以前使用 StarLogo 或 StarLogoT 建立了模型，我们建议你阅读[编程指南](#)学习 NetLogo，特别是“Ask”和“Agentsets”部分。看看模型库中的例子模型和代码例子也有所帮助。

如果将StarLogo 或 StarLogoT模型转换到NetLogo时需要帮助，尽管向[NetLogo Users Group](#)寻求帮助。也可以向我们feedback@ccl.northwestern.edu获得帮助。

NetLogo 语言与其他 Logo 语言有什么区别？

Logo 是个松散的语言家族，没有一致接受的 Logo 标准定义。我们相信 NetLogo 与其他 Logo 有足够多的相同部分，可以配得上 Logo 之名。当然 NetLogo 与其他 Logo 有一些不同之处。最重要的区别如下：

表面区别 (Surface differences)

- 数学运算优先级不同。中缀数学操作（像`+`*等）比有名报告器优先级低。例如在许多Logo里，`sin x + 1`被解释为`sin (x + 1)`。相反NetLogo像多数其他语言那样解释，解释为`(sin x) + 1`。
- `and` 和 `or` 报告器是特殊形式，不是一般函数，进行“短路”计算，即如果必要的话仅评估第 2 项输入。
- 例程只能在例程页定义，不能在命令中心交互输入。
- 报告器例程必须用[to-report](#) 定义。返回值的命令是[report](#)。
- 定义例程时，例程输入必须用`[]`，例如`to square [x]`。
- 变量名不使用标点。

最后三项区别由下面的例程定义演示：

most Logos **NetLogo**

```
to square :x  to-report square [x]
  output :x * :x report x * x
  end
```

内部区别 (Deeper differences)

- NetLogo 是固定范围, 而不是动态范围.
- NetLogo没有“word”数据类型 (Lisp叫做 “symbols”)。最终可能会加上, 但不太常用。我们有strings类型, 有些Logo使用“word”的地方, 我们使用strings。例如在Logo里可以写[see spot run] (a list of words), 但在NetLogo你必须写“see spot run” 串)或 [“see” “spot” “run”] (串列表)
- NetLogo’s 的[run](#) 可操作strings, , 但不能操作列表, 不允许重新定义例程。
- 如[if](#)和[while](#)的控制结构是特殊形式, 你不能自己定义它们的形式, 也就不同定义自己的控制结构 (NetLogo’ s 的[run](#)在这没用)
- 像多数Logo一样, 不能以函数为值。多数Logo提供了相似但一般性交叉的功能, 允许以列表形式传递和操作代码片段。NetLogo这方面能力受限。一些内建的UCBLogo-风格的模板实现相似目的, 如 sort-by [length ?1 < length ?2]
string-list。在某些情况下使用[run](#) and [runresult](#)可以, 但不像其他多数Logo那样, 它们只对串而不是列表操作。

当然 NetLogo 还包括其他多数 Logo 没有的功能, 其中最重要的就是主体和主体集合。

用以前 NetLogo 建立的模型怎么不能正常工作?

参考用户手册的[迁移指南](#)部分获得帮助。

为什么代码里有奇怪的字符?

NetLogo 只能在英语本地化 (“en” locale) 下工作。本地化是一组设置, 告诉 NetLogo 使用哪种语言, 以及如何显示日期和数值。在启动 NetLogo 之前需要切换到英语本地化。通常在操作系统的“区域设置”或“国际化”部分设置。

未来的 NetLogo 版本计划支持不同的语言和本地化。

怎样取一个数的负值?

下面几种方法都可以:

[\(- x\)](#)

[-1 * x](#)

[0 - x](#)

第一种方法需要括号。

海龟前进 1，但还在原来的瓦片上，为什么？

只有当海龟的方向是 90 的倍数（即正北、正南、正东或正西）时，前进 1 才能保证移动到新瓦片。

因为海龟可能不在瓦片的中心点，可能在角部。例如海龟靠近瓦片的西南角，面向东北，瓦片对角长为 $1.414\cdots$ （ 2 的方根），因此 `fd 1` 后海龟只是移动到同一瓦片的东北角。

如果你不想考虑这个问题，一种方法是以某种方式编程，使海龟总是停在瓦片的中心。看下一个问题。

怎样保持海龟在瓦片中心？

当海龟的 `xcor` 和 `ycor` 是整数时，它就在瓦片的中心。

使用下面的两个等价的命令之一，将海龟移动到当前瓦片的中心：

```
move-to patch-here
setxy pxcor pycor
```

如果之前不允许海龟偏离瓦片中心，则不需要用这些命令。

`Sprout` 创建的海龟在瓦片中心，例如：

```
ask n-of 50 patches [ sprout 1 [ face one-of neighbors4 ] ]
```

另一种使海龟在开始时处于瓦片中心的方式是使用下面的命令，它将海龟移动到随机选择的瓦片的中心

```
move-to one-of patches
```

一旦海龟位于瓦片中心，只要它的方向恰是 90 的倍数（即正北、正南、正东或正西），并且前进或后退的数量是整数，则它将一直位于瓦片中心。

参考模型库 `Code Examples` 部分的 `Random Grid Walk Example`，看看使用的代码片段。

patch-ahead 1 返回海龟当前所在的瓦片，为什么？

看前两个问题的答案，这是同一个原因。

“`ahead`”的含义与你期望的可能不一样。使用 `patch-ahead` 必须指定要查看的前方距离。如果海龟连续前进，你想知道海龟将要进入的瓦片，这个命令能帮你找到它。参考模型库 `Code Examples` 部分的 `Next Patch Example`。

怎样给海龟视场("vision")？

使用 `in-radius` 让海龟看到它周围的圆形区域。

有几个原语让海龟看到特殊的点。`patch-ahead` 让海龟看到正前方，如果希望海龟看到其他方向，使用 `patch-left-and-ahead` 和 `patch-right-and-ahead`。

如果要海龟有完整的锥形视场，使用in-cone原语。

如果海龟连续前进，也可以看到要穿越的下个瓦片。参考模型库 Code Examples 部分的 Next Patch Example。

海龟能感知画图层（drawing layer）的东西吗？

不能。如果要创建海龟可以感知的标志，使用 patch colors

我得到的是像 **0.1000000004** 和 **0.799999999999** 这样的数，而不是 **0.1** 和 **0.8**。为什么？

参考用户手册中编程指南的数学部分，在那儿讨论这个问题。

文档说random-float 1 可能返回 **0** 但不可能返回 **1**，如果我想包括 **1** 怎么办？

实际上不需要。即使可能返回 1，但它需要 2^{64} 抽样才会出现 1 次，你需要等上几百年才会看到 1 的出现。

然而，如果你真的需要 1，你可以在某个小数位上进行四舍五入。例如

```
print precision (random-float 1) 10  
0.2745173723
```

（如果使用这种方法，注意 0 和 1 出现的可能性是其他数的 1 半¹³。为什么会这样呢？例如只保留小数点后 1 位，0 和 0.5 之间的数取为 0，但 0.5 到 1.5 之间的数取为 1，后一个范围是 2 倍大。如果你想等概率得到 0, 0.1, 0.2, …, 0.9 和 1，一种方法是写random 11 / 10，这样所有 11 个数是等概率的。）

怎样防止两个海龟占据同一个瓦片？

参考模型库 Code Examples 部分的 One Turtle Per Patch Example。

怎样判断海龟是否死亡？

海龟死亡后，它变成nobody。nobody是NetLogo的一个特别值，用来表明海龟或瓦片不存在。例如：

¹³ 译者注：这里的论述似乎有点问题

```
if turtle 0 != nobody [ ... ]
```

也可以使用is-turtle?:

```
if is-turtle? turtle 0 [ ... ]
```

NetLogo 有数组吗?

当前版本的 NetLogo 中列表是真正的链表，而不是像以前版本中基于数组实现。

通过数组扩展可以使用真正的数组。参考用户手册的数组和表 (Arrays & Tables) 部分。

NetLogo 有哈希表或关联数组吗?

有，使用表扩展。参考用户手册的数组和表 (Arrays & Tables) 部分。

怎样使用不同的瓦片邻域 ("neighborhoods") (圆形, Von Neumann, Moore, 等)?

in-radius原语用来访问任何直径的圆形领域。

neighbors给出半径为 1 的Moore邻域。neighbors4 原语给出半径为 1 的Von Neumann邻域。

关于半径大于 1 的Moore 或 Von Neumann 邻域，查看模型库 Code Examples 部分的 Moore & Von Neumann Example。

怎样将一个主体集合转换为主体的列表，或相反？

如果希望列表元素有特定的顺序，使用sort 或 sort-by原语。编程指南的列表部分告诉你怎么做。也可查看模型库Code Examples部分的Ask Ordering Example。

如果需要列表元素随机顺序，见下面：

```
[self] of <agentset>
```

因为对主体集合的操作都是随机顺序，因此得到的列表是随机顺序。

要把主体列表转换为主体集合，使用turtle-set, patch-set, 或link-set原语。

如何停止 foreach?

要停止 foreach 的执行，需要定义一个只包含 foreach 的独立例程，例如：

```
to test
  foreach [1 2 3] [
```

```

if ? = 2 [ stop ]
print ?
]
end

```

这段代码只会打印数字 1。stop 从当前例程返回，后面的不再执行。（如果是报告器例程，使用 report 而不是 stop）

行为空间问题

怎样每 n 个滴答收集一次运行数据？

使用的go命令用上repeat，例如：

```
repeat 100 [ go ]
```

则 100 个时间步测量一次。本质是每个实验步等于 100 个模型步。

我变动在例程页声明的一个全局变量，但没作用。为什么？

可能是setup命令或setup例程使用了clear-all，使得行为空间设定的值被清除了。

一种办法是改变实验的 setup 命令，保存变量值，例如：

```

let old-var1 var1
setup
set var1 old-var1

```

这能奏效，因为即使是clear-all也不能清除用let设定的局部变量。

另一种方法是改变模型的 setup 例程，使用特别的清除命令只清除所要清除的东西。

为什么在 Excel 里某些结果被截掉了？

有些版本的Excel电子表格不能多于 256 列(参见关于这一问题的[a Microsoft support article](#))

可能的方法有：

- 使用新一些的 Excel，如 Excel 2007.
- 使用 Excel 之外的其他程序.
- 要求行为空间使用表格形式产生结果，而不是或者另外使用电子表格形式。(Excel 也能读取表格格式)
- 改变实验，使得结果的列数少一些

扩展问题

我编写了一个扩展，为什么编译器说找不到 org.nlogo.api?

编译时需要把 NetLogo.jar 加到类路径 (classpath) , NetLogo.jar 文件在 NetLogo 目录。