中学生编程

最简单好学的教材

蔡学镛

版本：0.6.3

前言

编写本书的目的，是让没有任何编程经验的读者（尤其是中学生），可以在很短的时间内开始动手编程，并产生兴趣。读者只需要有基本的电脑操作经验（例如：拖拽文件、修改文件名、切换输入法、收发电子邮件），并对英文有最基本的认识（约中学生程度），即可阅读本书学习编程。

所谓的「编程」，就是「编写程序」的意思，让计算机（或手机）依据我们编写出来们程序而运行。想编程，就必须使用某个编程语言，市面上有许多编程语言，我推荐 Red 语言，因为它简单好学，且功能强大。读完本书后，你将可以在微软 Windows 或苹果 macOS 的电脑上，用 Red 语言写出一些简单的程序，并让程序在 Windows、macOS 操作系统上运行。

编写本书时，最新的 Red 版本是 0.6.3，这个版本不支持苹果 iOS 和 Android（安卓）操作系统，所以用 Red 写出来的程序目前无法在手机和 iPad 上运行。但这部分的研发正在进行中，预计 2018 年开始，手机将也可以运行 Red 语言编写的程序。

本书共有三章，第一章教导 Red 的安装与使用，第二章教导 Red 语言的基本概念，第三章教导「图像用户界面」（GUI）程序设计。第三章相当有趣，相形之下第二章略显无聊，但如果没有第二章的基础，就没有办法理解第三章的许多内容。我尽量将第二章写得简洁，好让读者能快速进入第三章。本书的另一种阅读方式是，读完第一章就立刻跳到第三章，等到第三章阅读出现困难后，再回到第二章。

本书写作的方式，秉持一个理念：不只教你语言的用法，还教你解决问题的方法。所以你会在本书的范例中看到渐进的、解决问题的思路，这才是培养编程能力的重点。为了达到这个目标，且为了避免读者认知超出负荷，我有时候必须牺牲叙述的完整性与精确性。为此，我未来会编写一套内容更完整与更精确的书，教导 Red 语言与 GUI 程序设计，做为本书的补充（或者说是本书的进阶读物）。

本书目前的版本是 0.6.3，本书的版本号将与写书时最新的 Red 版本号一致。本书将持续改版，你现在正阅读的有可能不是最新版，如果你想知道本书最新版本的发布消息与下载网址，中国大陆的读者可以关注作者蔡学镛的新浪微博（@蔡学镛），网址在 http://www.weibo.com/rebol，其他地区的读者可以关注蔡学镛的 Twitter（@CaiXueYong），网址在 https://twitter.com/CaiXueYong。

阅读本书过程中有任何问题，可以在 Red 的 Gitter 中文专区与大家交流，网址https://gitter.im/red/red/Chinese。如果你学习得很好，没有任何问题，也欢迎到这个讨论区帮忙回答问题，协助其他人，帮助别人的过程也会让你进步。

Red 语言是免费的，这本电子书也是免费的，欢迎将这本书散布给你认识的每个人。我衷心希望这本书能开启大家的编程之路。

如何加入Red的Gitter中文讨论区

Red 语言在 Gitter 开了多个讨论区聊天室，每个聊天室都有不同的主题范围，但都使用英文交流。目前有一个讨论区是专门针对中文用户的，欢迎大家加入这个中文讨论区。

我们要通过 Github 登录 Gitter。如果你没有 Github 帐号，必须先申请一个。申请的 Gitter 帐号的过程需要用到电子邮箱，如果你没有电子邮箱，请去申请一个。你可以申请新浪邮箱、网易邮箱、QQ 邮箱、Outlook 邮箱等免费邮箱。申请邮箱的方式，请参照各邮箱公司的说明。

创建Github帐号

如果你还没有 Github 帐户，必须先创建一个。网页浏览器连接到 https://github.com/join，分别填入三个字段：用户名称（Username）、电子邮箱地址（Email Address）、密码（Password）。如果字段填入正确，该字段就会出现绿色的勾。

填写用户名称的时候请注意，不能使用中文，只能使用英文字符和数字和「-」，且「-」不能连续出现两个。用户名称必须是还没有被人在此网站用过的，如果你拿不定主意，可以用你名字的拼音后面加上生日当你的用户名称（例如 `WangMing20041215`）。填写电子邮箱地址的时候请注意，必须是还没被人在此网站用过的邮箱。设置密码时，必须至少七个英文字符，还必须至少包含一个数字。

当三个字段都填写完毕且正确，按下写著「Create an account」的绿色按钮，就可以马上建立一个 Github 帐号。接下来要验证邮箱主人是否你本人，现在去收取电子邮件，你会看到一封验证信，点选信中内容的「Verify email address」超链接，页面自动跳转到 Github，就完成验证了。此时你已经成功建立一个 Github 帐号，通过验证，并登录这个帐号了。如果还没登录，请在此时通过刚才注册的帐号和密码进行登录。

关注Red语言

在你已经登录 Github 的情况下，将浏览器连到网址 https://github.com/red/red，这是 Red 语言「开放源码」所在地。请按下页面右上角的「★Star」（使其转变为「★Unstar」），这表示你开始关注并重视 Red 语言。也请随时到这个网址看看 Red 最新版本的进展。

加入Red语言Gitter中文讨论区

在你已经登录 Github 的情况下，用浏览器连到网址 https://gitter.im/red/red/Chinese，这是 Red 技术的中文讨论区。点按最下面的橙色「SIGN IN TO START TALKING」，这时会跳出一个窗口，从中选取黑色的「SIGN IN WITH GITHUB」。自动跳转到一个新的页面后，按下绿色的「 Authorize gitterHQ」。接下来会自动跳回原来中文讨论区的页面，点按最下面橙色的按钮「JOIN ROOM」，你就加入这个讨论区了。以后可以在这里交流 Red 语言相关问题。

1

安装与使用Red

首先，你必须有一部台式计算机或笔记本电脑，在其上运行微软 Windows 或苹果 macOS 操作系统。（目前不能使用 iPad，因为其操作系统是 iOS，不是 macOS。但可以使用 Windows 平版）

Windows操作系统上准备Red环境

如果你使用 macOS 操作系统，请跳过此节，直接阅读下一节。

通过网页浏览器连到 http://red-lang.qiniudn.com/dl/win/red-063.exe ，即可下载 Red 语言交互式环境。由于 Red 非常小，可能只需要一秒就可以下载完成。

> 补充
>
> 你的浏览器可能会对下载执行文件的行为提出安全警告，请忽视这个警告，继续下载。

将刚才下载回来的 red-063.exe 文件，更名为 red.exe，移放到「家目录」下。Windows 操作系统的「家目录」是 C:\Users\<当前用户名>，把其中<当前用户名>取代成你在操作系统上真实使用的用户名，以我来说，我的用户名是「JerryTsai」，我的家目录是「C:\Users\JerryTsai」。我用「文件资源管理器」将 red.exe 拖拽到「此电脑 => 用户 => JerryTsai」。现在 red.exe 文件的完整路径是 C:\Users\JerryTsai\red.exe。

> 补充
>
> 「文件资源管理器」是 Windows 自带的管理程序，用来浏览并操作文件系统。你可以在「开始」菜单中找到它，或在操作系统桌面下方的工作条中找到它，其图标是一个黄色的文件夹。

由于这是从网络下载回来的程序，操作系统为了防止恶意的程序搞破坏，「可能」会锁定不让这类程序运行。你「可能」必须先解除操作系统对此程序文件的锁定。在 Windows 操

作系统中「解除锁定」的做法是在「文件资源管理器」中用鼠标右键点选 red.exe，选择菜单中的「属性」，然后在「常规」页选择「解除锁定」，再按下「确定」即可。

现在你可以在「文件资源管理器」中用鼠标连续双击 red.exe，程序就会开始运行。第一次运行这个程序时，需要花费约一分钟的时间做预处理，处理完毕之后，就会出现「>>」提示符号，这就是 Red 交互式控制台。以后在「文件资源管理器」中用鼠标双击 red.exe 会立刻出现「>>」，不再需要预处理。

补充

如果你的系统安装了杀毒软件，可能会发生误判的情况，导致 red.exe 被拦截无法运行。请察看杀毒软件的操作说明，取消对 red.exe 的拦截。

macOS操作系统上准备Red环境

如果你使用 Windows 操作系统，请跳过此节。

在 macOS 中打开「终端」，看到提示符号「$」，在终端中输入「pwd」，按下 enter 键，你就会看到当前目录，以我的例子是 /Users/jerrytsai：

```
SkittlesJerry:~ jerrytsai$ pwd
/Users/jerrytsai
```

接下来输入下面的命令：

```
curl http://red-lang.qiniudn.com/dl/mac/red-063 -o red;chmod +x ./red
```

按下 enter 后，会立刻下载 red-063 到当前目录，并将下载回来的文件改名为 red，添加执行权限，这些都会自动进行。请注意：由于程序非常小，可能不到一秒就下载完成，再度出现终端的提示符号「$」。

> 补充
>
> 「终端」是 macOS 自带的文字控制台，用来对操作系统下命令。运行在桌面下方最左边的「Finder」，在其「应用程序」中找到「实用工具」，终端就在实用工具中。

现在你可以在「Finder」中找到刚刚下载回来的 red 执行文件，用鼠标双击它，程序会开始运行。第一次运行这个程序时，需要花费约一分钟的时间做预处理，处理完毕之后，就会出现「>>」提示符号，这是 Red 交互式控制台。以后在「Finder」中用鼠标双击 red，会立刻出现「>>」，不再需要预处理。

Red交互式控制台

不管是Windows或macOS，现在你都应该会看到「Red交互式控制台」的提示符号「>>」，让我们马上写个程序，试试看一切是否正确。在 >> 的后面输入「1 + 2」（加号的前后必须有空格），然后按下enter（或return），你会看到下面的结果：

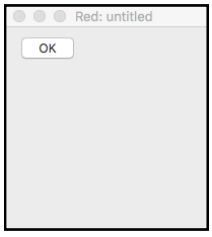
```
>> 1 + 2
== 3
```

下面再来稍微复杂一点的程序，印出1到5的整数（输入时，注意该有的空白）：

```
>> i: 1 while [ i <= 5 ] [ print i i: i + 1 ]
1
2
3
4
5
```

前面这两个例子都是文字输出的程序，下面来一个「图像用户界面」（GUI）程序：

```
>> view [ size 200x200 button "OK" [ unview ] ]
```



你会看到一个视窗，上头有个按钮。按下按钮，或按下视窗上方角落的关闭按钮，就可以回到Red交互式控制台。

> 补充
>
> 本书采用的截图，都是在macOS上进行的。如果你采用的操作系统是Windows，画面可能会略有差异。

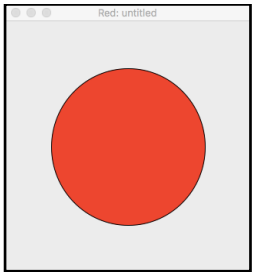
让我们做个小时钟：

```
>> view [ t: text 250x50 font-size 24 rate 1 on-time [ t/data: now/time ] ]
```

你会看到一个程序，显示现在的时间。按下按钮，或按下视窗上方角落的关闭按钮，就可以回到Red交互式控制台。

来个简单的绘图程序吧：

```
>> view [ box 300x300 draw [ fill-pen red circle 150x150 100 ] ]
```



这些程序都是让你试验并简单体会 Red 之用，在此不解说。等第三章看完，你自然会完全理解这几个代码。

Red脚本文件

另一种执行方式，是把程序写成一个脚本文件，保存起来，以后可以多次执行，不用每次都反复输入。

编写脚本文件，需要使用「编辑器」。在 Windows 上可以使用操作系统自带的「记事本」，在 macOS 上可以使用操作系统自带的「文本编辑」。现在就打开你的编辑器。

Red 脚本文件开头必须是「Red [ ]」。以上面的小时钟为例，写成脚本就变成：

```
Red [ ]
view [ t: text 250x50 font-size 24 rate 1 on-time [ t/data: now/time ] ]
```

在你的编辑器内输入上面的程序，把这个程序保存为 clock.red，放到和 red （或 red.exe）执行文件相同的目录下，以后我们就可以在 Red 交互式控制台用下面的方式执行这个程序了：

```
>> do %clock.red
```

你会看到一个程序，显示现在的时间。按下按钮，或按下视窗上方角落的关闭按钮，就可以回到 Red 交互式控制台。

> 补充
>
> 有些编辑器为了美观，会做出下面的行为，这两点都会导致 Red 脚本程序无法执行，请特别注意：
>
> 1. 编辑器可能会修改你的文字（例如把双引号 "" 变成“ ”）
> 2. 编辑器可能会添加一些属性描述（例如文字颜色），把文件由「纯文本」变成「多信息文本」。
>
> Windows 平台自带的「记事本」不会做这样的事，但 macOS 自带的「文本编辑」可能会做这样的事。
>
> 如果你采用 macOS 「文本编辑」当作编辑器，必须注意上面的第二点。请确定「文本编辑」上方菜单的「格式」菜单中出现「制作多信息文本」选项，这表示目前已经是纯文本。如果你看到的选项是「制作纯文本」，你必须选取这个选项，将目前的文本转成纯文本。

2

Red语言基本概念

本章介绍 Red 语言常用基本数据类型（datatype），并介绍这些数据类型相关的重要特色。

计算后不变的字面值

在下面的例子中，我们输入整数「100」，然后按下「enter」键，随即看到运算的结果（还是原来的整数）。如下所示：

```
>> 100
== 100
```

> 补充
>
> 「==」用来告诉你运算结果。如果运算后没有返回值（结果值），就不会出现 ==。下面的例子，目的是在一个文件内（%test.txt）写入一个名字（"Jerry Tsai"），这个例子执行后就没有返回值：
>
> ```
> >> write %test.txt "Jerry Tsai"
> ```
>
> 下面的程序用来读出刚刚写入的结果，有返回值：
>
> ```
> >> read %test.txt
>
> == "Jerry Tsai"
> ```

整数的运算结果会维持不变（也就是说，运算后的返回值和原来的值一样），这类「运算后前后都一样」的数据类型相当多，下面列出某些有此特质的数据类型：

- 整数（integer!）：例如 100
- 浮点数（float!）：也就是小数，例如 3.14
- 百分数（percent!）：例如 50%
- 字符（char!）：例如 #"J"、#"语"
- 字符串（string!）：例如 "Jerry Tsai"、"Red语言"

- 网址（url!）：例如 http://www.abc.com
- 电邮（email!）：例如 someone@abc.com
- 文件（file!）：例如 %resume.pdf、%clock.red
- 标签（tag!）：例如 <html>、</p>、<img source="flower1.png">
- 二元（binary!）：用来表示原始的二进制数据，例如#{00420AFF08600000}
- 修饰字（refinement!）：例如 /abc、/skip
- 期号（issue!）：例如 #DF-2324、#00FF00
- 时间（time!）：例如 16:30:25.28、12:00
- 日期（date!）：例如 2017-12-31、28-Jun-2017/13:09:30+8:00
- 数组（tuple!）：通常用来表示颜色或 IP 地址，例如 255.0.0、127.0.0.1
- 数对（pair!）：通常用来表示屏幕上的座标、体积、分辨率，例如 1024x768
- 区块（block!）：例如[1 2 3]、[someone@abc.com 1988-12-13 100]

以上这 17 个数据类型的值，可以说是最简单的，我称之为「计算后不变的字面值」，因为它们：

1. 字面的内容就是其值。这十七个数据类型的值每个都具备字面上的特征，一看字面就知道它属于什么数据类型，例如一看到 % 出现在前就知道是文件（file!），一看到用<和>包夹就知道是标签，一看到 / 出现在前面就知道是修饰字（refinement!）。有这种字面特色的值就称为字面值（literal）。
2. 计算前后不变。

看过这 17 个数据类型之后，你可能也注意到了，Red 语言的数据类型名称都以「!」结尾，相当好识别。

> 补充
>
> 本书写作过程会交替使用「运算」、「计算」、「执行」、「运行」这四个词，它们的意思一样。

单字、设字、原字

简单地看过这 17 个数据类型之后，接下来要介绍的数据类型是单字（word!）。单字也是字面值（看到字面文字就知道它的值和数据类型），但单字和上述 17 种数据类型不同的是：单字计算前后会改变。

一个单字可以和另一个值发生关连，这个关连我们称为「绑定」（binding）。对一个单字进行运算，就会得到其绑定值（但如果绑定值是函数的话，效果不同，稍后再详细说明）。例如，执行圆周率 pi 的结果如下：

```
>> pi
== 3.141592653589793
```

pi 的数据类型是 word!，运算 pi 的结果就是得到它所绑定的值，也就是 3.141592653589793 这个浮点数（float!）。下面再试试运算另一个单字 green（绿色），会得到一个有三个成分的数组：

```
>> green
== 112.191.65
```

随便拿一个没有定义（也就是没有绑定值）的单字来运算，会得到错误。如下面的例子所示：

```
>> age
*** Script Error: age has no value
```

我们可以通过下面的方式，为还没有绑定值的单字绑定一个值，或者为已经绑定的单字修改其绑定值：

```
>> age: 18
== 18
```

上面的例子中，我把 age 这个单字绑定到一个整数值 18。用下面的方式，试试看是否已经绑定成功：

```
>> age
== 18
```

特别注意，「age:」这四个字符（包含英文冒号）整体是一个值，其数据类型是设字（set-word!）。设字也是一种字面值，它字面上的特征是「单字最后紧跟著一个英文冒号」。

当运算的时候遇到设字，就会「先计算出紧跟其后的一个值」，然后把这个值绑定给「设字所对应的单字」。所以「age: 18」的运算结果，就是把 age 绑定为 18。

通过设字的方式进行值的绑定，在完成绑定后，设字还会把这个值再次当作设字的计算结果，所以会看到「== 18」。因此我们可以连续放多个设字，让这些设字对应的单字一起被绑定到相同的值。

```
>> age1: age2: 18
== 18
>> age1
== 18
>> age2
== 18
```

设字的后面要有空白（或其他适当的分隔符号），否则可能会被误认为是其他数据类型的值。例如，下面的例子，age:和18之间没有空格，「age:18」整体被视为一个值:

```
>> age:18
== age:18
```

age:18 字面特征符合网址（url!）的要求，于是被视为网址，这误会大了。

原字（lit-word!）也是一种字面值，字面上的特色是「单字前面加上单引号」，例如 'line。原字计算后的结果会变成相应的单字:

```
>> 'line
== line
>> 'name
== name
```

函数与参数

上述这些单字除非重新绑定，否则不管运算多少遍，值都会一样，因为绑定到一个「计算前后一样」的值。但有些单字绑定到函数，执行单字就等于执行这个函数，如下所示：

```
>> now
== 28-Jun-2017/13:09:30+8:00
```

now 是一个单字，它的绑定值是「取得现在时刻」的函数。每次执行 now，就会执行这个函数，而函数执行的结果可能会每次都不一样。例如下面再执行一次 now，时间就变了：

```
>> now
== 28-Jun-2017/13:09:32+8:00
```

函数如果需要参数，参数必须紧跟其后。前面例子的 now 不需要参数，下一个例子的 type? 函数（精确的说法是 type? 单字，其绑定值是一个函数）执行时就需要一个参数。type? 函数的作用是取得「后面参数计算后的值」的数据类型。如下所示：

```
>> type? 100
== integer!
>> type? pi
== float!
```

100 计算后的值还是 100，其数据类型是 integer!。

pi 虽然是 word!，但其计算后的结果是 3.141592653589793，这是 float!，所以 type? 函数的计算结果不会是 word!，而是 float!。

再看另一个例子。? 函数（精确的说法是 ? 单字，其绑定值是一个函数）执行时就需要一个参数。? 函数的作用是取得后面参数（而不是「后面参数计算后的值」）的说明。例如下面的操作，可以取得 type? 的用法：

```
>> ? type?
USAGE:
     TYPE? value
...略
```

看过不需要参数的函数，也看过需要一个参数的函数，再来看需要两个参数的例子。as-pair 需要两个参数，它会把这两个数当成 X 和 Y，组合成一个数对（pair!）：

```
>> as-pair 1024 768
```

```
== 1024x768
```

Red 语言提供相当多的函数（严格来说是单字，其绑定值是函数）。想知道有哪些函数，可以通过 what 函数：

```
>> what
     %          op!           Returns what is left over when one value is ...
     *          op!           Returns the product of two values.
     **         op!           Returns a number raised to a given power (ex...
     +          op!           Returns the sum of the two values.
     -          op!           Returns the difference between two values.
     /          op!           Returns the quotient of two values.
… 省略 …
     about      function!     Print Red version information.
     absolute   action!       Returns the non-negative value.
     acos       function!     Returns the trigonometric arccosine.
     action?    function!     Returns true if the value is this type.
     add        action!       Returns the sum of the two values.
     all        native!       Evaluates, returning at the first that is no...
     all-word? unction!       Returns true if the value is any type of all...
     also       function!     Returns the first value, but also evaluates ...
… 省略 …
```

路径

路径（path!）数据类型的值，是由两个以上的节所组成，第一个节必须是单字，后面跟著一到多个单字或数字（还有其他可能，本书不介绍），每个节之间用「/」隔开，例如「face/offset/x」。

通过路径取得成分

下面的例子中，我们把 loc 绑定到一个数对（pair!）。通过对路径「loc/x」或「loc/1」进行运算，我们就可以取得这个数对的第一个数字。通过对路径「loc/y」或「loc/2」进行运算，我们就可以取得这个数对的第二个数字。

```
>> loc: 10x4251
== 10x4251
>> loc/x
== 10
>> loc/y
== 4251
>> loc/2
== 4251
```

上面例子的路径都只有两节（只需要一个/）。下面展示三节路径的例子:

```
>> info: [ 2017-12-18 someone@abc.com [ 18 A ] "Jerry" a/b/c 10:20:30]
== [2017-12-18 %someone@abc.com [ 18 A ] "Jerry" a/b/c 10:30:30]
>> info/1/year
== 2017
>> info/2/host
== "abc.com"
>> info/2/user
== "someone"
>> info/3/2
== A
>> info/4/1
== #"J"
>> info/5/3
== c
>> info/6/hour
== 10
```

从上面的例子可以看出：我们可以使用路径的方式，取得区块、路径、字符串、电子邮件、日期…等数据类型内部的某元素、或某成分、或某节。

除了上面的各种用法，下面展示另一种使用路径的例子，取得对象（object!）值内的某单字的绑定值：

```
>> system/version
== 0.6.3
```

对象（object!）是一种特殊的数据类型，它是许多单字的集合，而对象内的每个单字可以有一个绑定值。上面的例子中，system 是一个本来就存在的对象，version 是此对象内定义的一个单字，其绑定值是 0.6.3。

通过路径调用函数时指定修饰字

下面展示最后一种使用路径的例子。前面看过 now 函数（严格来说是 now 单字，绑定值是一函数），用来取得现在的时间（含日期）。

```
>> now
== 17-Jul-2017/16:45:48+08:00
```

我们可以通过下面的路径，来表示我们想要的是现在的时分秒，不需要日期：

```
>> now/time
== 16:47:01
```

或者用下面的方式，表示我们要的是现在的日期，不需要时分秒：

```
>> now/date
== 17-Jun-2017
```

对于「now/time」和「now/date」这样的路径表示法，由于 now 被绑定到函数，而不是之前例子看到的区块、字符串等数据类型，所以 date 和 time 不是用来表示内部的元素或成分。而且 now 也不是被绑定到对象，所以 time 和 date 不是对象内的单字。

在这个例子中，date 和 time 会被视为调用 now 函数时传入的一种「特殊参数」，这种特殊参数会被转换成「修饰字」（refinement!）数据类型。想知道某函数有哪些修饰字可以使用，可以通过 ? 函数：

```
>> ? now
USAGE:
     NOW

DESCRIPTION:
     Returns date and time.
```

```
     NOW is a native! value.

REFINEMENTS:
     /year         => Returns year only.
     /month        => Returns month only.
     /day          => Returns day of the month only.
     /time         => Returns time only.
     /zone         => Returns time zone offset from UCT (GMT) only.
     /date         => Returns date only.
     /weekday      => Returns day of the week as integer (Monday is day 1).
     /yearday      => Returns day of the year (Julian).
     /precise      => High precision time.
     /utc          => Universal time (no zone).

RETURNS:
     [date! time! integer!]
```

下面的例子中，我们调用 now，使用两个修饰字 /time 和 /precise，表示我们要取得现在时间（不需要日期），而且秒数要精确（precise）：

```
>> now/time/precise
== 17:35:09.206777
```

当有多个修饰字的时候，次序可以调动，效果一样：

```
>> now/precise/time
== 17:35:18.619552
```

有些修饰字可能会需要参数，看下面的例子。我们先在一个文件中写入一段文字：

```
>> write %test.txt "I love coding!!!"
```

然后我们通过下面的方式，把文件读取出来：

```
>> read %test.txt
== "I love coding!!!"
```

读出来的结果是正确的，正是我们刚刚写入的文字内容。现在我们通过 /seek 修饰字，来表示我们不要从头读取文件内容，我们要从「某个位置」开始读取。所以使用 /seek 修饰字的时候，必须搭配一个参数指定开始读取的位置。

```
>> read/seek %test.txt 7
== "coding!!!"
```

现在我希望读取时不要全部读取（不要读取到尾部），这时要使用 /part 修饰字。这个修饰字需要一个参数，指定要读取的长度：

```
>> read/seek/part %test.txt 7 6
== "coding"
```

两个修饰字的顺序也可以改变，不过相应的参数次序也必须改变：

```
>> read/part/seek %test.txt 6 7
== "coding"
```

设径

前面的例子告诉我们，路径可以用来：

1. 取得某个复合数据的内部成分或元素
2. 访问对象的内部单字
3. 调用函数

路径这三大功能中，前两者可以有对应的「设径」（set-path!）数据类型。路径加上冒号，就成了设径（set-path!），设径也是一种字面值。通过设径，可以设置某值的内部成分，例如：

```
>> pt: 100x200
== 100x200
>> pt/x: 0
== 0
>> pt
== 0x200
>> system/version
== 0.6.3
>> system/version/3: 4
== 4
>> system/version
== 0.6.4
```

请注意，如果路径代表函数调用，则不可以使用对应的「设径」格式。例如 now/time 是函数调用，不具有对应的设径，所以下面的程序运行时会报错：

```
>> now/time: 100
*** Script Error: unsupported type in now/time: set-path
```

对于 Red 语言来说，路径和设径是相当动态的机制，且一个语法有多种实际含意，使用时一定要特别小心。

运算子

red 语言的函数有多种数据类型，分别是 native!、op!、function!、action!、routine!，初学者不太需要知道它们的差别，目前只需要知道：调用函数时一般都是函数在前，参数在后，唯一的例外是运算子（op!）。下面是一般函数的例子：

```
>> add 1 2
== 3
```

op! 数据类型比较特殊，这类函数一律需要两个参数，一个在前，一个在后，例如下面的「+」就是一个运算子：

```
>> 1 + 2
== 3
```

绑定到 op! 数据类型的单字非常少，而且大多看起来很像符号，包括 +（加） -（减） *（乘） /（除）。通过下面的方式可以看到所有绑定到的 op! 数据类型的单字：

```
>> ? op!
    +       => Returns the sum of the two values.
    %       => Returns what is left over when one value is divided by a...
    <<      =>
    or      => Returns the first value ORed with the second.
    =       => Returns TRUE if two values are equal.
    *       => Returns the product of two values.
    /       => Returns the quotient of two values.
    -       => Returns the difference between two values.
    <       => Returns TRUE if the first value is less than the second.
    >       => Returns TRUE if the first value is greater than the second.
    <=      => Returns TRUE if the first value is less than or equal to...
    <>      => Returns TRUE if two values are not equal.
    >=      => Returns TRUE if the first value is greater than or equal...
    >>      =>
    **      => Returns a number raised to a given power (exponent).
    //      => {Wrapper for MOD that handles errors like REMAINDER. Neg...
    ==      => Returns TRUE if two values are equal, and also the same ...
    =?      => Returns TRUE if two values have the same identity.
    >>>     =>
    and     => Returns the first value ANDed with the second.
    xor     => Returns the first value exclusive ORed with the second.
    is      => Defines a reactive relation whose result is assigned to ...
```

当正常的函数遇上 op! 的时候，op! 比较强势。下面的计算结果是 2 * 3 先计算，而不是「add 1 2」先计算，所以结果是 7 不是 9。

```
>> add 1 2 * 3
== 7
```

上面的计算次序相当于下面这样:

```
>> add 1 (2 * 3)
== 7
```

圆块与区块

如果你希望 add 比 * 先计算，可以把「add 1 2」放进一个圆块（paren!）中，如下所示：

```
>> (add 1 2) * 3
== 9
```

一旦被放进圆块中，就导致 2 和 3 无法先做乘法计算，因此调整了计算次序。圆块和区块非常像，两者也都是字面值，差别在于圆块使用圆括号（）。

圆块（paren!）和区块（block!）行为上最主要的差异是：

* 圆块会被主动计算「内部」，把「内部计算的最后一个结果值」当作最终的计算结果。

* 区块不会被主动计算内部，只计算「外部整体」，计算结果前后一样。

请看下面的例子：

```
>> (1 + 2)
== 3
>> [1 + 2]
== [1 + 2]
>> [1 + 2 3 + 4]
== [1 + 2 3 + 4]
>> (1 + 2 3 + 4)
== 7
```

如果希望计算区块的内部，可以采用 do 函数：

```
>> do [ 1 + 2 3 + 4 ]
== 7
```

这时，区块就如同圆块一样。

请注意，不管是圆块或区块，只要是计算其内部，得到的返回值都是最后一个算出来的值。所以上面的例子会得到 3 + 4 的结果，也就是 7。

计算区块内部，除了可以采用 do 函数，也可以采用 reduce 函数，两者的差异是，do 函数传出最后一个值，但 reduce 传出整个区块：

```
>> reduce [ 0 1 + 2 3 + 4 ]
== [0 3 7]
>> reduce [ 'now now ]
```

```
== [now 21-Jul-2017/11:43:06+08:00]
```

reduce 是很常用的函数，一定要熟记。第三章最后一节「小时钟」会用到 reduce。

逻辑与空值

介绍三个特殊单字（true、false、none）和与其对应的两个数据类型(logic!、none!)。true 和 false 都是单字，这两个单字分别绑定到代表「真」和「假」的值，它们的数据类型是「逻辑」(logic!)。none 是一个单字，这个单字绑定到代表「空」的值，其数据类型是「空值」(none!)。

许多计算也会得到 true、false、none 的值，例如：

```
>> 1 = ( 3 - 2 )
== true
>> now/year < 2016
== false
>> data: [ 1 2 3 ]
== [1 2 3]
>> data/4
== none
```

true、false 的值常被使用来当作 and、or 这两个 op! 的参数，如下所示：

```
>> true and true
== true
>> true and false
== false
>> false and true
== false
>> false and false
== false

>> true or true
== true
>> true or false
== true
>> false or true
== true
>> false or false
== false
```

对于 and （且）计算，前后两个参数都为 true 时，才得到 true，其他情况都得到 false。对于 or （或）计算，前后两个参数都为 false 时，才得到 false，其他情况都得到 true。

true 和 false 也常用来当作 not 函数的参数，not 函数的意思是相反：

```
>> not true
== false
>> not false
== true
```

true 和 false 也常用在 if 和 either 这两个函数上。对于 if 来说，需要两个参数，第一个参数如果是 false 或 none，就表示条件不成立。如果是 true 或其他任何值，就表示條件成立。条件成立的话，就执行第二个参数（是个区块）内的程序，以下面的例子来说，就是在控制台上打印（print）出「AM」字样。

```
>> n: now/time
>> if n/hour < 12 [ print "AM" ]
```

> 补充
>
> 时间数据类型（time!）的单字，可以通过 hour 路径（例如上面例子的 n/hour），取得「小时」的部分。

对于 either 来说，需要三个参数，第一个参数和 if 一样，用来判断条件成立或不成立，条件成立的话，就执行第二个参数内的程序，条件不成立的话，就执行第三个参数内的程序。

```
>> n: now/time
>> either n/hour < 12 [ print "AM" ] [ print "PM" ]
```

true 和 false 也常用在 while 和 until 这两个函数上。

对于 while 来说，需要两个参数，第一个参数也是条件判断，但必须是区块。while 函数会计算此区块的「内部」（跟 do 一样）得到一个值，根据这个值来判断条件是否成立，成立就运行第二个参数（也是一个区块）的内部，不成立就结束这个函数。

下面的例子中，把 i 设置为 1，判断是否 i 小于或等于 5，成立则打印当前 i 的值，并把 i 的值加 1。i 的值会一直加 1，直到「小于或等于 5」不成立。

```
>> i: 1
== 1
>> while [ i <= 5 ] [ print i i: i + 1 ]
1
2
3
4
```

```
5
```

> 补充
>
> 「i: i + 1」的计算方式是：先遇到设字「i:」，于是计算后面的值，再把这个值设置（绑定）给 i。这个值的计算方式是，取得现在 i 的值，将其加 1。整体来看，就是把 i 的值加 1。

上面的程序用 until 取代 while，可以改写成：

```
>> i: 1
== 1
>> until [ print i i: i + 1 i > 5 ]
1
2
3
4
5
```

until 只需要一个参数，这个参数是区块。until 函数类似 do 函数，会进入区块中执行，差别在于 do 只做一次，但 until 可以做一次以上，直到区块内最后一个值的结果（以本例来说就是 i > 5 的结果）是「条件成立」才停止。

case/switch

如果我们要写一个会打招呼的程序，能根据时间区段说出「早上好」、「下午好」、「晚上好」、「怎么不睡？」，程序可以这么写：

```
t: now/time
h: t/hour
either (h >= 6) and (h < 12) [
    print "早上好"
] [
    either (h >= 12) and (h < 18) [
        print "下午好"
    ] [
        either (h >= 18)  [
            print "晚上好"
        ] [
            print "怎么不睡？"
        ]
    ]
]
```

虽然这个程序是正确的，但写法不好，因为每次条件不成立就要判断更多条件，这样子层层嵌套，让程序结构越来越不好理解。下面改用 case 函数改写，变得更清晰：

```
t: now/time
h: t/hour
case [
    (h >= 6) and (h < 12) [
        print "早上好"
    ]
    (h >= 12) and (h < 18) [
        print "下午好"
    ]
    (h >= 18)  [
        print "晚上好"
    ]
    true [
        print "怎么不睡？"
    ]
]
```

case 只需要一个参数，是一个区块。case 会逐一计算区块内的值，一旦「条件成立」，就会执行其后的区块内部。一旦「条件不成立」，就会跳过其后的区块，继续计算下一个条件。

在这个例子中，当前面三个条件都不成立時，就会计算到 true，而这个单字的计算结果是是「真」值，也就是条件成立，会打印「怎么不睡？」

上面的程序可以用 switch 改写如下：

```
t: now/time
switch t/hour [
    0 1 2 3 4 5 [
        print "怎么不睡? "
    ]
    6 7 8 9 10 11 [
        print "早上好"
    ]
    12 13 14 15 16 17 [
        print "下午好"
    ]
    18 19 20 21 22 23  [
        print "晚上好"
    ]
]
```

switch 需要两个参数，算出的第一个结果值（本例中就是 t/hour 计算出来的值），会拿来用在第二个参数（是个区块）内进行比对，比中了就会找出后面出现的第一个区块，运行该区块内的程序。

对于系列类型的基本操作

有相当多数据类型都属于系列（series）类型，通过下面的方式，可以找出这些数据类型：

```
>> ? series!
SERIES! is a typeset! value: make typeset! [block! paren! string! file! url!
path! lit-path! set-path! get-path! vector! hash! binary! tag! email! image!]
```

其中字符串（string!）和区块（block!）是最值得我们注意的系列类型。系列类型有一个共通特色：它们都是由一系列的值组成的，一个接著一个串在一起。因此，它们都可以进行在某位置插入某个值、删除其中某个值的操作。通过下面的例子，我们可以学到三个常用函数：append（在最后的位置插入）、insert（在当前的位置插入）、remove（在当前的位置移除）：

```
>> name: "Sean"
== "Sean"
>> family-name: "Lee"
== "Lee"
>> append name family-name
== "SeanLee"
>> name
== "SeanLee"
>> data: [ 2001-1-26 male ]
== [26-Jan-2001 male]
>> insert data name
== [26-Jan-2001 male]
>> data
== ["SeanLee" 26-Jan-2001 male]
>> remove data
== [26-Jan-2001 male]
>> data
== [26-Jan-2001 male]
>> append data [ 1 2 ]
== [26-Jan-2001 male 1 2]
```

这几个例子很容易理解，我不再赘述。但请一定要熟悉这三个函数的用法，它们实在太常用了。特别注意，通过 append 或 insert 在区块内插入另一个区块时，被插入的区块会被展开。所以下面的例子得到 [1 2 3 4] 而不是 [1 2 [3 4]]。

```
>> data: [ 1 2 ]
== [1 2]
>> append data [ 3 4 ]
== [1 2 3 4]
```

3

GUI程序设计

这一章介绍 GUI 程序设计。GUI 是「图像用户界面」的意思，读音接近「估以」。这里出现的程序范例，你都可以通过「Red 交互式控制台」执行，或通过「脚本文件」的方式执行。关于这两种执行方式，请复习第一章。

VID与view函数

请执行下面的程序。想关闭这个程序，用鼠标点按视窗的关闭按钮即可。

```
view [ button "OK" ]
```

在上面的例子中，我们通过使用 view 函数（严格来说是 view 单字，其绑定到一个函数），表示我们想要创建并显示一个视窗。view 函数需要一个参数，其类型是区块（block!），用来描述视窗的内容。在这个例子中，[button "OK"] 就是 view 的参数。参数必须紧跟在其函数之后。

[button "OK"] 虽然看起来一长串，但整体是一个值，其数据类型是区块（block!）。区块类型有一个特色，就是用左右方括弧 [ ] 将一些其他值包围起来。在这个例子中，此区块内包含两个值，分别是 button 和 "OK"。其中 button 的数据类型是单字（word!），而 "OK" 的数据类型是字符串(string!)。

[button "OK"]的意思是视窗内要有一个按钮（button），按钮上面的文字是「OK」。其中 button 是「样式名称」(style name)，"OK" 是该样式的选项设置之一。

> 强调
>
> 每个样式都会用来创立一个视觉元件（face!），每个选项设置都会被纪录到此视觉元件中。

这个区块（也就是 view 的参数）是自成一格的语言，叫做 VID（Visual Interface Dialect）视觉界面方言。之所以说它是方言，是因为它沿袭了 Red 语言的数据类型，但语法不尽相同。VID 的语法是为了简化视觉界面编程而专门设计的。

强调

Red 语言是为了一般编程目的而设计的，VID 是为了GUI 编程目的而设计的，它们是两个不同的语言，但彼此联系密切，且语法有颇多相似之处，所以说 VID 是 Red 语言的方言。

要特别注意，某些中文输入法或编辑器可能会自作主张地将简单的双引号""转变美观的“”，这会导致运行时出错，所以写程序的时候，一定要用「代码编辑器」，尽量不要用「文字编辑器」，更是绝对不要用「文书处理器」。

「文字编辑器」和「文书处理器」常常会自作主张地转变文字，或插入一些字体颜色等信息，这些文字和额外的信息固然会让文字呈现得更加美观，但对程序来说却是致命的。「代码编辑器」是专门为编写代码设计的，所以不会犯这些美丽的错误。

补充

第一章为了不让大家下载太多程序，只教导使用操作系统自带的编辑器（也就是「记事本」与「文本编辑」）。但如果你想专业一点，我推荐使用微软的 Visual Studio Code，这是一款很方便的代码编辑器，请自行上网搜索下载。

如果你希望按钮上的文字是中文，就把 OK 换成中文，如下所示：

```
VIEW [Button"确定"]
```

从上面这个例子同时可以看出，Red 语言是不区分英文大小写的，所以我可以把 view 写成 VIEW，把 button 写成 Button，效果完全一样。

补充

全角和半角是有区别的，我们不可以把半角的 view 写成全角的ｖｉｅｗ。也不可以使用全角的空白、冒号、引号、括号来代替其半角的相应字符。

从上面这个例子还可以看出，「有一些」空格是没有作用的，这些空格的存在只是为了让我们阅读时更清晰。例如：我把 [ ] 内的空白全都删除了，程序的效果还是一样。

到目前为止，我们程序上的按钮都没有实际作用，现在让我们为按钮加上一个简单的功能：当按钮被按下之后，视窗就会被关闭。代码如下所示：

```
view [ button "OK" [ unview ] ]
```

这次新增的选项设置是 [unview]。unview 是一个函数（严格来说是一个单字，绑定到一个函数），它的作用和 view 刚好相反：把视窗关闭。

现在 button 有两个选项设置，分别是 "OK" 和 [unview]。"OK" 描述的是按钮上面的文字，[unview] 描述的是按钮被按下后要执行的 Red 程序。

一个样式名称后面可以指定多个选项设置，且这些选项设置的顺序可以自由替换。例如，下面的例子中，我们把上面例子的两个选项设置次序对调了，但运行的结果还是一样：

```
view [ button [ unview ] "OK" ]
```

一个 VID 内可以有多个样式，每个样式后面可以跟著数量不等的选项设置。在下面的例子中，我们使用了两个 button 样式。

```
view [button "现在时刻?" [ print now/time ] button "结束程序" [ unview ]]
```

在上面的程序中，按下第一个按钮会在控制台上打印出（print）现在时刻（now/time）。按下第二个按钮会关闭视窗（unview）。

之前提到过，适当地使用空白可以帮助程序更清晰好懂。除了使用空白来让程序更清晰，也可以使用换行（return）或跳格（tab）。上面的程序写成下面这样，执行效果一样，但更容易阅读。

```
view [
    button "现在时刻?" [ print now/time ]
    button "结束程序" [ unview ]
]
```

视觉元件的体积与颜色

前面已经看到过，样式后面跟著字符串，则字符串会被当作视觉元件上面的文字；样式后面跟著区块，则区块会被当作视觉元件被触发时的 Red 代码。在下面的例子中，我们使用整数当作选项设置，此整数会被当作视觉元件的「宽度」，其单位是像素（pixel）。元件的高度由系统自动决定：

```
view [ button "OK" 100 ]
```

> 补充
>
> 屏幕上一个发光的点就是一个像素。

如果我们想要同时指定宽度和高度，可以用数对（pair!）。下面的代码中，100x50 就是一个数对，表示宽为 100，高为 50。

```
view [ button "OK" 100x50 ]
```

不管数对或整数，都是用来指定视觉元件的大小，因此不能重复出现，否则会报错，如下所示：

```
view [ button "OK" 100x50 100 ]
*** Script Error: VID - invalid syntax at: [100]
```

想指定视觉元件的颜色，可以用数组（tuple!），颜色的表示方式是红色、绿色、蓝色三个颜色的成分（0~255）之间用小数点连接。例如：255.0.0 表示红色最强，绿色和蓝色都没有；255.255.255 表示三个颜色成分都最强，混在一起就是白色。0.0.0 自然就是黑色了。

在下面的代码中，我们试图让按钮的底色是红色。

```
view [ button "OK" 255.0.0 ]
```

上面的代码运行后，却看不到红色的效果。那是因为有「某些」视觉元件被底下的操作系统（Windows 和 macOS）限制住了，不允许做某些改变。试图更改按钮的底色是不会有效果的，但也不会报错。

想看到更改颜色的实际效果，我们把样式由按钮改成文字（text），代码如下：

```
view [ text "OK" 255.0.0 ]
```

如果样式后面出现两次颜色，第一个颜色就是底色（背景色），第二个颜色就是文字的颜色（前景色）。下面的例子中，底色是白色（255.255.255），文字是红色（255.0.0）：

```
view [ text "OK" 255.255.255 255.0.0 ]
```

用这样的方式，也可以指定按钮的文字颜色，只是底色依然没有作用。如下所示：

```
view [ button "OK" 255.255.255 255.0.0 ]
```

既然只想（也只能）指定文字的颜色，与其第一个颜色随便写，不如通过关键字 font-color 清楚地说明我们要指定字体的颜色（也就是文字的颜色），如下所示：

```
view [ button "OK" font-color 255.0.0 ]
```

其实除了数组（例如255.0.0）可以用来表示颜色，VID 也允许我们使用「期号」（issue!）来表示颜色。#FF0000 代表红色（十六进位的 FF 是十进位的 255），所以下面的程序和上面的程序效果完全一样。

```
view [ button "OK" font-color #FF0000 ]
```

Red 语言已经预先定义了许多颜色，包括 Red（红）、Green（绿）、Blue（蓝）、Purple（紫）。这些单字都已经事先绑定好他们相应的颜色值（tuple! 数据类型）。在 VID 中出现这些单字，会被替换成相应的值，例如下面代码中的 red 在运行时会被自动替换成 255.0.0：

```
view [ button "OK" font-color red ]
```

想知道有哪些数组（tuple!）已经被事先定义好，可以通过下面的方式：

```
>> ? tuple!
    Red               236.93.87
    transparent       0.0.0.255
    black             0.0.0
    aqua              40.100.130
    beige             255.228.196
    blue              3.101.192
    brick             178.34.34
    brown             139.69.19
    coal              64.64.64
    coffee            76.26.0
    crimson           220.20.60
    cyan              81.167.249
```

```
    forest              0.48.0
    gold                255.205.40
    gray                128.128.128
    green               112.191.65
    ivory               255.255.240
    khaki               179.179.126
    leaf                0.128.0
    linen               250.240.230
    magenta             255.0.255
    maroon              128.0.0
    mint                100.136.116
    navy                0.0.128
    oldrab              72.72.16
    olive               128.128.0
    orange              255.150.27
    papaya              255.80.37
    pewter              170.170.170
    pink                255.164.200
    purple              179.106.226
    reblue              38.58.108
    rebolor             142.128.110
    sienna              160.82.45
    silver              192.192.192
    sky                 164.200.255
    snow                240.240.240
    tanned              222.184.135
    teal                0.128.128
    violet              72.0.90
    water               80.108.142
    wheat               245.222.129
    white               255.255.255
    yello               255.240.120
    yellow              245.211.40
    glass               0.0.0.255
    color-for-primary-element  230.230.230
    color-for-non-primary-element  245.245.245
```

前面提到过，样式后面的特性可以用任何次序出现，对于需要前置关键字的特性来说，也是如此。只不过关键字和特性之间必须前后紧连著，中间不可以有其他值，所以下面是正确的次序调动：

```
view [ button font-color red "OK" ]
```

而下面就是错误的次序调动，因为 "OK" 出现在 font-color 和 red 之间：

```
view [ button font-color "OK" red ]
```

前面提到 red 是一个预先定义好的单字，它的值是 255.0.0。你可以在交互环境中输入 red，然后按下 enter 看看：

```
>> red
== 255.0.0
```

你也可以改变 red 的值，例如把 red 重新定义为 168.0.0，红色的成分从原本的 255 变成 168，少了许多，就变成暗红色了。运行下面的代码，看看画面上的文字颜色是否也真的改变了。

```
red: 168.0.0
view [ button font-color red "OK" ]
```

但这么做不太好，因为我们把 red 这个常用的颜色定义改变了。比较好的做法是，我们应该为新的颜色定义一个新的单字。例如，我把 168.0.0 定义为 dark-red（暗红色），如下所示：

```
dark-red: 168.0.0
view [ button font-color dark-red "OK" ]
```

视窗的颜色、大小、标题

到目前为止，我们都是让 Red 语言帮我们决定视窗大小。我们可以使用 size 关键字来描述我们所希望的视窗大小，如下所示：

```
view [ size 300x400 button "OK"  ]
```

如果想要指定视窗标题的文字，使用 title 关键字。如果想要指定视窗底色，使用 backdrop。如下所示：

```
view [ size 300x400 title "我的程序" backdrop 255.0.0 button "OK" ]
```

size、title、backdrop 这三者的次序可以彼此调动，但一定要放在 VID 的最前面，不可以出现在样式名称后面，例如下面的例子，是正确的：

```
view [ title "我的程序" backdrop 255.0.0 size 300x400 button "OK" ]
```

而下面的例子，title 出现在 button 后面，所以程序出错。

```
view [ backdrop 255.0.0 size 300x400 button "OK" title "我的程序" ]
```

视觉元件摆放位置

我想要在画面上放三个视觉元件，一个是用来输入文字的字段（field），一个是按钮（button），一个是静态文字（text）。我们指定字段的宽度为 100，按钮上的文字为 "Copy", 静态文字宽度为 100，且上头的文字为 "--"

```
view [
    field 100
    button "复制"
    text "--" green 100
]
```

上面的程序执行时，视觉元件由左至右排成一行，感觉不太美观。我们可以通过 return 关键字，来使其换行，代码如下所示：

```
view [
    field 100 return
    button "复制" return
    text "--" green 100
]
```

return 关键字会把当前的行（或列）结束，换到下一行（或列）的开头。这就跟打字时，按下 return 键的效果一样。

下面的代码加上 across，意思是「采用横排」（由左到右排列）。其实上面的代码，跟下面的代码效果完全一样，因为 VID 默认就是横排的。所以「across」可以省略不写。

```
view [
    across
    field 100 return
    button "复制" return
    text "--" green 100
]
```

我们也可以使用 below 关键字，把排列方式改成直排，这么一来，我们就可以不需要这两个 return。如下所示：

```
view [
    below
    field 100
```

```
    button "复制"
    text "--" green 100
]
```

below 关键字后面可以选择性地加上对齐的描述。right 表示「靠右」对齐。还可以选择靠左（left），和靠中（center）。不指定的话，默认靠左对齐。下面的例子靠右对齐：

```
view [
    below right
    field 100
    button "复制"
    text "--" green 100
]
```

动态修改视觉元件属性

我们可以为视觉元件取名字，方面以后参考到。为视觉元件取名字的方式，就是在该视觉元件的前面，放上一个「设字」（set-word!）。所谓的 set-word! 其实就是单字后面紧接著一个英文冒号。例如下面代码中的「value:」、「output:」。特别注意的是：设字后面必须要有空格（或制表符、或换行），否则会出错（前后连接会被误认为是网址）。

```
view [
    below right
    value: field 100
    button "复制" [ output/text: value/text ]
    output: text "--" green 100
]
```

在上面的例子中，随便在字段（field）上输入一些文字，当按钮按下时，会把字段中的文字，复制到静态文字（text）上头。所以我们需要为「字段」和「静态文字」分别取名，以便在按钮的代码区块中使用。

按钮内的代码值得好好介绍。这部分是 Red 语言，而不是 VID。这部分只有两个值，分别是「output/text:」和「value/text」，第二个值的类型是路径（path!），第一个值的类型是设径（set-path!）。执行 Red 程序时遇到设径，就会计算后面的值，把这个值设定到「该设径所对应的」路径上。以这里的例子来说，就是把「value/text」计算后的值设置给「output/text」。value/text 计算后的值是该字段上面的文字（用户输入的），类型是字符串（string!）。把这个字符串设置到 output/text，就会使得 output 显示出一样的字符串。

对于这里出现的两个路径「output/text」「value/text」，我需要花一些时间更详细地解释清楚。

VID 中不管出现什么样式名称，内部都会被转换成视觉元件（face!）。视觉元件其实就是对象（object!）。我们以前学习过对象内可以包含许多单字，每个单字都有自己的绑定值。至于视觉元件（face!）内所包含的单字，称为 facet，我姑且将其翻译为属性。

任何视觉元件都会固定包含 25 个属性，每个属性都有各自的值（绑定值）。只要修改视觉元件的这些属性值，就会导致画面立刻跟著改变。

下面列出其中比较重要的 12 个属性并简单说明其用途：

- type：视觉元件的样式名称，例如 button、text、field。
- offset：左上角位置。例如：0x0、50x100。
- size：体积。例如：300x400。
- text：文字。例如："OK"、"确定"。
- color：底色。例如：0.0.0、255.0.0。
- data：元件特定的数据。关于这个单字，每个元件有自己的定义。
- enabled?：是否启用？必须是 logic! 值。
- selected：选取。某些视觉元件会用到这个单字，纪录现在被选取者。
- rate：频率。描述此视觉元件被更新的频率。
- font：字体。描述字体的名称、颜色、大小。
- actors：动作。使用 VID 时，选项设置若为区块，则会被放到这里。
- draw：绘图。这里描述画面上要绘制的图。

其中，text 是视觉元件（face!）内属性的名称，也是 type 属性内的一种样式名称，不要把两者混淆了。当我提到 text 的时候，要分清楚说的是何者。

我们可以通过路径的方式取得这些属性的值或设置它们，这正是「output/text: value/text」做的事：从名为 value 的视觉元件中取出 text 属性的值，将其设置成为 output 对象的 text 属性值。

加法计算器

在下面的例子中，我们设计了一个加法计算器。

```
view [
    below right
    value1: field 100 hint "被加数"
    value2: field 100 hint "加数"
    button "加法" [ output/data: value1/data + value2/data ]
    output: text "--" 100
]
```

这里我们使用的 field 样式的一个选项：hint，它用来描述提示用的字符串。这个字符串会在字段内容为空的时候出现。

这个程序的用法是当按下「加法」按钮时，会把上面两个字段的数字相加，结果显示在下面的 output 中。Red 程序写法是「output/data: value1/data + value2/data」，而非「output/text: value1/text + value2/text」，这是因为 value1/text 和 value2/text 的绑定值都是字符串（string!），而两个字符串是不能相加的。

field 和 text 这两种视觉元件有一个特色，他们的 text 属性和 data 属性之间会自动数据双向同步，且 text 属性值一定是字符串，但 data 会试图把 text 属性值（字符串）「转换成」其他数据类型的字面值（至于如何转换，下面补充解释）。例如：text 的内容是 "1"（字符串）时，data 的内容就是 1（整数）；data的内容是 255.0.0（数组）时，text 会是 "255.0.0"。

> 补充
>
> 把字符串转成对应的字面值，用的是 load 函数。例如：
>
> ```
> >> load "1"
> == 1
> >> load "http://www.abc.com"
> == http://www.abc.com
> >> load "abc:"
> == abc:
> ```

```
>> load "1 2 3"
== [1 2 3]
```

只要用户在 value1 和 value2 中输入数字（例如 1 和 2）或任何可以相加的字面值（例如2017-07-24 和 2，或者 12:00:00 和 5），然后按下「加法」按钮，就可以计算两个相加的结果，然后设置到 output 的 data 中，并被转成字符串，同步到 text 中，显示出来。

但万一用户在 value1 和 value2 中输入的是「转换后无法被采用加法计算」的值（例如 jerrytsai@abc.com），就会导致错误。你可以在 value1 和 value2 中分别输入「abc」和「123」，你会看到控制台中显示出错误信息。

一个好的程序应该要考虑到用户操作错误的情况，并好好地处理这种情况。出现错误信息给用户看，这样的体验不好，应该尽量避免。于是我们在下面的程序中，把计算加法的部分用 attempt 函数包围起来。这个函数会计算后面的区块内容，并将结果传出来，如果发生错误则传出一个表示「空」的特殊值。当「空」的特殊值被设置到 output/data 中，output/text 就会变成空字串 ""，导致其画面被清空。

```
view [
    below right
    value1: field 100 hint "被加数"
    value2: field 100 hint "加数"
    button "Add" [ output/data: attempt [ value1/data + value2/data ] ]
    output: text "--" 100
]
```

四则计算器

我希望这个程序可以继续改良，用来做更多种类的计算，包括加减乘除。下面是改良后的结果：

```
view [
    below right
    value1: field "0" 100
    value2: field "0" 100
    operation: drop-list data [ "加法" "减法" "乘法" "除法" ]  select 1
    button "计算" [
        switch operation/selected [
            1 [ output/data: attempt [value1/data + value2/data] ]
            2 [ output/data: attempt [value1/data - value2/data] ]
            3 [ output/data: attempt [value1/data * value2/data] ]
            4 [ output/data: attempt [value1/data / value2/data] ]
        ]
    ]
    output: text "0" 100
]
```

我通过 drop-list 样式来放置四种计算方式，供用户选择。设置 drop-list 选项的方式是通过 data 关键字。我这里还用了 select 关键字来表示默认所选取的是 data 中的第一个选项（也就是"加法"）。

现在把按钮上的文字由「加法」改成「计算」，要判断到底是哪一种计算，才做对应的计算。

switch 需要两个参数，算出的第一个结果值（本例中就是 operation/selected 计算出来的值），会拿来用在第二个参数（是个区块）内进行比对，比中了就会找出后面出现的第一个区块，运行该区块内的程序，并将结果当成 switch 的返回值。当下拉清单目前选取的是减法（第二个），这时候 operation/selected 的计算结果就会是 2，所以 switch 会计算 2 之后的区块内部 [output/data: attempt [value1/data - value2/data]]。

上面的程序还有很大的优化空间，因为 switch 里面有重复的地方，可以简化。像这种把程序改写成更好，但不影响其运行效果的行为，我们称为重构（refactoring）。简化后的版本如下：

```
view [
```

```
    below right
    value1: field "0" 100
    value2: field "0" 100
    operation: drop-list data [ "加法" "减法" "乘法" "除法" ] select 1
    button "计算" [
        output/data: attempt [
            switch operation/selected [
                1 [ value1/data + value2/data ]
                2 [ value1/data - value2/data ]
                3 [ value1/data * value2/data ]
                4 [ value1/data / value2/data ]
            ]
        ]
    ]
    output: text "0" 100 gray
]
```

简单来说，就是把一再重复的部分提取出来。

如果你这个计算器尝试一下除法计算，你可能很快就会发现问题：整数除以整数的结果还是整数，馀下的部分都被省略了。例如：7 除以 2 会得到 3，而不是 3.5。用 Red 交互式控制台试试看，也是一样的效果：

```
>> 7 / 2
== 3
```

这是因为 Red 语言有一个规则：整数之间的计算还是得到整数。如果你想得到小数（浮点数），你必须用小数的计算，写成这样：

```
>> 7.0 / 2
== 3.5
```

响应式四则计算器

我希望继续改进这个程序，只有在 value1 和 value2 都是数字时，「计算」按钮才启用（enabled? 属性设置为 true），否则就禁用（enabled? 属性设置为 false）。被禁用的按钮会变成灰色，不能被点按。

```
view [
    below right
    value1: field "0" 100
    value2: field "0" 100
    operation: drop-list data [ "加法" "减法" "乘法" "除法" ] select 1
    button "计算" [
        output/data: attempt [
            switch operation/selected [
                1 [ value1/data + value2/data ]
                2 [ value1/data - value2/data ]
                3 [ value1/data * value2/data ]
                4 [ value1/data / value2/data ]
            ]
        ]
    ]
    react [
        face/enabled?: (number? value1/data) and (number? value2/data)
    ]
    output: text "0" 100 gray
]
```

上面的例子中，我在「计算」按钮的后面多了一个 react 选项设置，表示这个按钮有某些特性会随时受到外面的影响而改变。react 选项后面需要一个区块，区块里面是 Red 语言，描述必须保持的条件。在这个例子中，face/enabled? 随时要和后面的值一致。由于 react 区块是这个「计算」按钮的选项，这里的 face 指的是这个按钮本身。

「face/enabled?:」后面的值是 and 运算子（op!）的计算结果。对于 op! 类型的函数，一定具有两个参数，且函数一定放在两个参数的中间。

第一个参数是 (number? value1/data)，判断是否 value1/data 的值为数字（包括 integer!、float!、percent!），如果是则传出「真」值，否则传出「假」值。and 运算子只有在两个参数都为真时，才为真。只有当 value1/data 和 value2/data 都是数字时，face/enabled? 才会被设定为「真」。任何一者不是数字时，face/enabled? 会被设定为「假」。因为放在 react 后的区块中，所以这个设置会自动进行，自动维护。

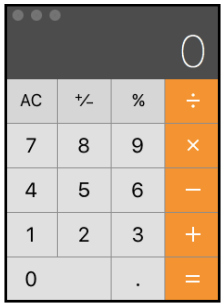
补充

所谓的响应式，就是我们把某些数据之间的关系描述清楚，只要其中的数据有变动，相应的数据就会自动跟著调整，不用我们操心。以上面的例子来说，只要 value1/data 或 value2/data 的数据有变动，face/enabled? 的值就会跟著变动。

通过上面的程序，你应该体会到 react 选项的方便。下面的例子依然使用 react，但更进一步。我希望移除「计算」按钮，让计算自动进行。也就是说，只要 value1/data 或 value2/data 或 operation/selected 的值一有变动，就立刻重新计算，并将结果展现在 output。

```
view [
    below right
    value1: field "0" 100
    value2: field "0" 100
    operation: drop-list data [ "加法" "减法" "乘法" "除法" ] select 1
    output: text "0" 100 gray
    react [
        output/data: attempt [
            switch operation/selected [
                1 [ value1/data + value2/data ]
                2 [ value1/data - value2/data ]
                3 [ value1/data * value2/data ]
                4 [ value1/data / value2/data ]
            ]
        ]
    ]
]
```

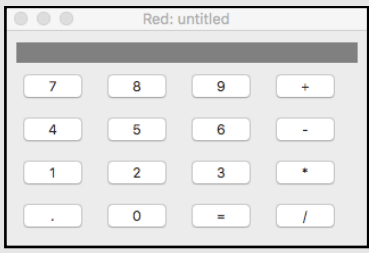
拟真计算器画面安排



上面的计算器毕竟不太像真实的计算器，而且只能做一次简单的计算。接下来我希望做出真正的计算器，像苹果 macOS 操作系统自带的计算器那样，你可以通过 Finder => 应用程序 => 计算器.app ，找到这个程序。执行起来如右图所示：

思考一下如何设计这个程序，我决定先从简单的事情动手：先来做画面安排：

```
view [
    text gray 300
    return
    button "7"
    button "8"
    button "9"
    button "+"
    return
    button "4"
    button "5"
    button "6"
    button "-"
    return
    button "1"
    button "2"
    button "3"
    button "*"
    return
    button "."
    button "0"
    button "="
    button "/"
]
```

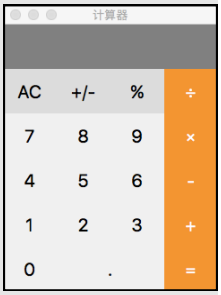


在上面的例子中，我们通过 return 来表示换到下一行。我们轻松地安排出计算器的画面，虽然外观差异很多。

button 受到操作系统的控制，比较不好做出大改变（例如，按钮颜色无法改变）。我决定不用 button，改用 base。

base 是一种没有任何内定行为和风格的视觉元件，适合用来设计我们自己个性化的视觉元件。接下来我们做其颜色的调整，好更接近苹果的计算器外观：

```
lite-gray: 220.220.220
white-gray: 240.240.240
btn-size: 60x50
view [
    title "计算器"
    origin 0x0
    space 0x0
    text gray 240x50
    return
    base "AC" lite-gray btn-size font-size 16
    base "+/-" lite-gray btn-size font-size 16
    base "%" lite-gray btn-size font-size 16
    base "÷" orange btn-size font-size 16 font-color white
    return
    base "7" white-gray btn-size font-size 16
    base "8" white-gray btn-size font-size 16
    base "9" white-gray btn-size font-size 16
    base "×" orange btn-size font-size 16 font-color white
    return
    base "4" white-gray btn-size font-size 16
    base "5" white-gray btn-size font-size 16
    base "6" white-gray btn-size font-size 16
    base "-" orange btn-size font-size 16 font-color white
    return
    base "1" white-gray btn-size font-size 16
    base "2" white-gray btn-size font-size 16
    base "3" white-gray btn-size font-size 16
    base "+" orange btn-size font-size 16 font-color white
    return
    base "0" white-gray btn-size font-size 16
    base "." white-gray 120x50 font-size 16
    base "=" orange btn-size font-size 16 font-color white
]
```



上面的例子中，VID 一开始还使用两个关键字 origin 和 space。

origin 用来指定视觉元件的开始座标位置，这个位置必须是数对（pair!）。座标系统是这样的：左上角是 0x0，往右 X 座标增加，往下 Y 座标增加。origin 0x0 的意思是：左上角不要留任何空白，直接把视觉元件从最左上角的位置开始摆放。如果你不这么做的话，默认的 origin 值是 10x10。

space 用来指定视觉元件两两之间的缝隙大小，这个值必须是数对（pair!）。这个值的 X 部分表示元件之间的水平空隙、Y 部分表示元件之间的垂直空隙。我不希望有任何空隙，所以使用 space 0x0。如果你不这么做的话，默认的 space 值也是 10x10。

这个例子还用到 font-size （字体大小）和 font-color （字体颜色）这两个关键子。默认字体实在太小，所以我们将字体设置为 16。

画面安排已经差不多了，但代码中有大量的重复，有改进的空间。我们可以定义新的样式（style），然后反复使用这个新样式，来减少代码的重复。定义新样式的方法很简单，使用 style 关键字，其后跟著一个「新样式的名称」所对应的设字，后面再跟著沿袭的既有样式名称，再后面就是各种选项的设置。如下面的例子所示：

```
lite-gray: 220.220.220
white-gray: 240.240.240
view [
    title "计算器"
    style btn: base white-gray 60x50 font-size 16 draw [pen gray box 0x0 59x49]
    origin 0x0
    space 0x0
    text gray 240x50
    return
    btn "AC" lite-gray
    btn "+/-" lite-gray
    btn "%" lite-gray
    btn "÷" orange font-color white
    return
    btn "7"
    btn "8"
    btn "9"
    btn "×" orange font-color white
    return
    btn "4"
    btn "5"
    btn "6"
    btn "-" orange font-color white
    return
    btn "1"
    btn "2"
    btn "3"
    btn "+" orange font-color white
    return
    btn "0"
    btn "." 120x50 draw [ pen gray box 0x0 119x49 ]
    btn "=" orange font-color white
]
```

以上面的例子来说，我们定义了一个名为 btn 的新样式，它采用 base 当基本样式，white-gray 60x50 font-size 16 draw [pen gray box 0x0 59x49] 是其选

项设置。定义了这个样式之后，以后任何需要样式名称的地方，只要写 btn，就相当于写 base white-gray 60x50 font-size 16 draw [pen gray box 0x0 59x49]，简洁了许多。

我们希望在每个按钮的周围都绘制方框，所以我们在定义 btn 的时候，使用了 draw 选项。draw 关键字的后面需要跟著一个区块，区块内是另外一个语言，就叫做 draw 方言，用来描述画图指令。「pen gray」的意思是「以后画图都采用灰色的笔」，「box 0x0 59x49」表示画一个方框（box），此方框的左上角座标是 0x0，右下角座标是 59x49。关于绘图，目前只说明到这里，本章最后一节会有更详细的例子。

> 补充
>
> 如果视觉元件的大小是 60x50，那么它的左上角位置是 0x0，右下角是 59x49。想刚好在边界画一个框，就要在 draw 方言中写 box 0x0 59x49，而不是 box 1x1 60x50。

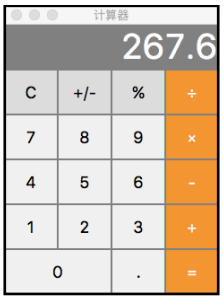
拟真计算器功能

到现在为止，我们的按键式计算器都只是在画面设计上做文章，还没有真正的计算功能。在下面的代码中，我们开始为它加上真正的计算功能。

这一节要采用不同的解说方式，我把解说写成注释（comment），穿插在代码中。注释是给人看的，并不会执行。Red 语言的注释方式是：从分号开始到该行的结束，都属于注释。

如果不在代码中写注释，拿到代码的人可能不容易理解这份代码。事实上，就连当初写程序的人，隔一段时间之后，也非常有可能无法快速理解自己的代码。所以适当地写注释是一个好习惯。

除了代码中穿插注释之外，我们还有一个好习惯应该要养成，那就是描述每个脚本文件的名称、目的、版本、作者等信息。脚本文件要求我们必须以 Red [ ] 开头，而这个区块就是让我们写这些信息的，只是之前我们一直偷懒没写。



下面是全功能的计算器，注释的部分用绿色标示出来。执行的结果如右边的图所示：

```
Red [
    Title:   "计算器"
    Author:  "Jerry Tsai"
    Date:    2017-07-20
    Version: 1.0.0
    purpose: "模仿苹果电脑和手机的计算器"
]

lite-gray: 220.220.220
white-gray: 240.240.240

; 以下四个单字用来在内存中记录「当前状态」：
is-point?: false    ; 是否已经按下小数点
value1: 0           ; 第一个值
op: "="             ; 上一次按下的计算键。一开始必须设置为"="
value2: 0           ; 第二个值，也就是目前正在输入的值

view [
    title "计算器"    ; 视窗标题
    ; 从按键的外观和功能来分析，我把按键分为三类，分别是：
    ; * 数字键：十个数字和小数点
```

```
; * 计算键：加、减、乘、除、以及等号
; * 功能键：「C/AC」、「+/-」、「%」
; 为此三类我们定义了三个样式，分别是 num-btn、op-btn、func-btn。
; 这三个样式都可以沿袭自 btn 样式，以减少这三个样式的定义长度，让代码更短。
; 在真正的布局中我们没有使用到 btn，只有使用到 num-btn、func-btn、op-btn。
style btn: base 60x50 font-size 14 draw [pen gray box 0x0 59x49]

; num-btn是数字键，沿袭自 btn，改变了颜色，并定义了行为
style num-btn: btn white-gray [
    ; 「C/AC」按键上面的文字会在 "AC" 和 "C" 之间切换：
    ; * AC 用来清除「当前状态」。
    ; * C 用来清除当前画面，使得画面显示0。
    ; 当数字键被按下之后，画面上有数字了，「C/AC」键就会显示「C」
    c-btn/text: "C"
    ; 小数点是否已经被按过
    either not is-point? [
        ; 如果尚未按下过小数点，则把目前的值（也就是value2）乘以10，再加上
        ;「刚按下的数字」。这会有什么效果呢？举例来说。现在的值是0，按下1，
        ; 就会显示 0 * 10 + 1 的结果，也就是1。再按下2，就会显示
        ; 1 * 10 + 2 的结果，也就是 12。这么简单的公式，就达到我们要的效果。
        ;
        ; 「刚按下的数字」为什么是 load face/text ？请复习「加法计算器」一节的说明。
        display/data: value2: value2 * 10 + load face/text
    ] [
        ; 如果已经按下过小数点，则前面的公式 value2 * 10 + load face/text
        ; 就不管用了，我们改用的方法是直接把画面上显示的字符串「后面」补上新按下
        ; 的数字，例如：现在是 "3."，我们按下 1，则显示 “3.1"，再按下2，则显示
        ; "3.2"。
        ;
        ; 关于 append，请复习第二章最后一节的说明。
        append display/text face/text
        ; value2此时必须跟著画面一起更新
        value2: display/data
    ]
]

; func-btn 是功能键，沿袭自 btn，改变了颜色，但不定义共同的行为，
; 因为每个功能键的行为差异太大。
style func-btn: btn lite-gray

; op-btn 是计算键，沿袭自 btn，改变了颜色和字体颜色，且定义了行为。
style op-btn: btn orange font-color white [
    ; 「计算键」被按下之后，开始计算。请注意：算的不是现在按下的计算键，而是
    ; 上次按下的计算键。上次按下的计算键被纪录在 op 中。
```

```
        value2: display/data
        value1: switch op [
            "÷" [ value1 * 1.0 / value2 ]
            "×" [ value1 * value2 ]
            "-" [ value1 - value2 ]
            "+" [ value1 + value2 ]
            "=" [ value2 ]
        ]
        ; 计算完了，把 value2 清空，并把 is-point? 恢复原状（false），
        ; 再把现在按下的计算符号记下来，以便下次计算。
        value2: 0
        is-point?: false
        op: face/text
        ; 计算完了把 value2 清空，并把 is-point? 恢复原状（false）
        display/data: value1
    ]

    origin 0x0
    space 0x0

    ; 在 VID 中，data 是一个关键字，后面跟著一个值，表示把此视觉元件
    ; 的 data 设置为这个值。
    display: text gray 240x50 font-color white font-size 30 right data 0
    return

    c-btn: func-btn "AC" [
        either "AC" = face/text [
            ; 当上面的字样是"AC"时，表示要清除内存，把 is-point?、value1、op、value2
            ; 这四个值都恢复到原状。但因为出现 "AC" 表示已经出现过 "C"，而处理 "C" 的时候
            ; 会把 value2 和 is-point? 复原，所以这里只要复原 value1 和 op 即可。
            value1: 0
            op: "="
        ] [
            ; 当上面的字样不是"AC"时，就会是"C"，这表示要清除画面。
            ; 同时value2和is-point?也会被清除。
            display/data: 0
            value2: 0
            is-point?: false
            ; 按下"C"之后，上面的字样会变成"AC"
            face/text: "AC"
        ]
    ]
    func-btn "+/-" [
        either #"-" = display/text/1 [
            ; 如果第一个字符是减号，就去除减号
            remove display/text
```

```
        ] [
            ; 如果第一个字符不是减号，就插入减号
            insert display/text #"-"
        ]
    ]
    func-btn "%" [
        display/data: display/data / 100.0
    ]
    op-btn "÷"
    return

    num-btn "7" num-btn "8" num-btn "9" op-btn "×"
    return

    num-btn "4" num-btn "5" num-btn "6" op-btn "-"
    return

    num-btn "1" num-btn "2" num-btn "3" op-btn "+"
    return

    num-btn "0" 120x50 draw [ pen gray box 0x0 119x49 ]
    num-btn "." [
        if not is-point? [
            append display/text "."
            is-point?: true
        ]
    ]
    op-btn "="
]
```

如果你能看懂这个程序的每个细节，对初学者来说，已经相当厉害了。

小时钟

写个小时钟的程序，其实很容易，重点在于：

* 取得现在的时间，显示出来

* 每秒重复上面的动作一次

在 VID 中，通过 rate 关键字就可以为视觉元件设置定时器（timer），让此定时器以固定的频率发送通知给该视觉元件。视觉元件收到来自计时器的通知时，就会执行 on-time 函数。在 VID 中，我们可以通过 on-time 关键字来定义这个函数的行为。

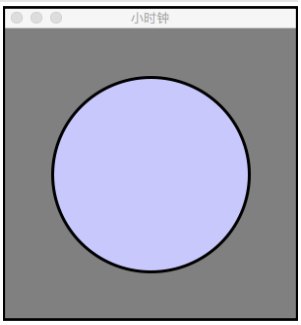
```
view [
    title "小时钟"
    t: text 250x50 font-size 24 rate 1
        on-time [ t/data: now/time ]
]
```

上面虽然是一个有实际功用的小时钟，但我更希望是有时针、分针、秒针的圆形时钟。接下来我们朝这个方向努力。

我们依然是先从最简单的部分下手：绘制圆形的部分。我们使用 draw 关键字来描述我们想要绘制的图形。draw 后面跟著一个区块，这个区块内使用的语言是 draw 方言（不同于 Red 语言和 VID 方言）。

```
view [
    title "小时钟"
    origin 0x0
    base 300x300 draw [
        pen black               ; 设置画笔的颜色是黑色，画笔用来绘制线条
        line-width 3            ; 设置线条的宽度是三个像素
        fill-pen 200.200.255    ; 设置填充笔的颜色是 200.200.255
        circle 150x150 100      ; 绘制一个圆，绘制这个圆的时候，采用上面的设置
    ]
]
```

简单来说，draw 方言就是 先做环境的设置，然后进行绘图，而绘图时会采用之前的设置。以上面这个例子来看，圆形的圆心在 150x150，半径是 100，而圆周的绘制会采用黑色，圆周线条的宽度会是 3，圆的内部会采用 200.200.255 的颜色填充。



特别注意，绘图的座标系统和数学的座标系统不一样。绘图的座标系统是：

* 左上角是座标原点 0x0

* 往右是 X 轴正向

* 往下是 Y 轴正向（这和数学座标不同）

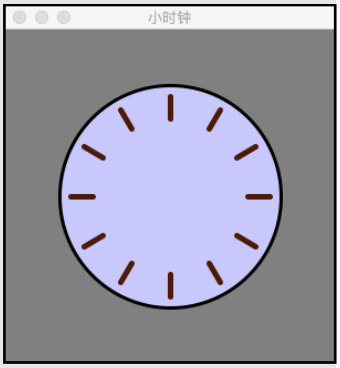
绘制完圆形，接下来绘制十二个刻度（短线条）。我希望刻度颜色是咖啡色，宽度是五个像素，我还希望线条的两个端点都是圆形（而不是横切断面）。于是我在绘制圆形之后，做了这样的设置「pen coffee line-width 5 line-cap round」。

设置完后应该要开始绘制十二条线段了。画线的方式是用 line 关键字，后面跟著两个端点（都是数对），例如 line 0x0 100x100 就表示在 0x0 到 100x100 之间画一条线。

这十二条线共 24 个点的座标，必须利用三角函数 sine 和 cosine 计算出来，但 draw 方言内没有计算能力。我们只好藉助 Red 语言。在 VID 内，通过 do 关键字，可以随时跳回 Red 语言。我们利用这个方法写了一段 Red 语言的代码，在刚才的 draw 区块最后面补上这十二个画线的描述。

请复习第二章「动态修改视觉元件属性」这一节的内容。所有视觉元件内都有一个 draw 属性，在 VID 内使用 draw 关键字做的描述都会被纪录到这个属性中。所以我们在下面的程序中，通过 clock-face/draw 就可以取得这个属性，是个区块。我们在该区块的尾部填入（append）这十二个画线的描述。 我们通过 while 函数，让 append 被执行 12 次，每次都会写入一组画线的命令。

```
view [
    title "小时钟"
    origin 0x0
    clock-face: base 300x300 draw [
        pen black
        line-width 3
        fill-pen 200.200.255
        circle 150x150 100
        pen coffee
        line-width 5
        line-cap round
    ]
    do [
```



```
        ; 通过 do，可以回到功能强大的 Red 语言。这个区块内是 Red 语言。
        angle: 0    ; 当前刻度的角度
        while [ angle < 360 ] [
            append clock-face/draw reduce [
                'line  ; 'line是原字(lit-word!)，reduce之后会得到单字line
                ; 下面是第一个点，分别通过三角函数计算出x和y，
                ; 然后用as-pair函数把x和y组成一个数对（pair!）
                as-pair 150 + (90 * sine angle) 150 + (90 * cosine angle)
                ; 下面这是第二个点
                as-pair 150 + (70 * sine angle) 150 + (70 * cosine angle)
            ]
            angle: angle + 30   ; 下次刻度的角度必须差30度(这是360除以12的结果)，
                                ; 因为有12个刻度，而一个圆周是360度，
        ]
    ]
]
```

上面的例子中，我们使用三角函数 sine 和 cosine 来计算出刻度的位置。下面的例子要介绍另一种作法：采用旋转的方式。

我们可以不改变画线的命令，只是每次画线之后，都把底下的图以圆形中心为中点旋转 30 度，12 次之后也可以得到相同的效果。

使用旋转命令的时候，旋转的中心点是座标上的原点，也就是 0x0。以这个例子来说，我们必须改变原点，把座标系统「平移」 150x150，这个时候。原本的 150x150 座标变成了新的0x0 座标（原点）。

平移使用的关键字是 translate，旋转使用的关键字是 rotate。不管平移或旋转，效果都是累积的。所以先平移，再旋转，这个旋转就会受到前面平移的影响。如果再次旋转，这次旋转还会受到前面平移与旋转的影响，效果持续累积。

在这个例子中，我们每次旋转只转 30 度，但其实这个角度整体来说是累积的。也就是说，第一次距离 0 度角 30 度，第二次 距离 0 度角 60 度，第三次距离 0 度角 90 度 。

```
view [
    title "小时钟"
    origin 0x0
    clock-face: base 300x300 draw [
        pen black
        line-width 3
        fill-pen 200.200.255
```

```
        translate 150x150 ; 座标系统平移 150x150，在这之后 150x150变成新的原点
        circle 0x0 100    ; 所以我们现在的圆心不再是 150x150，而是 0x0
        pen coffee
        line-width 5
        line-cap round
    ]
    do [
        counter: 0
        while [ counter < 12 ] [
            ; 不用再计算三角函数，现在我们用一样的命令绘图，每次画完就旋转30度
            append clock-face/draw [ line 0x90 0x80 rotate 30 ]
            counter: counter + 1
        ]
    ]
]
```

接下来终于到了绘制时针、分针、秒针的时候了。

> 重要
>
> 在 VID 中，当视觉元件的领土有重叠的时候，越晚出现在代码中的视觉元件会出现在越上面，而越早出现在代码中的视觉元件会被压在下面。

我不希望把时针、分针、秒针的绘制指令和刚才的绘制刻度的绘图命令放在同一个视觉元件的 draw 属性中，因为刻度是固定不变的，而指针是会动的。我决定另外准备一个底色透明的视觉元件，交叠在刻度的视觉元件之上，指针就在这个上层的视觉元件内绘制。

通过 at 关键字可以明确指定后面的一个视觉元件要摆放的位置（该元件左上角放置的座标位置）。现在我们有两个视觉元件，第一个绘制刻度的视觉元件，被自动放在 0x0 的位置（因为 origin 设置为 0x0），而第二个用来绘制指针的视觉元件，通过「at 0x0」的描述，也会被放在 0x0 的位置。不只相同位置，两者体积也一样（都是 300x300），所以不只交叠，而是完全重叠。这个时候，第二个视觉元件会放在第一个视觉元件的上头。

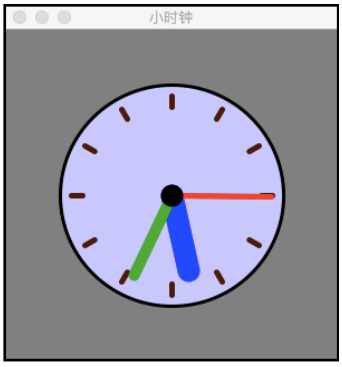
我们选用的第二个视觉元件样式是 box，这其实就是 base，差别在于 box 的底色已经预先设置为透明了。所以尽管第二个视觉元件叠在第一个之上，但仍可以看到第一个视觉元件。这正是我们要的效果。

```
view [
    title "小时钟"
    origin 0x0
    ; 第一个视觉元件
```

```
clock-face: base 300x300 draw [
    pen black
    line-width 3
    fill-pen 200.200.255
    translate 150x150
    circle 0x0 100
    pen coffee
    line-width 5
    line-cap round
]
do [
    counter: 0
    while [ counter < 12 ] [
        append clock-face/draw [ line 0x90 0x80 rotate 30 ]
        counter: counter + 1
    ]
]
; 第二个视觉元件，通过 at 的指示，把自己叠在第一个视觉元件之上。
; 为了达到秒针平滑移动的效果，我不再使用 rate 1，而是改用 rate 30，
; 也就是说，定时器在每一秒钟会发出30次通知。
at 0x0 hands: box 300x300 rate 30 draw [
    line-cap round
    ; 把原点平移到圆心所在的位置，计算这三根针的线段端点时会简单一些。
    translate 150x150

    ; 绘制时针
    pen blue
    line-width 20
    hour: line 0x0 0x0    ; 两个端点先暂时写成0x0，反正 on-time 一来马上更新
    ; 上面的 hour: 没有实际的作用，只是用来纪录区块内的一个位置，供on-time参考使用

    ; 绘制分针
    pen 50.168.50
    line-width 10
    min: line 0x0 0x0
    ; 上面的 min: 没有实际的作用，只是用来纪录区块内的一个位置，供on-time参考使用

    ; 绘制秒针
    pen red
    line-width 5
    sec: line 0x0 0x0
    ; 上面的 sec: 没有实际的作用，只是用来纪录区块内的一个位置，供on-time参考使用

    ; 绘制中心的一个小黑点
    pen off     ; off 表示我们不需要绘制线条，所以小黑点的圆周没有颜色。
```

```
        fill-pen black
        circle 0x0 10
    ] on-time [
        ; 由于我们要做出平滑移动的指针，所以我们需要取得精确时间
        t: now/time/precise
        ; 特别注意，time!数据类型是二十四小时制，我们必须把它转成十二小时制，
        ; 因为时针每十二小时就转了一圈。
        if t/hour >= 12 [ t/hour: t/hour - 12 ]
        ; 下面三行分别取得时针、分针、秒针的角度
        sec-angle: t/second * 6 - 90
        min-angle: (t/minute + (t/second / 60.0) ) * 6 - 90
        hour-angle: (t/hour + (t/minute / 60.0)) * 30 - 90
        ; 通过个别的角度，计算出这些针的端点位置，设置到绘制线条的后的第二个参数
        ; 特别注意：我们稍早在绘制这三个针的line命令前面加上了设字，
        ; 分别是hour: min: sec:
        ; 这三个设字所对应的单字，可以用来访问这个区块的位置，如此一来，hour/1 就是 line
        ; line/2 就是画线的第一个参数，line/3就是第二个参数。下面通过 hour/3: 等方式
        ; 就可以设置新的端点。
        sec/3:  as-pair 90 * cosine sec-angle  90 * sine sec-angle
        min/3:  as-pair 80 * cosine min-angle  80 * sine min-angle
        hour/3: as-pair 70 * cosine hour-angle 70 * sine hour-angle
    ]
]
```

程序运行起来如右图所示。



后语

通过这本书，我向你展示了编程能做的事情中极小一部分，我的目的是希望读者能快速入门，产生兴趣。这只是一个开端，前方的路还很长，乐趣也还很多。

如果你还没根据前言所说的加入中文讨论区，现在就去加入吧！也希望你能到 Github 按下星号，表达你对这个语言的重视，以及你对 Red 语言开发团队的鼓励。

你可以到 https://github.com/red/red 下载完整的 Red 源码。选择绿色的「Clone or download」，再选择「Download ZIP」，将下载回来的文件解压缩。通常鼠标快速双击这个文件，就可以解压缩。解压缩之后，在其中的 tests目录下有很多测试用的程序，你也可以执行看看这些程序，并阅读其源码。

如果你能够阅读英文，你可以到 https://doc.red-lang.org/en/ 找到更多 Red 的学习参考资料。中国大陆以外的读者，也可以访问 Red 语言作者的英文博客 http://www.red-lang.org/。