



标题还没想好

阿姆斯特朗回旋加速喷气式阿姆斯特朗炮使用手册

作者：马猴烧酒

组织：万国马猴阿姆斯特朗回旋加速喷气式阿姆斯特朗炮研究所

时间：2022

版本：0.0.1



任何足够先进的科技，都和魔法难辨差异。— 阿瑟·克拉克

# Preface

I'm boring.

我不过就是一个业余的业余计算机爱好者, 不会什么很硬核的东西和技能, 想要尝试写这个教程不过是自己的无聊想法而已. 基本上是在边学边写. 主要的写作动力来自于我的朋友们和我的计算机科学导论课程.

所以在最前面我应该声明一点: 我的这本教程应该定位为一个普及性质的东西, 并且和严格的科班教材应该还是有区别的. 呃, 怎么说呢, 也不能这么说, 毕竟我还是参考了许多的书, 并且你可以说, 某些部分我就是基本照抄的, (因为写得太好了, 并且我也只是刚学而已). <sup>1</sup>

但是毕竟这个教程是边学边写, 并且我是一个很随心所欲的菜狗. 所以感觉我的风格可能就像是那种从前互联网时代的计算机爱好者, 吃饱了撑着在网络上漫游, 然后不知不觉就不小心学到了知识... 学的东西不是很成体系, 并且还很支离破碎. 可能会想到哪就写到哪. 所以我在这里先立下 (反向)flag, 大概我想要讲的内容和东西如下:

- 计算机语言因为我只稍微接触过一点点 Ruby, (ぎりぎり勉强能用的水平). 所以我将以它为主要的介绍对象. <sup>2</sup>
- 计算机玩具嗯, 为了留住我的读者, 我的朋友不是理科的, 然后她/他/它好摆烂 (bushi, 我什么都没说), 不是很喜欢玩这种不好玩的东西.
- 程序逆向和计算机构造因为我就是通过这些才学会的, 计算机... 不要不信啊.
- ...

嗯, 毕竟是草稿性质的东西, 写到哪里算哪里. 又, 我的文章不过是拾人牙慧, 山寨他人思想的普通文字罢了.

(害怕, 我们的院长说他整了好几年还是没能够写完一本原子物理, 可见写一本教材的困难, 难道我现在正面对着世界上的一个极其困难的任务?)

## 如何阅读本书

要由这本书来传达的只是一个单一的思想, 可是尽管我费劲心力, 除了用这全本的书以外, 还是不能发现什么捷径来传达这一思想.

叔本华《作为意志和表象的世界》(石冲白译)

啊哈哈, 因为这本书有些像是在和我的朋友对话的过程中写成的, 也就是说, 其中的内容是非常零碎和比较混乱的(就是一本注释可能比正文还要长的书了, 并且里面除了计算机编程, 还乱七八糟地扯了很多的别的东西). 所以对于阅读本书的读者来说, 可能会有一种"这家伙讲的东西怎么和一般的东西不太一样啊!"的想法.

我的特点就是废话特别多, 所以还请谅解, 如果你实在是看不下去的话, 请跳过我的那些废话吧(笑). 我想要构造的教程应该是这样的一个"思想的系统"(参见叔本华的《作为意志和表象的世界》第一版序): 在结构上相互关联, 在整体中涵蕴部分而又部分中体现整体. "整个思想通过各个部分而显明, 而不预先理解全部, 也不能彻底了解任何最细微的部分".

也许你会觉得这样不就会导致"为了学会 A, 就要学会 B, 但是为了学会 B, 却要学会 A"这样无厘头的矛盾循环吗? 所以为了防止这样的困境, 我打算在一开始就简单地介绍贯穿全书的概念"抽象". 然后在这之后来不断反复地应用这个概念.

## 全书的写作大纲

这个是我在写作的时候起草的一个大纲性质的东西, 不过希望也能够给你阅读的感觉:

<sup>1</sup>不过放心, 我不会像是某些编程语言的书一样, 从差劲的 hello world 开始, 然后开始无情的分类介绍, 什么字符串操作, 什么变量赋值, 什么控制流之类的. 虽然这样会比较系统, 但是对于计算机是什么以及如何看待计算机好像没什么用.

<sup>2</sup>其实选择 Ruby 语言的还有一个原因是在学习它的过程中, 我遇到了一个很厉害的家伙: \_why, 他的 Why's (poignant) Guide to Ruby 对我的影响真的很大. 呃, 算是夹带私货了吧. 并且我这本书将会尽可能地去学习 why 先生的讲故事的风格.

---

## 第一章 - 介绍语言

- 抽象是... (参考 John Locke)
- 形式地介绍一门编程语言: Ruby
- 稍微介绍形式上的含义, 用 Ruby 来理解 Ruby, 但是不是那么严格的解释
- 写代码的一些规范

## 第二章 - 分解过程

- 用 Ruby 来模拟逻辑运算: 计算的过程是子过程的组合
- 用 Ruby 来模拟一个计算器
- 用 Ruby 模拟一个简单图灵机, 模拟一个计算机
- 如何将"分解的过程" 用过程来描述

## 第三章 - 组合过程

- 如何将小过程组合成大过程
- 数据的流动

## 第四章 - 描述过程的算法

•

# 目录

<b>Preface</b>	<b>i</b>
<b>第 1 章 Language and What I mean Language</b>	<b>1</b>
1.1 Magic is Basic . . . . .	1
1.1.1 Intro: Once upon a Time . . . . .	1
1.1.2 异乡人啊, 你的管风琴在唱着什么歌谣呢? . . . . .	4
1.1.3 特殊和一般的形式 . . . . .	8
1.2 Get Your hands dirty . . . . .	15
1.2.1 过程视为数据 . . . . .	15
1.2.2 用你的名字呼唤我 . . . . .	18
1.3 理论法师也要会近战暴击 . . . . .	25
1.3.1 六芒星的内角是 60°, 不是 36° . . . . .	25
<b>第 2 章 Break Things Down</b>	<b>26</b>
2.1 . . . . .	26
<b>第 A 章 Environment Setting Up</b>	<b>27</b>
A.1 macOS . . . . .	28
A.1.1 Homebrew . . . . .	28
A.1.2 Ruby . . . . .	28
A.1.3 NeoVim . . . . .	28
<b>附录 B A Quick (and Hopefully Painless) Ride Through Ruby (with Cartoon Foxes)</b>	<b>30</b>
B.1 语言以及我所指的语言 . . . . .	30
B.2 语句的组成部分 . . . . .	32

# 第 1 章 Language and What I mean Language

The fundamental aim in the linguistic analysis of a language L is to separate the grammatical sequences which are the sentences of L from the ungrammatical sequences which are not sentences of L. The grammar of L will thus be a device that generates all of the grammatical sequences of L and none of the ungrammatical ones.

Noam Chomsky - Syntactic Structures (2nd Edition) de Gruyter Mouton (2002)

因为我只是一个业余的爱好者, 所以对于计算机语言的思考本该不是我所能够发言的, 我只能做到一个将前人的思想和智慧做一个整理和转述而已. 并且我的理解不一定正确, 请见谅.<sup>1</sup>

## 1.1 Magic is Basic<sup>2</sup>

心智的活动, 除了尽力产生各种简单的认识之外, 主要表现在如下三个方面:

- 1) 将若干简单认识组合为一个复合认识, 由此产生出各种复杂的认识.
- 2) 将两个认识放在一起对照. 不管它们如何简单或者复杂. 在这样做时并不将它们合而为一. 由此得到有关它们的相互关系的认识.
- 3) 将有关认识与那些在实际中和它们同在的所有其他认识隔离开. 这就是抽象. 所有具有普遍性的认识都是这样得到的.

John Locke - An Essay Concerning Human Understanding (1690)<sup>3</sup>

### 1.1.1 Intro: Once upon a Time

回忆小学学习算术的过程: 老师亲切地站在讲台上, 告诉下面的一帮小屁孩, 同学们, 你们看,  $1 + 2$  就是把 1 加上 2, 然后就是 3.<sup>4</sup>我们就不解释这个加法的过程了,(很遗憾, 我也不得不默认你们知道一些东西, 像那些一般的教材一样为了简洁而略去许多的说明), 如果只是形式地看看这个算式的话:<sup>5</sup>

$$1 + 2$$

我们会发现, 这个算式有着这样的结构: 符号 + 将它左边的东西, 不管是什么, 和右边的东西加在一起, 然后得到了一个值(在这里就是 3).

然后, 我们会接触到更加复杂的数学表达式, 比如:

$$1 + 2 \times (3 + 4/2)$$

<sup>1</sup>假如你觉得不够过瘾的话, 可以看看我的参考资料, 或者你觉得我在哪里有所不足的话, 欢迎指出.

<sup>2</sup>我记得阿瑟克拉克曾经说过, 一切科技发展到了一定程度都会像是魔法一样. 这样的原因究竟是因为科技发展得太过高级和复杂呢? 还是因为科技的使用的过程中存在着抽象的构造, 导致了人们不需要了解科技内部的构造和原理, 所以导致了使用过程中的无意识呢? 我说不清楚. 虽然抽象的方式可以使用简单的原理构造出复杂的过程, 也可以将复杂的过程抽象为简单的过程, 这样让使用者可以轻松地使用, 也会导致使用者可能会盲目地使用, 最终失去了对技术的理解而变成了像是面对魔法一样的敬畏和恐惧. 但是我认为, 哪怕是魔法, 也是有吟唱的技术的, 出了错的话就会爆炸之类的. 所以我们完全没必要神话技术, 只需要找到技术背后的计算过程的一些简单的原理, 我们就能够理解并实现最后的咒语吟唱. 然后成为大魔法师. 所以魔法是很简单的, 科技也是如此, 其实是一种每个人都能掌握的技术.

<sup>3</sup>来自 SICP. 一本我觉得很好的书.

<sup>4</sup>具体的到底有没有说是一个手指加上两个手指就是 3, 还是一个苹果加上两个苹果就是三个苹果什么的, 我已经完全记不得了. 但是这个时候假如有个天赋异禀的小孩子说了一句, 老师, 我还知道  $2 + 1 = 1 + 2$  因为这个关于加法组成一个 Abel 交换群. 害, 一个法国数学笑话了.

<sup>5</sup>下面的例子来源于王垠的博客, 里面的东西比我的要多, 并且我觉得写得比较有意思.

这个公式计算肯定不太麻烦的, 但是如果我们在乎这个公式到底在做什么, 而是形式上的思考这个公式的形式结构,(为了减少计算符号的优先级, 我们可以将这个公式加上括号来看):

$$1 + (2 \times (3 + (4/2)))$$

虽然这个公式非常的简单, 估计大多数的人都能一眼望穿, 但是请想想当年那个还在抠鼻屎玩的那个小屁孩, 小孩子接触到这个算式的时候是如何计算的? 大概是这样的一个过程:

首先看到的是最外层的这个加号, 然后就要将 1 和括号的值加起来, 于是去计算括号, 也就是  $2 \times (3 + (4/2))$  的值, 然后就要将 2 和新的括号  $(3 + (4/2))$  的值乘起来, 于是为了计算  $3 + (4/2)$  的值, 继续往括号里面看... 如此往复, 直到到了最后一层的括号里面, 因为没有更深的括号了, 所以就只是一个简单的表达式, 可以进行简单的计算了.<sup>6</sup>

于是我们计算出了最终的  $4/2$  的值 2, 然后将这个值代回上一层的括号里面, 计算出  $3 + 2$  的值 5, 然后计算  $2 \times 5$  的值 10, 然后再计算  $1 + 10$  的值...

天呐, 原来口算竟然是这么复杂的一件事情吗? 看来我口算水平差也是情有可原了. 当然不是, 我们会发现, 第一步的过程, 也就是"看"表达式的过程, 在我们的大脑里面几乎是潜意识一般自动地处理的, 而第二步的算值的过程, 则不过是非常简单的两个数之间的四则运算而已.<sup>7</sup>

在第一步的时候, 我们并不关心表达式的值, 而是关心表达式的形式结构, 然后在解构了表达式的形式结构之后, 我们得到了表达式的一个类似于下面的图形一样的逻辑结构:

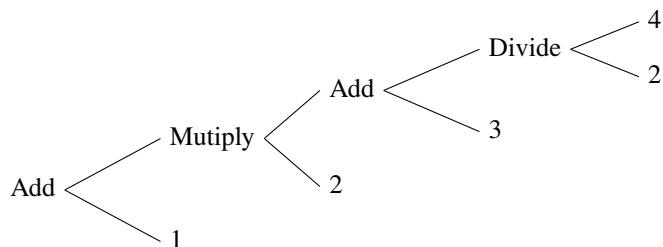


图 1.1: 一个简单的图示

下面的例子虽然看起来非常"可怕"<sup>8</sup>, 但是我们只会从形式上去看看这个东西, 所以不要害怕. 因为你会发现除了简单的加法, 一些复杂抽象的数学符号也是同样拥有着类似的形式结构(因为排版的缘故, 所以图跑了, 但是没关系, 翻到下一页就能够见到了.):

$$\forall \varepsilon > 0 (\exists N \in \mathbb{N} (\forall n > N \in \mathbb{N} (|a_n - a| < \varepsilon)))$$

这个东西虽然看起来好像是那么一回事, 但是在形式的分解下, 就会变成一堆非常简单的小关系, 或者是小逻辑, 小运算. 这样的分解就让我们能够理解和处理庞大的东西或者是抽象的东西.

其实我认为不仅是学习算术的过程, 比如我们在日常表达的过程中, 或者是读一句话的时候, 我们也有类似的解析过程. 比如说下面这句话:

幸せな君の隣にいたいよ

我们可以将这个句子分解成下面的结构<sup>9</sup>: (((幸せな)君)の)隣に)いたいよ, 变成这样的结构之后, 每一个

<sup>6</sup>这里我最近有一个想法, 就是是不是可以理解为我在看到优先级更高的计算的时候, 我将目前在做的事情先搁在一边, 类似于保存一个状态, 等到优先级更高的计算完成之后, 再回到原来的状态, 和新的结果进行一个运算, 相当于是推迟了计算. 有一个延时的感觉.

<sup>7</sup>相比对于四则运算你应该是没什么问题的了, 所以我们就将第二步的分析给略了先.

<sup>8</sup>其实不过就是一个极限的定义, 是从我的教科书上抄来的, 为了方便讲解

<sup>9</sup>首先声明一下, 我不会日语, 我也不会什么语法分析, 也许以后会学, 但是现在不会, 这个可能只能算是我的一种瞎掰而已, 不过无伤大雅, 可以使用就好.

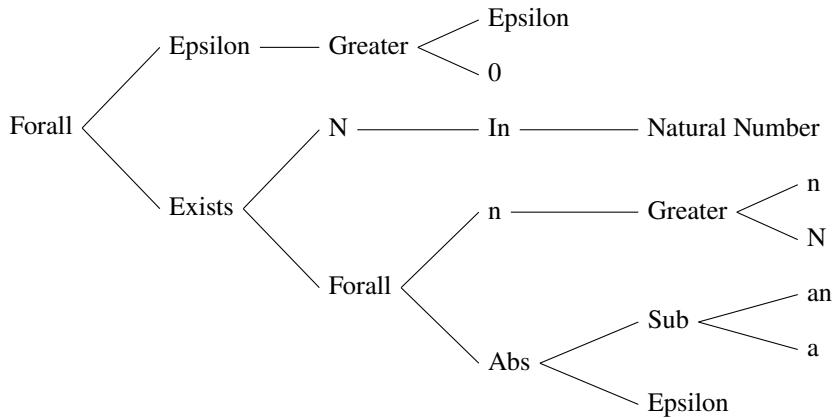


图 1.2: 还是一个简单的图示

部分都不过是一个(看起来)简单的东西了。<sup>10</sup>

那么这和计算机语言又有什么关系呢? 抖个机灵, 没关系, 因为世界上不存在所谓的计算机语言:

My conscience won't let me call Ruby a *computer language*. That would imply that the language works primarily on the computer's terms. That the language is designed to accommodate the computer, first and foremost. That therefore, we, the coders, are foreigners, seeking citizenship in the computer's locale. It's the computer's language and we are translators for the world.

But what do you call the language when your brain begins to think in that language? When you start to use the language's own words and colloquialisms to express yourself. Say, the computer can't do that. How can it be the computer's language? It is ours, we speak it natively!

We can no longer truthfully call it a *computer language*. It is *coderspeak*. It is the language of our thoughts.

### why the lucky stuff - Why's (poignant) Guide to Ruby

当然, 相比自然语言, 计算机语言为了方便计算机的理解, 它的结构形式就非常的明显并且简单, 比如经常被自己的圆括号表达形式而被嘲讽的 Lisp 语言,<sup>11</sup>或者是形式比较单一呆板的汇编语言,<sup>12</sup>虽然它们的形式结构虽然在某种程度上面给人一种不明觉厉的高级感, 但是假如你不要把它们想得那么复杂可怕, 其实编程语言就是这样的一个东西: 一种有着特定形式的语句, 按照这些语句所表现的一种结构能够用来描述逻辑过程. 就像是普通的汉语或者英语一样, 是用来描述或者表达一种过程的.

本来到了这里我应该至少开始介绍一门编程语言了, 然后让你在计算机屏幕上上面打印个什么, Hello World, 是吧? 然后让你乐乐, 觉得自己很厉害, 然后被我骗进来学一堆乱七八糟的东西, 最终成为掉头发的社畜. 呃, 我说了, 我只是一个业余爱好者, 并且我发现前面的风格如此随意, 就不太好突然降智, 画风突转什么的, 所以我打算把语言的介绍再往后推一点点. 就一点点.<sup>13</sup>

回到之前的小学算术题目:  $(1 + (2 \times (3 + (4 / 2))))$ , 假如我们已经分解好了结构, 那么每一次的计算只不过是加减乘除的最简单的操作了.

<sup>10</sup>然后在理解的时候,(有点像是上面的求值计算过程),就会变成这样: (((幸福的) 你的) 身边) 待着. 假如再加入一点点"达"的成分, 没准可以翻译成待在幸福的你的身边. 不过这个应该没关系了. 假如你觉得上面的例子有点不太过瘾, 你没准可以将这个模型应用到其他的语言里面. 虽然我不能保证一定适用但是我能够保证这个一定适用, 但是总归是一个有意思的思考方式. 并且我认为我的英语就是这么一个理解的过程的. 嗯, 加入以后有可能的话, 感觉可以试试往这个方向上做一些东西.

<sup>11</sup>有一个非常不恰当的比喻, 就是 Lisp 就像是中国的社会主义的思想是非常的美妙, 但是其他某些人说, 呃, 你好无聊啊, 有那么多括号, 你的形式好奇怪啊. 然后就被边缘化了. 然而很多语言都在"暗地"里学习 Lisp 的特性, 比如宏(macro), Lambda 演算, eval 等等. 因为感觉是真的香.

<sup>12</sup>呃, 这个暂时不要求. 因为我认为学习汇编对理解计算机程序设计没有帮助. 但对理解计算机构造比较有用. 还是先学设计, 对文科生友好一点...

<sup>13</sup>可以去读一下 why 的书的第三章, 我最近会把它翻译掉.

计算机其实也就是这样的, 它能够做到的也就只有简单的计数和运算而已, 之所以能够实现那些像魔法一般的神奇的结果, 其背后也不过就是最简单的运算而已. 计算机在处理编程语言的时候会将它们分解成一定的结构, 然后再对那些最小的结构进行运算和处理.

举一个例子:

$$\sum_{i=1}^5 i$$

假如我们不是伟大的高斯同学, 看到了这个  $\sum$  符号, 我们可能会把它看作是这样的东西: 对于每一个  $i$  从 1 到 5, 将它们都加在一起. 对于人来说, 这很轻松:  $1 + 2 + 3 + 4 + 5$ , 虽然麻烦了一点.<sup>14</sup>

那么按照前面朴素的想法, 肯定会想要知道这个结构是什么样的呢? 没错, 就是:  $(1 + (2 + (3 + (4 + (5))))).$  假如我们是一个不那么聪明的计算机, 我们会这样做: 将  $4 + 5$  的结果和 3 相加, 然后将这个结果再和 2 相加... 虽然这个过程用语言来描述会又臭又长. 但是假如你写了很多的话, 就会发现好像这个过程有一定的规律: 将什么和什么相加, 这个相加的结果重新用来进行新一轮的加法. 用语言来描述太复杂了, 用数学来表示:

$$\text{sum}(i) = \text{sum}(i - 1) + i, \text{sum}(1) = 1$$

这样的写法是不是比较抽象? 那么要不我们把它们替换看看?

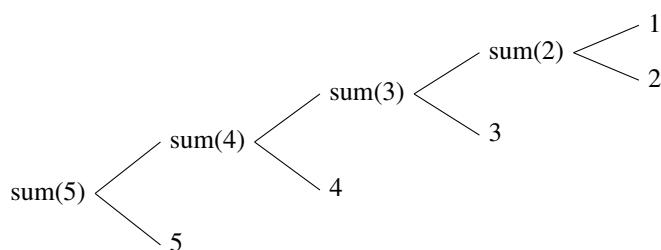


图 1.3: 还是一个简单的图示

这样的展开的形式是不是就能够理解了? 每一次做的事情是不是就不过是加法而已? 并且假如你想要计算  $\sum_{i=1}^5 f(i)$  的话, 是不是只需要将上面的数学公式描述改为  $\text{sum}(i) = \text{sum}(i - 1) + f(i), \text{sum}(1) = f(1)$  即可了? 假如你想要计算  $\sum_{i=1}^{5000} f(i)$ , 那么也非常轻松, 结果不就是  $\text{sum}(5000)$  嘛. 什么? 你问我结果是什么? 至少我是不会真的用手去算的. 我只会告诉你,  $\text{sum}(5000)$  就是一个这样的过程得到的结果: 这样的结果是将  $f(i)$  从 1 到 5000 的取值相加的过程的结果.<sup>15</sup>

上面的过程里面, 我们做的事情就是将一个求和的过程来分解成一组计算机能够理解的简单过程. 这样的话, 我们就可以让计算机来为我们干活了. 不过可惜的是, 我们的数学语言和自然语言可能很难对计算机生效. 所以学习一点点的计算机语言有点必要了.

## 1.1.2 异乡人啊, 你的管风琴在唱着什么歌谣呢?

一日一夜に 月は照らずとも  
悲傷しみに 鶴鳥鳴く  
吾がかへり見すれど 花は散りぬべし  
慰むる心は 消ぬるがごとく  
新世に神集ひて 夜は明け鶴鳥鳴く  
咲く花は 神に祈ひ禱む

<sup>14</sup> 传说这是高斯同学在小学的时候发明了对这样的级数的求和公式. 但是我们不会这样来做.

<sup>15</sup> 这里我觉得有必要来一点点的代码了: `def sum(i, f); i == 1 ? 1 : sum(i - 1, f) + f.call(i); end.` 之所以写成行的形式是因为我不知道怎么在注释里面写换行的代码. 但是我还是不开始讲编程语言, 唉, 就是玩.

生ける世に 我が身悲しも  
 夢は消ぬ 怨恨みて散る  
 傀儡謡 怨恨みて散る<sup>16</sup>

这是一首日语的歌曲的歌词。我摊牌了，其实我不会日语，所以下面的所有关于日语的语法的说法，绝对是我一己之见，请学日语专业的同学绕过我。但是没见过猪跑，总是吃过猪肉的。看到这么一串奇怪的东西，你的眼睛（大概）肯定会想要去寻找你熟悉的汉字。然后在这个过程中，虽然这些汉字怪怪的，但是你估计会下意识地将它们转换为简体字，然后再尝试处理它们的意思。<sup>17</sup> 比如在“新世に神集ひて 夜は明け鶴鳥鳴く”一行里面，我们可以先将那些词语进行一个替换（以及断句和联想）：“新世（界）\* 神（聚）集 \* \* 夜（晚）\* 明（亮）\* 鶴鸟<sup>18</sup> 鸣（叫）”。

假如说 I 告诉你，在日语里面的很多的语法结构有像这样的形式：修饰 + 对象 + 动作。<sup>19</sup> 那么我们再用之前的结构的形式来给上面的那句话一个结构：(((新)(世界))(神)(聚集)), (((夜)(明))(鹤鸟)(鸣叫))。这样的话，根据结构代入那些小单元，解释含义就能够得到语句的意思：（大概翻译）新世界的神聚集，夜明（黎明）时鹤鸟鸣叫。

假如没有前面的一个分析过程，直接遇到这样一个陌生的语言的文本，哪怕是其中好像有一些熟悉的字眼，可能也难以理解文本的意思吧。那么思考里面的过程，因为有了结构将语句分解成了简单的单元，于是只需要理解简单的单元的含义，就能够（通过结构）知道复杂语句的内容了。并且这个时候文本中的熟悉的字眼就成为了帮助我们理解语句的工具了。

其实对于一开始学习计算机语言的话，我们还是先从形式上来理解语言，这样的话，我们就能够阅读。这里建议阅读附录中对 Why's (Poignant) Guide to Ruby 中的部分翻译文字，我会从别的角度来介绍形式。<sup>20</sup>

Peter Norvig 在其书 PAIP 中写道：“Learning a language involves remembering vocabulary items (or knowing where to look them up) as well as learning the basic rules for forming expressions and determining what they mean.” 在我读到这段话后，我就决定按照这样的方法来展开我的书本。<sup>21</sup> 即，我们先仅仅从形式的角度来观察语言。而不必理解其中的太多的原理和含义。

毕竟每一个方士做法的时候，都要搭起法台，做好仪式的形式工作。并且形式的摆放有时候也能够产生巨大的作用。<sup>22</sup> 那么没关系，为了进入这个魔法世界，所以我们还是先从形式上入手，至少搭好一个作法的平台，能够读懂前辈道士用魔法咒术的形式传来的 Wi-Fi 密码...

(3 - 2) \* (3 + 2)

1 \*\* 3

<sup>16</sup> 来自我很喜欢的电影里面的音乐。Apple Music: Kugutsu Uta Uramite Chiru.

电影里面的故事比较有意思，当虚假和现实难以区分的时候，那么我们该对虚拟保持什么态度？计算机技术其实已经发展到了能够实现电影里面哲思的地步了：比如押井守（导演，或者叫监督？）就认为手机就像是影片中的电子脑——影片中的世界，人们无法离开资讯世界与电子脑，现实生活中，人们也同样无法离开手机或者电脑，或者说，由它们组成的资讯的世界。人与人之间的连接越来越多（lain），但是连接的关系越来越脆弱... 就像是蜘蛛结网，为了在摇曳的风中保持着网的完好，不得不向四面八方射出自己的丝，肚囊因此越来越扁。

虚拟的困境在于其是转瞬即逝的。这个无关技术是否先进。我还记得蒋勋说红楼梦里面有一段故事，（大意）讲的是一个大户人家，千金小姐嫁人，华贵的轿子在路上遇雨，躲到和另一户贫穷人家的小轿子遇见了，处于同情心，她就将自己妈妈送给她的陪嫁品——一个小小的像首饰盒一样的东西送给了那位穷小姐。毕竟那个盒子看起来也十分不起眼。然而命运多舛，她嫁的家庭一瞬间荣华富贵全部消失，自己也沦落一个普通的打杂的人，在一家大户人家打工。这家大户人家的主人有一间珍贵的阁楼，不让任何人进去，有一次这位曾经的千金小姐机缘巧合地走进了那间阁楼，却突然发现里面供奉着的就是自己当初送给那位穷小姐的首饰盒——原来那位穷小姐一家靠首饰盒里的财富做生意才发了家。荣华富贵就是这样一瞬间来，一瞬间去，好像一切都是那么虚幻和飘渺，转眼之间一切都改变了。

啊，好像有点偏离我们书本的内容了。

<sup>17</sup> 这也是为什么我选择日语的原因，因为它们借鉴了很多的中文，假如直接放汉字的话，就太容易识别了。无法体现出我接下来讲的语法处理的过程。又，还有一个原因是由于日语和中文一样，好像都是属于图形字符，这也是为什么弹幕视频网站在中国和日本比较流行，而在欧美国家很少。因为英语之类的东西很难像汉字一样模糊地阅读。

<sup>18</sup> 有说是一种妖怪，也有说是一种画眉鸟的，这我就没法说了。维基百科

<sup>19</sup> 注意，这个语法说法是我杜撰出来的。科不科学我不知道，但是秉承着物理的不完全归纳法，猜的厚脸皮我还是有的。

其实之所以这样讲，也是为了用来引出之后 Ruby 的语法：`UI::Button.new()`，如果你会的话，估计会觉得比较像。没错，就是因为我故意选的例子的原因。这也是为什么你觉得很多教科书上的例题都很好，然后实际拿来计算的时候就觉得不太对劲的原因之一吧。

<sup>20</sup> 之后我们会接触各种各样的语言，也就会接触各种各样的形式。不过和自然语言相比，计算机语言因为是一种类似于人造的语言，所以相比有着更多形式的，更多语法的自然语言，计算机语言的形式其实都是非常简单的。

<sup>21</sup> 其实 why 的书的做法就是这样的。

<sup>22</sup> “谓东方青帝灵威仰，主春生。南方赤帝赤熛怒，主夏长。西方白帝白招拒，主秋成。北方黑帝汁光纪，主冬藏。中央黄帝含枢纽，则寄王四时。以四时化育，亦须土也。更加昊天上帝耀魄宝，则为六帝。”摘自吕思勉《理学纲要》第二篇

5 / 3

5.0 / 3

这些数学运算大概是比较简单的吧? 加 (+) 减 (-) 乘 (\*) 除 (/) 还有乘幂 (\*\*), 都是一些比较熟悉的操作, 并且和数学里面的运算优先级一样, 都是从左到右, 括号优先, 乘幂最大, 其次乘除, 最后加减.<sup>23</sup>

倘若我们想要把数学运算的结果保留下来, 那么我们可能在计算器(以卡西欧 fx-991CN X 为例)上这样操作:  
 1 [+] 1 [STO] [x], 于是我们就能够把  $1 + 1$  的计算结果储存到变量  $x$  中了. 那么在 Ruby 里面, 我们可以这样形式地记做:

```
x = 1 + 1
money = 100 * x
lunch_fee = 50
money = money - lunch_fee
```

并且就像是数学里面对变量的计算一样, 我们可以像这样用变量来计算. 不过是一个形式而已. 于是看到类似的结构之后, 我们不妨就理解为  $x$  is  $1 + 1$ .

```
door = Door.new
door.paint :red
door.open
```

假如你会一点点英文的话, 上面那段文字估计就算不需要学计算机也是会的吧.<sup>24</sup> 翻译翻译就是: door is a kind of Door. paint red to door. open the door.<sup>25</sup> (door 是一扇 Door(门), door 涂成了红色, door 打开<sup>26</sup>)

这个时候我们相当于接触到了第一个在平时比较陌生, 但是在 Ruby 程序编程中的比较普遍存在的一种语言形式: a.b(c). 这种形式在某种程度上可以说是 Ruby 的一个非常普遍存在的形式:

```
player.move(:up, 1 * cm)
player.attack :excalibur, enemy
player.defence
```

上面那段有点像是一个 RPG 游戏了, player(玩家)在地图上向上走了 1cm(厘米), player 使用了 excalibur(EX-咖喱棒) 攻击 enemy(敌人), player 防御.

如果仔细观察的话, 会发现这里的一些操作和上面的门的语句有一些十分类似的地方: 都是 什么东西. 做了什么动作(描述动作的一些信息), 或者是把括号去掉的形式.<sup>27</sup> 假如用一些专业一些的语言来说的话就是, 上面的语句的结构就是对象. 方法(方法参数).<sup>28</sup> 上面的"对象", "方法", "方法参数" 我们不妨先采用形式化的定义而不是深究其具体的实现和原理.

<sup>23</sup> 唯一需要注意的是, Ruby 在处理整数(Integer)的除法的时候, 会和 C 一样保留整数, 也就是说,  $5 / 3$  会得到 1, 而对于浮点数(Float, 或者可以理解为带小数位的数)参与的除法, 如  $5.0 / 3$ , 则会得到 1.66666667 类似这样的结果. 不过这些不是我们目前需要着急在意的事情, 这些只是出于设计考量而做的一些妥协而已. 是一种类似于约定主义的东西. 请把目光聚焦在这些表达式的形式上吧.

<sup>24</sup> 这个时候就可能会有人说, 啊, 少来了, 不过就是什么现在我们已经学会了  $1 + 1$ , 然后开始积分求和求累乘吧之类的. 呃, 怎么说呢, 还真是. 因为我觉得 Ruby 的很多美妙的操作是编程的程序员在相互维持着的, 比如那些很好看的 method 名字和很标准的 ri 文档, 这些虽然有 Ruby 本身的一些工具或者属性的夹持, 但是更多时候还是有赖于开发者的自觉. 比如要求良好的命名规范, 良好的注释, 以及 duck philosophy(If something quarks like a duck, then treat it as a duck.) 等等. 不过在自己写代码的时候可就不一定能够保证自己能够写出这样高质量的代码了, 所以假如你不注意的话, 就会写出一些看起来非常"极客"的东西: var\_0xffff.f\_0x999.f\_0x233 :aaaza. 假如还没有很好的排版(缩进)和注释, 就会导致整体的混乱以及难看. 不过嘛, 我们又不是什么自虐狂, 何必要这样折磨自己呢? 一般我们还是希望除了写代码的基本逻辑以外, 花一些时间和精力在美化代码, 或者让代码变得更加优雅之类的都是非常重要的.(虽然我觉得自己目前还没有那种能力...) 不过也有因为这样导致的诟病, 比如说, 虽然和 python 很像, 但是 python 以其简洁和标准常常带来的唯一的问题解决方案, 而 Ruby 却是写完了功能之后程序员为了追求奇技淫巧还在一直重构... 嘛, 没有什么好坏之分啦.

<sup>25</sup> 虽然我突然发现日语的语序和这个真的是超级像诶: door を赤く塗る, door を開ける.

<sup>26</sup> 这里我们就不要考虑什么主动型还是被动型了, 这点语法小问题而已, 宽容一点啦.

<sup>27</sup> 这是因为 Ruby 比较灵活, 可以在它能够理解的范围内自动添加上括号. 但是假如会出现歧义的语句的时候, 就不能够这样忽略括号了, 否则会导致 Ruby 无法理解你想要表达的意思, 最后出错.

<sup>28</sup> 英文的说法是 object.method(arguments), 其中方法的参数不一定只有一个, 并且也不一定只有特定的长度.

有些时候, 虽然有时候会遇到某些看起来不像是满足上面的形式的代码, 这些代码其实是因为 Ruby 支持简化书写一小部分语法格式.<sup>29</sup> 比如:

```
print("I'm Groot. ")
gets()
```

上面两个虽然看起来没有`a.b(c)`的形式, 但是假如将其完全书写, 就会发现其形式仍然是满足的. 即`Kernel.print("I'm Groot. ")`以及`Kernel.gets()`<sup>30</sup>的形式.

我们传入的方法参数的形式有时候可以是空空如也的, 有时候可以是一连串的值, 有时候也可以是一些特殊的形式:

```
5.times { print "I'm Lucky!!! " }
```

这句话有点像是: do the action of printing "I'm Lucky!!! " 5 times. (打印五次"I'm Lucky!!! ", 嗯, 人类的本质就是复读机.)

或者是:

```
"I'm Lucky".each_char do |c|
  print c.upcase
end
```

这句话就像是: do the action of printing the upper case of c, aka. each single character in the string "I'm Lucky". (取出"I'm Lucky" 这串文字里面的每个字符, 简记为 c, 将每个字符都以大写的方式打印到屏幕上去.)

也许你会觉得, 啊, 好麻烦啊, 上面两个例子又出现了一些新的形式, 这些形式好怪啊, 要学的东西好多啊. 还请不要怕, 其实我们并没有引入任何新的形式.

其实在上面的代码中, 前半段的形式相比一定是让人十分眼熟的吧: `5.times`, `"I'm Lucky".each_char`就像是一种`a.b`的形式. 而后半段, 则出现了一个`do ... end` 和`{ ... }` 的形式. 这个时候, 我们不妨在形式上抽象地将其视为是一个整体, 就好像两者是一个特殊的"括号", "括号" 里面包含了许多的内容. 这些内容具体是什么我们不必担心, 我们只需要像是对待字符串(一个被引号括起来的东西)一样, 对待这样的东西就好. 形式化地命名为代码块(block).

所以我们在形式上就是这样理解的: 对于对象的某个方法, 我们传入了一个代码块来进行处理.<sup>31</sup>

接着, 假如我们承认了这样的形式的话, 我们会发现, 数字和一个用引号包围的文本(字符串)也就是处于对象的一个位置, 那么从这个形式上猜测, 我们也不难说, 数字和字符串也不过是对象(object)罢了.<sup>32</sup> 所以既然是对象, 就可以像我们之前一样的形式来处理和工作. 这就是一种形式而已.

那么简单的分析我们就介绍到这里了. 总结一下:

- 在 Ruby 里面, 所有的东西都是对象(object), 所以处理起来的形式就像是`object.method(arg)`这样的形式.<sup>33</sup>
- 有一种叫做代码块(block)的东西, 通常出现的形式是`do ... end` 的形式<sup>34</sup>, 里面可以包一些"馅料"并传递给方法.

<sup>29</sup>不知道这算不算是一种语法糖

<sup>30</sup>Kernel 是内核的一个意思.

<sup>31</sup>在 ruby 里面, 方法定义是这样形式记录的: `def method_name(param, *multi_param, &block)`, 我们的代码块用`block`标记, 不难发现其传入的形式就是用一个`&`在前缀标记的.

<sup>32</sup>其实在 Ruby 里面, 所有东西都是对象, 并且所有的表达式都有返回值. 这个就是 Ruby 的一个核心的概念和特性.

<sup>33</sup>比如四则运算的运算符, 其实也可以定义为`3.* 2`这样的形式, 只是为了方便人的阅读, 所以一般不会这样写而已; 或者是赋值语句, 也是内部有类似方法的定义的, 并且为了最后的美观提供了一个额外的写法而已.

<sup>34</sup>也可以只是一对花括号`{}`, 这样的写法和`do`与`end`是等价的, 只是在单行写起来会好看一点.

那么在最后给出几个例子, 来结束我们这部分对简单的语句形式的介绍.<sup>35</sup>

**例题 1.1** 请将下面的一些代码片段变成a.b(c)的形式: 首先是一个循环的代码:<sup>36</sup>

```
loop do
  # ...
end
```

然后是一个更加普通的打印输出的代码:<sup>37</sup>

```
puts "How are you? "
```

并且就像是之前的算数一样, 我们的这个形式也是可以堆叠起来形成一个更加复杂的结构的:

```
"(A) (good) (quote) (is) (something) (like) (this)".gsub("( ", "") .gsub(")", "") .upcase
```

这是因为在 Ruby 里面, 所有的表达式都有返回值, 于是语句前面的表达式的计算结果就会作为后面的表达式中的对象的身份来继续执行. 用括号来表示运算的优先级就像是: (a.b(c)).d(e). 这样的话我们就拥有了通过简单的形式来构造复杂的语句的能力了.

**例题 1.2** 像之前的加法一样, 画出执行过程的分解, 比如上面嵌套的过程:

```
string.gsub("( ", "") .gsub(")", "") .upcase
```

就可以写成:

```
gsub ————— gsub ————— upcase
```

图 1.4: 虽然很短, 但是这是为了防止你觉得太简单

那么来点稍微给力点的:

```
puts("Hello, " + "cigaM ykcuL".reverse.upcase)
```

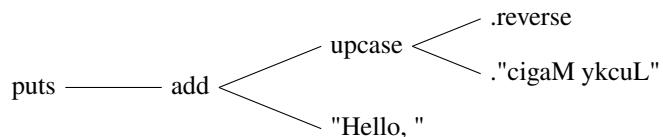


图 1.5: 执行的顺序是从右到左的, 请思考是为什么结构上看是从左到右的但是执行的顺序却相反.

那么试着看看别的代码来理解它们形式和执行的意思吧, 这个时候, 你已经可以在形式上读懂很多的代码啦. (又是一个关于你已经学会了 $1+1$ 了, 现在来看... 的笑话了.)

### 1.1.3 特殊和一般的形式

其实除了上面介绍的那种最最一般的形式, 在 Ruby 里面还有一些特殊的规则形式. 这些规则形式也许可能没有办法化简成普通的a.b(c)的形式, 所以会被 Ruby 特殊地去处理和执行. 我只会简单地介绍最常见的一些形式和规则, 然后在之后, 如果有需要的话, 再会增加介绍新的内容.

<sup>35</sup>注意, 这里的语句形式实际上是体现了一种叫做面向对象编程的思想的东西. 但是并不是说我们必须用面向对象编程的方式来进行编程. 并且我也不强调面向对象的编程. 只是因为这样的介绍方法非常符合 Ruby 的思想: "所有东西都是对象". 并且这样的面向对象的编程方法比较容易让别人接受.

<sup>36</sup>这个东西可以写成这样的形式: `Kernel.loop(&block)`. 虽然看起来像是一种特殊结构, 但是实际上也是一种a.b(c)的形式啦.

<sup>37</sup>可以写成这样的形式: `Kernel.puts(param)`

## 数据形式和字面值

我们简单的赋值形式其实就算是一种比较特殊的形式了. 不过一个等号的形式如此简单... <sup>38</sup> 那就暂时放过它吧.

那么对于我们赋值的东西, 也就是数据. 正如前面我们看到的1, 2, 3, 以及" 文字"之类的值的形式都是比较常见的数据的字面形式.

比如说, 我们可能想要储存一个三年 E 班花名册, 但是这个时候, 使用变量的话, 就意味着我们可能需要这样命名:

```
student_no_1 = "Akabane Karuma"
# ...
student_no_7 = "Kayano Kaede"
# ...
student_no_11 = "Shiota Nagisa"
# ...
```

这样是不是太麻烦了呢? 没错, 所以 Ruby 中构造了一种叫做数组 (Array) 的数据类型, 让我们能够将一连串的数据储存组织在一起. <sup>39</sup>

```
students = ["Akabane Karuma", "Kayano Kaede", "Shiota Nagisa"]
```

这个时候, 我们用方括号来表示边界, 然后在其中用逗号来区分两个不同的元素. 这个时候, 我们就定义了一个数组. 然后在访问的时候, 只需要告诉数组元素的位置, Ruby 就会为我们去寻找相对应的值. <sup>40</sup>

```
puts "Hello! " + students[0]
puts "Good night. " + students[1]
```

唯一需要注意的是, 这里有一个关于计算机的笑话: 某些程序员数数的时候老是和别人差一, 因为他们都是从零开始数数的. 这是因为计算机是从零开始表示最开始的位置.

但是有时候我们也会想要通过名字来访问, 就像是变量名字一样. 比如说我们想要一个字典, 对于输入的一个单词, 我们想要给出它的一个定义, 在形式上, 我们可以记作:

```
dictionary = {
    "book" => " ほん", "Tao" => " 道", " 气" => "Qi",
    " 镜子" => "mirror", " つき" => "moon"
}
```

我们把类似这样的形式叫做哈希 (Hash). 其中我们会发现花括号的形式和之前的代码块的形式类似, 但是它们的含义并不一致 (并且你也不能够用 do..end 来替换上面的形式, 尽管对代码块来说这样是可以的). 不过你可以理解为它们都表示了一大块的东西打包在一起的形式. 需要注意的是, 和数组的形式类似, 都是用逗号来分隔元素, 只是 Hash 中的元素是成对出现的而已. 并且上面的代码中的换行并不是必须的, 只是为了美观加入的.

<sup>38</sup>其实也可以有一点点有意思的地方, 这个我会在后面写啦. 因为这个东西有的时候可以看作是a.b(c)的形式, 但是有的时候又不能够这样看. 虽然这样的不一致性让人很不爽, 但是我们还是先接受吧, 我会想想有没有一种好的方式来解决这个问题的. 虽然这个时候 Lisp 的列表的形式就看起来很香了. 因为解释起来非常便利, 即列表的第一个元素就是函数, 而其后面的都是函数的参数.

<sup>39</sup>我们只会形式化地定义, 目前不会介绍如何具体实现. 后面会介绍如何实现.

<sup>40</sup>唉, 为什么这么想要知道怎么实现的呢? 好吧, 好吧. 我的答案是, 并没有一种绝对的实现方式, 比如说, 我们可以像 C 一样, 用针对寄存器的思考方式来实现: 通过设定基值和偏移量在地址上来寻找所对应的值. 但是我们又可以构造一个链表, 通过递归的方式来在列表上移动, 然后寻找所对应的值. 可是, 现在你知道了这些有什么用呢? 我觉得只会限制你的思想, 因为我觉得计算机就是一种约定主义: 如何通过约定的方式来约定一种结构是比较有意思的东西.

这个以后就体现了一种抽象的方式. 就像是在数学中, 我们定义了一种东西叫做集合. 对于集合的严格定义, 在使用的时候, 我们可以不必太过在意. 这就是一种抽象的概念. 但是这并不是说具体的实现和定义并不重要, 因为据我所知, 集合的定义也不是只有一种的, 比如 ZF 和 ZFC 集合论.

然后就能够用形式上处于 => 前面的标记来访问箭头后面的标记了:<sup>41</sup>

```
puts "The meaning of Tao is " + dictionary["Tao"]
puts "And the meaning of book is " + dictionary["book"]
```

上面我们简单介绍了一些组织许多数据的方法. 这样的数据结构在大部分的编程语言中都有类似的实现. 比如 Python 中的元组, 列表, 字典之类的, C 语言中的数组之类的. 这是因为 Ruby 从这些语言中吸取了经验. 最后介绍一种 Ruby 的数据表示形式: 符号 (Symbol): `:this_is_a_simple`, 在形式上看起来, 符号就像是一种用冒号开头的没有引号的字符串. 在大部分情况下, 符号是用于表示一些方便区分名字, 但是这个名字不太需要编辑修改的问题.

不过现在只需要知道, 这些数据对应的形式大概是这样的就足够了.

## 为过程命名

前面我们看到过了为值命名的变量, 其实不仅可以为值命名, 我们还能够为过程命名. 就好像是道士们口中念念有词, 召唤了一位金刚力士. 然后你就可以通过大喊他的名字来指挥他来帮助我们来干活.

比如前面的`print "I'm Lucky."`, 这个时候我们的金刚力士的名字叫做"print". 就像是友人帐一样, 当我们大喊它的名字的时候, (假如它是记录在帐的话), 它就能够跳出来, 然后接受我们的命令 (也就是我们传入的参数), 最后就老老实实地去执行自己的任务.

上面的都已经是见过面的结构形式了. 但是我们可能也会想要增加新的"朋友" 到自己的友人帐上. 这个时候, 我们就会需要有这样的形式:

```
def method_name (parameter)
  # ...
end
```

没错, 这个的形式和我们之前见过的形式都不太一样. 所以我们就将这个形式称为一种特殊的形式.<sup>42</sup>

举个例子, 假如有这样的一个方法<sup>43</sup>定义:

```
def puts_each_char(str)
  str.each_char do |c|
    puts c
  end
end
print_each_char("Lucky")
```

一个朴素的理解方法就是, 每当出现有调用的形式, 即`print_each_char(str)`的时候, 也就是上面的`print_each_char("Lucky")`, Ruby 就会把这个在定义中的`str`记作 "`Lucky`", 然后执行在定义的那一段代码.

**例题 1.3** 我们可以用这样的形式结构来为过程命名, 然后在之后进行调用. 这样的好处就是我们可以讲复杂的过程用一个简单的名字来代替. 这样我们就拥有能够将一个复杂过程化简成简单过程的能力.

比如说, 我们想要一个能说废话的小精灵, 当我们喊出它的名字(这里我把它叫做 `say`)的时候, 就能够输出一些类似于"令人讨厌的棉线写问题"这样的虽然看起来挺符合语法, 但是却没什么意思的随机句子.<sup>44</sup>

<sup>41</sup>因为这样的形式, 所以我们常常把前面的形式部分叫做 key(键), 后面的叫做值 (value). 就好像是拿钥匙去开锁, 对应的钥匙开对应的锁, 得到对应的数据. (或者是在钢琴上按键, 会弹奏出对应的音符.), 我们会在后面介绍一下这个东西的原理.

<sup>42</sup>这个时候我就有点想要介绍 Lisp 的语法了, 毕竟它只有一种形式, 这样的话学起来非常的漂亮. 但是问题是, Lisp 的思维太美妙了, 我认为目前我的理解不能够很好地表现这样的东西. 不过我最近在读了, 所以应该会把一些类似的东西合成进来.

<sup>43</sup>其实也有很多人把这个东西叫做函数, 在这本书里面, 两种叫法我都会使用, 所以加入你一下子看到我说方法, 一下子看到我说函数的话, 请不要觉得奇怪. 我在之后会尽量规范些使用的.

<sup>44</sup>这个例子来自于 PAIP 一书. 顺带一提, 这本书我觉得也很有意思. 原书所用的是 Common Lisp, 我这里给出了一个 Ruby 的实现.

```

# 一句废话由主谓宾构成
def say
    return subject() + verb() + object()
end

# 主语就是形容词加上名词
def subject
    return adj() + noun()
end

# 宾语也是形容词加上名词，所以和主语一致
def object
    return subject()
end

# 动词是...
def verb()
    pattern = ["缠", "解", "是", "写", "讨厌", "觉得"]
    return pick(pattern)
end

# 形容词是...
def adj
    pattern = ["细长的", "令人讨厌的"]
    return pick(pattern)
end

# 名词是...
def noun
    pattern = ["棉线", "问题", "书", "论文稿", "梦想", "想法"]
    return pick(pattern)
end

# 从一个数组中随便选择一个
def pick(arr)
    length = arr.length
    return arr[rand(length)]
end

```

在上面的过程中，我们实现了一个能够说出狗屁不通废话的小机器人啦。<sup>45</sup> 在某种程度上说，我们通过构造方法和抽象的方式来实现了下图一样的一个简单的语言结构：

<sup>45</sup>作为一个例子，我say()得到的结果是“**令人讨厌的问题是令人讨厌的想法**”，竟然还算是有点意思。毕竟这是因为词库选得好嘛。又，据说在早期的人工智障 Siri 里面，某些句子就是通过编写很好的词库来实现的。这种生成语句的方法的缺点十分明显：即前一句和后一句话可能是毫无关联的话语，即上下文无关生成；并且句子中的成分之间的关系也是十分脱节的，比如我们可能会得到一些类似于“**细长的问题写细长的问题**”这样的东西。不过哪怕是一个简单的例子，其实我们也有可以玩着要的地方。

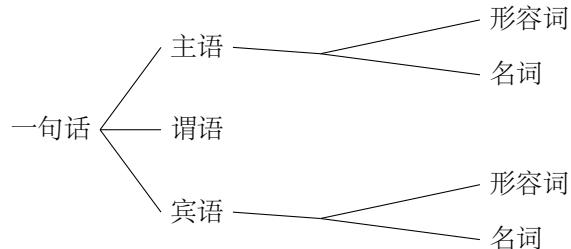


图 1.6: 注意, 我在这里写的语法结构是错的, 这是写完之后才发现的.

没错, 我在画完上面的那个树状图后就发现了不对劲的地方, 因为我把语法结构给搞错了.<sup>46</sup> 那么通常的想法应该是去修改函数之间的关系, 然后, 比方说, 加入一个补语的函数, 然后重新更改 `say` 函数的逻辑...

上面的逻辑体现了引入过程抽象(即函数)的好处: 我们只需要修改对应部分就能够达到控制整体的效果, 而在使用函数的时候, 只需要知道如何输入就会得到如何输出 – 即一个**黑箱方法**. 但是, 是不是总觉得这样仍然看不出来有什么特别便利的地方呢? 于是新的想法产生了, 如果我们能够构造这样的一个函数, 能够自动根据输入语法结构来生成结果的话, 在某种程度上来说是否就更加便捷了?

于是我们先做一些形式上的约定, 假如用一个哈希表来表示语法的转移规则, 每一个键对应着语法树上的一个节点, 用一个数组来表示节点的分支, 而用一个数组套数组的形式来表示从中任意选择一个值. 于是就可以构造出这样的一个树:<sup>47</sup>

```

grammar = {
    :句子 => [:定语_, :主语, :谓语, :定语_, :宾语, :补语_],
    :主语 => [:名词], :定语 => [:形容词], :宾语 => [:名词],
    :定语_ => [[ "", :定语]], :补语_ => [[ "", :补语]],
    :补语 => [:介词, :定语_, :主语], :谓语 => [:动词],
    :名词 => [["棉线", "问题", "书", "论文稿", "梦想", "想法"]],
    :形容词 => [["细长的", "令人讨厌的"]],
    :动词 => [["缠", "解", "是", "写", "讨厌", "觉得"]],
    :介词 => [["在", "于", "与", "和"]]
}
  
```

那么我们构造这样的一个过程来处理给定的 `grammar`:

```

# 根据规则来输出句子
# 定义中的 +res = : 句子 + 表示变量 res 的默认值就是 +: 句子 +
def generates(rules, res = :句子)
  out = ""
  rules[res].each do |item|
    if item.is_a? Array # 随机选择
      choice = pick(item)
      if choice.is_a? String
        out << choice
      elsif rules.include? choice
        out << generates(rules, choice)
      end
    else
      out << item
    end
  end
end
  
```

<sup>46</sup>形容词应该归为定语的成分的, 不过实在是尴尬, 语文学的不太行.

<sup>47</sup>在 `LATEX` 里面, 我使用的是 `minted` 包来输出代码的, 但是看来对中文字符的支持好像不太行?

又, 因为尝试使用 `tikz` 包画出语法树而形态爆炸, 所以我决定先不放语法树了, 请留作自己的思考吧. (放心, 用手画很快的, 只是 `tikz` 包有点坑爹. 标记型语言真是狗啊.)

```

    end
  elsif item.is_a? Symbol # 是节点
    out << generates(rules, item)
  end
end
return out
end

```

这个时候我们的程序实现中就只有一个函数<sup>48</sup>, 并且和之前为描述过程构造函数的想法不同, 这个时候我们是通过为带有一定格式的数据(也就是我们用来表达 **grammar** 的语法结构的数据)构造一个脚手架 - 你可以这样想象: 我们的程序就像是一个运行在有结构的数据上的一个机器.

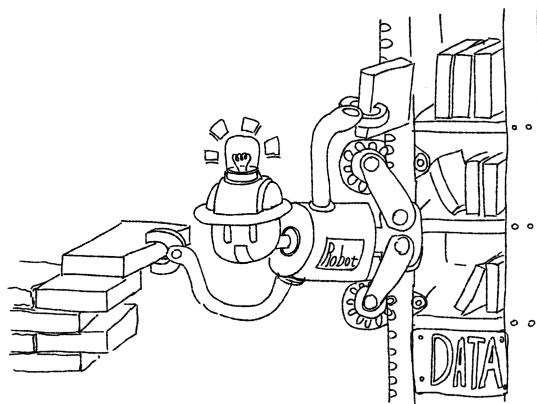


图 1.7: 我们的机器运行在有结构的数据上

这个时候我们的思路是这样的: 用一种方式描述数据, 然后构造能够将数据转化为过程的函数. 这个时候, 可以说我们不仅仅是将原本的过程抽象为一个名字了, 而是将数据当作过程, 进行了一次抽象.<sup>49</sup>

并且这样的抽象还有一个好处, 就是能够支持处理所有满足这样形式数据, 也就是说, 哪怕这段 **generates** 代码并不是用来生成上面的句子, 而是用来生成一篇狗屁不通议论文的话, 也能够通过类似的结构来生成文章.<sup>50</sup> 或者是来一个可以随机编程的机器, 比如通过 **a.b(c)** 的结构来随机生成一段 Ruby 代码. (就作为一个思考游戏吧. 没准能够有点用处.)

## 类型和模块

我们前面遇到的普遍类型都是有着 **a.b(c)** 的形式. 其中 **a** 被我们形式地称为对象, **b** 被我们形式地称为对象的方法, **c** 被我们形式地称为对象方法的参数.

并且我们也知道, 对象有许多的种类, 比如我们前面介绍过的数组, 哈希, 字符串, 整数, 符号等等. 这些种类有着它们自己的名字. 并且假如我们也想要构造类似的种类的话, 我们可以如下形式地定义:

```

class Name
  def method_name
    #
  end

```

<sup>48</sup>如果你试着运行的话 **generates(grammar)**, 你就会得到类似于"想法缠想法和书", "令人讨厌的论文稿是细长的棉线" 这样无厘头的废话.

<sup>49</sup>我们会在之后继续介绍一些关于这样抽象的想法.

<sup>50</sup>这里只是给一个我高中想到的无聊的框架, 只是一个概念, 实际的实现肯定没有那么简单吧. 比如说: 段落 => 论点, 例子, 论点; 文章 => 下定义的段落, 讲缺点的段落, 讲适度利用的好处的段落, 还有啥我忘光了...

```

end
# ...
end

```

比如说我们今天遇到了一个神奇动物师, 比如说他养了一堆不同的神奇动物在自己的帽子里.<sup>51</sup>因为他是专业养殖户(误), 所以对神奇动物的了解和分类是一流的. 比如说, 对于 **LazyCat**, 它的叫声是咪咪呼呼; 对于 **LostPigeon**, 它的叫声是咯咯咯咯; 而对于 **DearDeer** 和 **TwoEarRabbit** 尽管它们的叫声, 据这位资历丰富的动物学家所说, 还未被人类听过, 但是它们都属于 **MagicAnimal**.



图 1.8: 动物学家

于是我们就能够用上面的例子来形式地定义这些神奇的动物:

```

class MagicAnimal
  def bark
  end
end

class LazyCat < MagicAnimal
  def bark
    puts " 咪咪呼呼"
  end
end

class LostPigeon < MagicAnimal
  def bark
    puts " 咯咯咯咯"
  end
end

class TwoEarRabbit < MagicAnimal
end

class DearDeer < MagicAnimal
end

```

<sup>51</sup>其实我觉得在帽子里养神奇动物更有意思. 毕竟据说英国绅士都有一顶绅士帽.

于是他坐下来和我们聊了聊他和自己的小猫 **kitty** 的发现过程, 在神奇的一个月夜, 坐电梯到楼下扔垃圾, 然后在雾蒙蒙的视野里, 忽然看见了一双黄色的眼睛, 闯过了灰暗的雨雾在墨绿色的垃圾桶上闪耀. 两个家伙面面相觑, 就像是被施了定身术一般, 时间停止了流动.

嗯, 所以用 Ruby 的代码来描述就是:

```
kitty = LazyCat.new
kitty.bark # => 咪咪呼呼
kitty.class # => LazyCat
kitty.class.superclass # => MagicAnimal
```

这个时候我们就得到了一只 **LazyCat**. 并且我们也能够知道它的从属关系. 并且就像是那位魔法动物学家坚信的那样, 所有的 **MagicAnimal** 都应该会 **bark**, 只是不同种类的叫声不同. (虽然正如他所说, 还有一些种类的 **MagicAnimal** 的叫声他并没有听过)

除了像这样的 **class...end**, Ruby 中还有一种 **module...end** 形式的一组关键词. 后者在定义的形式上和前者别无二致, 所以我便不再介绍了. (但是在后面会详细介绍相关内容的<sup>52</sup>)

## 1.2 Get Your hands dirty

我的目的是想说明, 这一天空机器并不是一种天赐造物或者生命体, 它只不过是钟表一类的机械装置 (而那些相信钟表有灵魂的人却将这一工作说成是其创造者的荣耀). 在很大程度上, 这里多种多样的运动都是由最简单的物质力量产生的, 就像钟表里所有活动都是由一个发条产生的一样.

约翰尼斯 · 开普勒 (给 Herwart von Hohenburg 的信, 1605)

上面的故事里面, 我们只是在做类似于思想训练的东西, 没有接触很多实际的操作. 这怎么可以呢. 只是想想, 却不实际操作, 总觉得有一种画饼的意味. 所以接下来我们来实际地接触一些实际的东西.

(又: 我现在觉得实际的东西还是在迟一点点来吧, 下面的东西请务必操作<sup>53</sup>试试看. )

### 1.2.1 过程视为数据

舍利子, 色不异空, 空不异色, 色即是空, 空即是色, 受想行识, 亦复如是. 舍利子, 是诸法空相, 不生不灭, 不垢不净, 不增不减.

《般若波罗蜜多心经》<sup>54</sup>

我们的在前面构造的说狗屁不通废话的小机器人是将数据看作一种组织生成语言的过程. 我们构造了 **grammar** 这个数据表, 描述的是如何生成一句话的过程, 即通过主谓宾这样的结构来生成句子的一个过程.

比如说我们写下了一段代码, 并将其保存在文件里面, 然后运行这段代码<sup>55</sup>, 这个时候我们是把过程视为数据储存在计算机中. 在 Ruby 中 (以及其他的一些编程语言中) 都有一种 **eval** 的函数, 能够将储存在数据中的过程视为代码并执行:

(在 irb 中运行代码)

<sup>52</sup>这个东西叫做 Namespace, 如果想要提前了解的话, 可以去搜索相关的内容.

<sup>53</sup>关于操作的问题, 建议使用 irb(Interactive Ruby) 来操作. 关于这个, 可以看看我的这个 [翻译](#). 之后会在附录中加入一些关于这个 irb 的介绍的.

<sup>54</sup>在维基百科上对这句话的解释是: 一切的"形态", "影像", "物质" 都是"无常", "非固定", "不恒久" 的.

说一个没什么关系的事情. 我记得很久之前我读过一篇相关的文章, 写的很不错. 但是现在在网上怎么找也找不到了. 真是麻烦.

(下面的理解不过是我个人的主观臆断, 只是为了方便引入我下面介绍的内容所以介绍的. 请专业佛学院的读者放过我. ) 既然空 (无常与变化) 和色 (物质与形态) 并非是对立的关系, 那么便不必将二者对立起来. 与其坚持"说一切有", 或者是为了追求"空"而追求. 所以与其追求一极, 不如观察其中的统一性而辩证地去认识.

<sup>55</sup>具体的方法就是: 用文本编辑器编辑一段代码, 常见的文件名后缀是 **.rb**. 然后在终端中执行命令 **ruby filename.rb** 就能够执行代码了. 或者也可以通过 **irb** 来运行代码.

```

3.0.0 :001 > data = "puts 'Hello'"
=> "puts 'Hello'"
3.0.0 :002 > eval(data)
Hello
=> nil
3.0.0 :003 > puts 'Hello'
Hello
=> nil

```

可以发现,作为字符串储存的一段代码数据,现在就能够被作为代码来执行. 在 `eval` 函数的背后,其实有一个和我们前面构造的 `generates` 函数相类似的构造,能够处理代码. 将保存在文本文件里面的 Ruby 代码变成计算机能够理解的过程数据.

但是这个时候我们不妨这样想: 既然我们可以用数据来描述过程.<sup>56</sup> 那为什么不能够把过程看作是数据, 来对像数据一样的过程进行处理呢?

于是, 基于这样的想法就构造了一种 Lambda 表达式, 即一种能够生成过程的表达式.

```

func = -> (param) {
  # ...
}
func.call(param)

```

上面就是形式的 Lambda 表达式的定义方法. 在形式上, 我们就像是构造了一把宝剑.

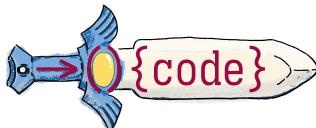


图 1.9: 看我的大师剑!

矮人族的工匠们都知道如何打造一把优秀的剑: 握住剑柄(也就是我们的`->`部分), 郑重地在其上镶上能够赋予其魔力的力量宝石(也就是我们传入的参数表), 最后和精心打造的剑身(也就是我们的`{}`的部分). 于是一把威力巨大的宝剑就诞生了.<sup>57</sup>

自古以来, 人们都认为矮人们是最伟大的工匠, 并且是固执重视荣誉, 直至死战到最后一兵一卒的英勇战士.<sup>58</sup> 他们将充满魔力的宝石附到剑上, 念动咒语, 宝剑就开始了魔法的转化, 对于其中的转化, 我们还是不必多说了吧,(因为那可是矮人们的技巧呢):

```

sword = -> (spell) { puts "#{spell.upcase} Sword!" }
spell = "fire"
sword.call spell # => FIRE Sword!

```

<sup>56</sup>其实我们的编程过程也就只是一种用数据来描述过程的想法. 就像是很早之前的纸带计算机, 据说是将程序用打孔器在纸带上按照规律来打孔, 然后传入计算机. 作为程序来运行.

<sup>57</sup>lambda 演算在 1937 年被 Alan Turing 证明和图灵机是等价的计算模型, 也就是一种满足图灵完备的计算模型. 并且 lambda 演算形成了所有函数式编程的基础.

"比起指令式编程, 函数式编程更加强调程序执行的结果而非执行的过程, 倡导利用若干简单的执行单元让计算结果不断渐进, 逐层推导复杂的运算, 而不是设计一个复杂的执行过程." (摘自维基百科)

不过在实用的工业领域, Lambda 表达式更多地会被叫做匿名函数. 因为它们就像是没有名字的函数, 可以不必用 `def` 这样的语句来为这个过程来定义名字就可以直接使用, 比如`-> { puts "Hello" }.call # => Hello`. 并且还能够将其保存到变量里面, 在日后再调用.

<sup>58</sup>来自《魔戒》中矮人的形象, 注: 根据维基百科, 矮人们贪恋金银财宝, 于是导致了因为诱惑而杀害了精灵王而与精灵结怨的悲剧. 不过我们都请求他们为我们打造了如此强力的工具了, 还是只介绍他们的优点, 而将这略有遗憾缺点留在脚注的空间里吧.

灵巧的矮人们打造出了一把把宝剑, 除了挥舞宝剑在战场上厮杀以外, 我们还能够看到史诗中的英雄们如何从矮人那得到传说中的宝剑, 最后凭借着这把宝剑取得神圣的胜利. 这个时候, 胜利不再是一个这样那样的过程, 而是被扼在工匠的手中, 由一锤又一锤的敲击下将无数命运塑造成一个实物传递到了英雄的手中. 这个时候, 我们不再是将 Lambda 表达式仅仅看作是一种描述过程的东西, 而是将其看作是一种将过程数据化的东西.<sup>59</sup> 现在我们能够将过程当作数据来传递了. 举个例子, 我们之前例子里的  $\sum_{i=0}^n a_i$  的求和表达式. 如果用 **def** 的方式来定义, 我们可能就要对每一个不同的  $a_i$  表达式来定义不同的求和函数了:

```
def sum_i(start, stop)
  sum = 0
  (start..stop).each do |i|
    sum += i
  end
  return sum
end

def sum_square_i(start, stop)
  sum = 0
  (start..stop).each do |i|
    sum += i ** 2
  end
  return sum
end

# ...
```

尽管上面的求和很方便, 但是在构造的过程中, 出现了许多重复的代码. 毕竟编程很多时候是为了帮助人们减少重复单调的操作, 但是倘若编程本身就成为了单调重复的事情, 那不就变成了舍本逐末的事情了吗? 所以我们可以通过这样的方式来简化 **sum** 的过程:

```
def sum(start, stop, func)
  sum = 0
  (start..stop).each do |i|
    sum += func.call(i)
  end
  return sum
end

# sum of i
sum(1, 5, -> (i) { i }) # => 15
# sum of square i
sum(1, 5, -> (i) { i ** 2 }) # => 55
```

上面的代码, 我们通过把过程用 Lambda 表达式`-> (i) { i }`和`-> (i) { i ** 2 }`来表示成数据, 然后传递给 **sum** 函数, 这样就简化了复杂过程的构造了(因为我们共用了 **sum** 的这段构造过程.)。

---

<sup>59</sup>感觉有点像是代数里面的映射到函数空间的映射. 不过这种我自己都搞不清楚的东西, 还是不要作为例子来介绍了吧.

不过其实往函数里面传入作为数据的过程其实我们早就已经见过了。回想一下，这不就是我们的代码块 block 吗？比如说，假如我们有这样的一个 `sum` 函数，能够像这样来调用：`sum(start, end) { |i| i ** 2 }` 之类的。我们可以像这样来定义这个 `sum` 函数：

```
def sum(start, stop, &block)
  sum = 0
  (start..stop).each do |i|
    sum += yield i
  end
  return sum
end

# sum of i
sum(1, 5) { |i| i }

# sum of i ** 2
sum(1, 5) { |i| i ** 2 }
```

(这部分还需要重新写... )

在上面结构中，我们把

### 1.2.2 用你的名字呼唤我

O Romeo, Romeo, wherefore art thou Romeo?  
 Deny thy father and refuse thy name.  
 Or if thou wilt not, be but sworn my love  
 And I'll no longer be a Capulet.  
 'Tis but thy name that is my enemy:  
 Thou art thyself, though not a Montague.  
 What's Montague? It is nor hand nor foot  
 Nor arm nor face nor any other part  
 Belonging to a man. O be some other name.  
 What's in a name? That which we call a rose  
 By any other name would smell as sweet;  
 So Romeo would, were he not Romeo call'd,  
 Retain that dear perfection which he owes  
 Without that title. Romeo, doff thy name,  
 And for that name, which is no part of thee,  
 Take all myself.

*from Romeo and Juliet, spoken by Juliet*

但是我不觉得名字不是很重要，每一个优秀的巫师都知道真名<sup>60</sup>的重要性。罗密欧抛弃了他的“名字”，但是他的自身并不会被因此抛弃。<sup>61</sup> 倘若连自身的值 (ego) 都给扔了，那么等待他的就只有被 GC(垃圾回收机制) 给消除的命运了。

乐，还是不要在意这些东西了，让我们给他一个新的名字吧。

<sup>60</sup>乐，其实这个的真名更像是地址。在我们熟悉的递上苹果的蛇的后裔之一 – Python 中，变量储存的就是值的地址的引用。

<sup>61</sup>这个我们可以用 Python 的变量来看看，`name = "Romeo"; new_name = name; id(name) == id(new_name) ## => True`，这个时候我们就会看见，哪怕变量的名字不同，只要它们指向的东西是一样的，那么它们的 `identify(id)` 就都是一样的。呃，这个有点没意思，因为 Python 的变量还是和 C 的指针有点区别，这里看不懂没关系，就跳过吧。

```

name = "Romeo"
price_of_the_sword = 1_000_000
the_name_of_my_kitty = "Lucky"

```

上面就是一串的命名的过程. 我们通过形式化的一个名字 = 值的一个表达式来告诉 Ruby, 我们要给这些值起一个名字, 其中值就是等号右边的内容, 名字就是等号左边的东西. 这样的表达式我们叫做赋值表达式.

在地海传奇里面, "当知晓了某物/人之真名时, 便能成为其主人."; 我们没必要使用那么大的杀器, 我只想在呼唤 kitty 的时候它能够回应一下.<sup>62</sup> 那么在我们 Ruby 里面, 每一次我们喊出变量的名字的时候, Ruby 就会帮我们找到那个变量名字对应的值, 然后拎起 kitty 的后颈, 把它轻轻地提到我们的面前.

```

print "Where are you? ", the_name_of_my_kitty, " Come here! \n"
print "Oh, yes! Here you are, my ", the_name_of_my_kitty
# =>
# Where are you? Lucky Come here!
# Oh, yes! Here you are, my Lucky

```

(在上面的代码里面, 我们用 # 号来表示注释, 所有在 # 后面的东西都不会被 Ruby 看到, 所以我们用它来做一些小笔记. 就像不能被老师看到的小抄一样.)

并且就像我们给东西取名字的时候, 要有一定的规则, 不能随便乱起名字, Ruby 里面, 给变量取名字也是一种小有学问的东西呢. 比如说, 我们不能够把名字取得让人混淆: `2333_is_not_a_variable`, `"name"_is_also_not_a_variable`, `:p_not_variable_name`, 这样的名字会让 Ruby 在阅读的过程中内心发出一堆的怀疑. 因为在 Ruby 看来, 这样的名字更像是数字, 字符串, 符号而不是变量名字, 但是后面跟着的东西有不像是数字, 字符串, 符号之类的东西, 这可真是让人头大. 所以 Ruby 就会非常生气地说我不干啦, 你好像出错了!<sup>63</sup> 所以我们起名字的时候, 还是老老实实地取一个不是那么古灵精怪的名字吧. 比如说`a_good_name`, `the_name`, 之类的名字吧.<sup>64</sup>

不知道你有没有发现, 很多专有名词名词在英文里面都是用大写字母开头的, 比如 **China, English, German** 之类的. 没错, 在 Ruby 里面, 我们也用大写开头的"变量"名字来表示一种"专有"的特色. 就像"一个中国"是不可动摇的原则一样, 用大写开头的"变量"名字也就像是在捍卫自己不可更改的坚定立场. 所以我们叫这样的量为常量. 不过 Ruby 也不是那种死脑筋, 假如你想要修改一个常量, 也不是不行. 因为这个是科学, 科学允许质疑, 否则你得到的只有教条.<sup>65</sup> 只是 Ruby 会给你一个提示: "warning: already initialized constant", "(irb):line: warning: previous definition of Constant\_Name was here". 不过你还是最后能够修改常数的值啦.

于是, 当我们想要把一个值保存起来, 然后以后再调用的话, 就可以利用赋值语句来实现我们的目的了. 在这个过程中, Ruby 会将这些名字保存在自己的一个小册子里面, 然后每次见到一个看起来像是名字的时候, 就会去这个小册子里面翻找 – 对了, 为什么不先看看这个神奇的小册子呢?

(在 irb 中: )

```

local_variables
# => [:the_name_of_my_kitty, :price_of_the_sword, :name, :_, :version, :str]

```

<sup>62</sup>但是我并没有小猫, 虽然我喜欢他们.

<sup>63</sup>这个时候会报错: **SyntaxError**, (以 `2333_is_not_a_variable` 为例, 其他的你可以试试看) 如果我们来尝试阅读一下报错的信息的话, 就会发现 Ruby 看到开头的数字就会认为是数字, 而看到紧跟的奇怪东西, 就会感到困惑, 最后爆出错误:

`SyntaxError (irb):*: trailing '_' in number`  
`(irb):*: syntax error, unexpected local variable or method, expecting end-of-input`

<sup>64</sup>虽然数字之类的也是可以用在变量名字里面的, 比如`var_233`之类的, 但是怎么说呢, 看到这种带数字的名字, 我常常会感到十分的头疼. `v1, v2, v3`这样的东西写起来和看起来都像是在糊弄, 所以我不在正文里面介绍, 也不希望你学习, 因为这样的代码真的很难看.

<sup>65</sup>比如说关于电子的荷质比的测量. 密立根油滴实验做的事情就比较坑爹了. 不仅卖了自己的学生, 还做了筛选数据的操作(让数据变得符合他的喜好). 导致了因为自己的诺贝尔奖, 把这个重要的科学常数的精确值的测量推迟了许久. 但是科学厉害之处就在于它是可以质疑, 不断修改精进的.

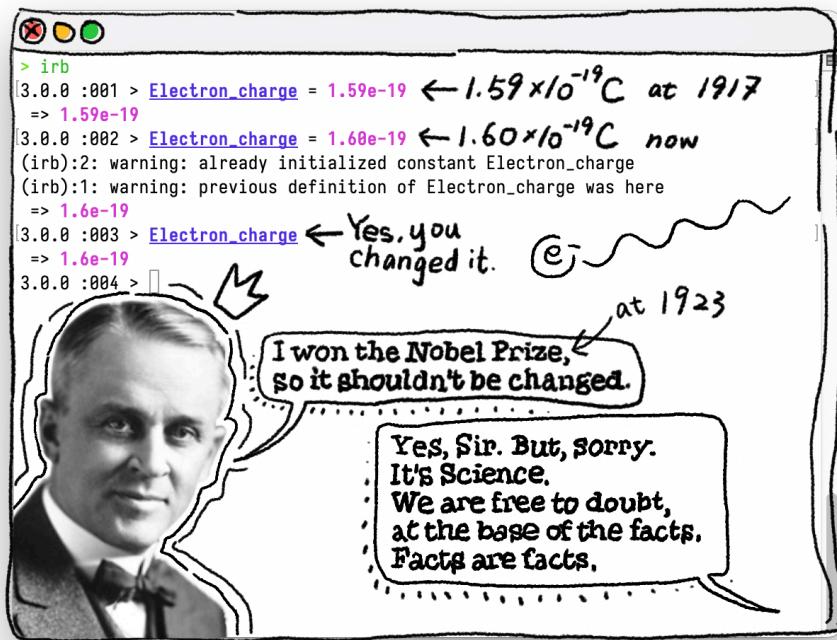


图 1.10: 常量也是可以改的

可以看到, 在 `local_variables` 这个名字<sup>66</sup>里面, 储存了我们的变量的名字。<sup>67</sup> 但是假如我们访问了一个不存在于这个小册子里面的变量的时候, 就好像有人突然在你面前提到了一个你从来没有听过的名字, 这个时候, Ruby 就会正直地提示你:

```
an_undefined_name
# =>
# Traceback (most recent call last):
#   4: from irb:*:in `<main>'
#   3: from irb:*:in `load'
#   2: from irb:*:in `<top (required)>'
#   1: from (irb):*:in `<main>'

# NameError (undefined local variable or method `an_undefined_name'
# for main:Object)
an_undefined_name = "Hey, Ruby, this variable is such a thing. "
an_undefined_name
# => "Hey, Ruby, this variable is such a thing. "
local_variables
# => [:an_undefined_name, :x, :the_name_of_my_kitty,
# :price_of_the_sword, :name, :_, :version, :str]
```

可以看到, 这里 Ruby 在最后告诉你出现了一个 `NameError`, 也就是对你的这个名字, Ruby 会说, 嗯, 这个名

<sup>66</sup>其实是一个方法的名字

<sup>67</sup>这里是用 `Symbol` 类型来记录的。又, 这里是一个无关紧要的小注释, 为什么要用 `Symbol` 而不是 `String` 呢? 这是因为相比容易可以发生变化的 `String`, `Symbol` 不太容易发生变化, 并且在比较和调用的时候比较快。也就是说, 当你想要标记但是又不是很想打印的时候, 使用 `Symbol` 就是一个比较好的选择。

字好像我没有见过. 当我们在 Ruby 里面定义了这个名字之后, 再次调用这个名字的时候, Ruby 就知道了该拿什么了.

众所周知, 情人眼里出西施. 罗密欧的眼里的" 西施" 是朱丽叶, 但是在其他人眼中的" 西施" 可就一定是朱丽叶啦. 这就像是" 一千个人眼中有一千个哈姆雷特" 一样, 在不同的区域里面的变量可能会有不同的意义.

在 Ruby 里面, 我们就有类似的" 区域" 的概念. 比如说我们在前面遇到过的代码块结构 (**do...end**), 或者是我们见到过的定义方法的结构 (**def name ... end**)<sup>68</sup>等等. 在这里, 我只会简单地介绍一下这个" 区域" 的概念, 至于更加复杂的问题, 恕我先搁置一边啦.

不知道你有没有玩过显微镜(其实望远镜也是一样的, 只是我没有玩过望远镜), 当我们选择了一个镜头看到了一堆 - 比方说洋葱表皮细胞吧 - 时, 你可能会感慨, 哇哦, 好厉害的细胞, 好漂亮的玩意儿! 然后, 为了更好地看清楚这些细胞, 你换上了一个更大放大倍数的物镜(嗯, 轻轻地旋转你的转换器, 不要让镜头压到你的载玻片), 这个时候, 你的视野聚焦在了一个更加精细的区域, 原来的密密麻麻的细胞们全部都从镜头的中心离开了, 只留下一个主角.

Ruby 中的" 区域" 就像是这样的东西. 这也就是为什么它的正式名字叫做 **Scope**(显微镜的英文名字叫做 microscope). 那么我们现在来一点点例子吧:

(在一个新的 irb 进程里面: )

```
a_variable = "This is a variable"
another_variable = "Oh, This is an other variable"
def test_method
  a_variable_inside_method = "This variable is inside method"
  another_inside_variable = "Inside variable are not available outside"
  print local_variables
end
test_method
# => [:a_variable_inside_method, :another_inside_variable]
local_variables
# => [:another_variable, :a_variable, :_, :version, :str]
```

上面我们就会发现, 在方法 **test\_method** 中的变量不是在外面的变量, 在外面的变量也不能够在方法中调用, 就好像是我们将显微镜的镜头对准了一个新的区域, 观察完后回到了原来的过程中. 于是在观察这个新的区域中时, 我们没法知道原来的区域里面有什么, 回到了原来的区域中后, 我们也不知道新的区域里面有啥. 这个时候, 我们就这样可以分别储存变量的区域叫做命名空间 (NameSpace), 因为在里面我们可以自行命名东西而不用顾及其他命名空间.

再来一些例子:

```
x = 2333
def print_x
  print x
end
print_x
# =>
# Traceback (most recent call last):
#   5: from irb:*:in `<main>'
#   4: from irb:*:in `load'
#   3: from irb:*:in `<top (required)>'
```

<sup>68</sup>以及在之后我们会遇到的 Module, Class 等的东西

```
#      2: from (irb):*:in `<main>'
#      1: from (irb):*:in `print_x'
# NameError (undefined local variable or method `x' for main:Object)
```

上面的例子体现了不同的命名空间的名字是不通用的, 其实这个原因是因为在 `print_x` 方法中的 `local_variables` 里面, 没有 `x` 的名字. (僕わ知らない、君の名前, 乐)

(呃, 上面那一部分写的不是很好, 所以之后再想办法优化, 现在不想写了.)

但是这个时候就出现了一个问题, 假如一个意气风发的家伙狂妄自大地高喊, 啊, 我的话语是真理, 所有的人眼中的真理都该是我所说的话语, 我即是自由与民主... 那么这个人一定也会很反感"一千个人眼中有一千个哈姆雷特" 的说法. 但是这个家伙太精明了, 他用金钱武装起自己的话语, 在全球分发传播自己的话术, 遇到阻碍就用金钱在前方开道, 于是他的话术就传播到了各个角落, 成为了" 真理". 不过 Ruby 没有这么黑暗和腐败, 只是作为一种戏谑, 我们定义了一种类型的变量, 它们在命名的时候用美元符号 (也就是金钱) 开头, 然后不论是在那个角落, 这样的变量都是一样的.

举个例子:

```
$what_mr_k_said = "I'll make world great again! "
$what_mr_k_did = "I just made the world worse first"
def people_in_area_A
  print "This is what Mr.K said: ", $what_mr_k_said, "\n"
  print "This is what Mr.K did: ", $what_mr_k_did, "\n"
end
def people_in_area_B
  print " 据说 K 先生说: ", $what_mr_k_said, "\n"
  print " 他做了: ", $what_mr_k_did, "\n"
end

people_in_area_A
# => This is what Mr.K said: I'll make world great again!
#      This is what Mr.K did: I just made the world worse first
people_in_area_B
# => 据说 K 先生说: I'll make world great again!
#      他做了: I just made the world worse first
```

果然, 用金钱开道的话, 所有的区域的人都知道这位意气风发的先生所说的东西了.

欸, 我们还是抛开这种低俗的打趣笑话吧, 我认为这样的东西没有什么意思. 除了让人哈哈大笑之外, 完全没有任何的营养. 还是回到竹林或者书斋之中, 培养一颗宁静的心吧.

人皆有不忍人之心. 先王有不忍人之心, 斯有不忍人之政矣; 以不忍人之心, 行不忍人之政, 治天下可运之掌上.

《孟子·公孙丑上》

假如孟子会一点点编程的话, 那么他一定会想说:

```
class Human
  @@common_character = :kind_heart
  def initialize(name)
    @name = name
```

```
end
end
```

那么对每个 `Human.new(" 张三")` 以及 `Human.new(" 李四")`, 那么在上面的代码中, 所有的 **Human** 类的对象都共同拥有一个变量名字叫做 **common\_character**, 也就是每个人都应该有的共同的品质都是一颗不忍人之心. 但是每个 **Human** 类的对象, 也就是每个人可以有不同的 **name**(名字).

可能你觉得有点奇怪, 这两个变量和我们之前的变量有什么区别呢? 不就是变量名字前面多了一些奇怪的前缀 - 没错, 就只是这样, 在形式上的前缀标志就表示了变量的" 类型".<sup>69</sup>

你可以这样想:<sup>70</sup>

前面的全局变量 (**global variable**), 因为它们前面拿着一个美刀符号 (\$), 于是所向披靡, 在各个地方都是适用的.<sup>71</sup>

对于前面带了一个艾特号 (@) 的实例变量 (**instance variable**), 你不妨把这个符号想象成一个小小的回形针, 轻轻地别在我们的实例对象 (**instance object**)<sup>72</sup> 上, 对象走到哪, 就带着这个实例变量来到哪里, 也就是说, 实例变量的作用区域就是我们的这个实例对象.<sup>73</sup>

而前面有两个 @@ 符号的类变量 (**class variable**), 除了和实例对象相连以外, 还和类 (**class**)<sup>74</sup> 相连. 所以只要是同一个类的实例对象, 那么它们就都共同分享着同样的类变量.<sup>75</sup>

对于那些普普通通的变量 (**local variable**), 没有任何的前缀, 只能够在一个小小的区域里面随风飘扬. 不过对于代码块之类在局部查找变量的东西, Ruby 还是能够找到它们的. 然而一旦你换到了一些新的区域的时候, 我们就不知道该怎么找到他们了.

这个并不是 Ruby 真正的实现方法, 但是假如我们想要实现一个类似的储存变量的方法的话. 这里提供一个简单的想法: 假如我们学过一点点物理的话, 我们就会想到, 嘿, 所有的参考系都应该是平等的, 好像没有什么" 绝对" 参考系吧. 所以要确定一个物体的位置, 我们总是要去寻求相对位置来找到. 同样的, 我们可以利用相对位置来实现变量的引用, 比如假如我们的区域是一张表格:

当我们想要访问一个变量的时候, 我们就会在当前所在的命名空间里面去寻找, 除非是看到了用 \$ 号开头的全局变量, 才会去约定的地方去寻找. 而我们的实例变量则是相对着实例对象来找的.<sup>76</sup> 而类变量则相对着类来寻找. 这里我们就体现了一种寻找变量值的想法啦.<sup>77</sup> 这里的寻找变量的方式就是一种相对位置的想法啦.

那么总结一下

嗯, 我们关于赋值的问题就介绍到这里. 接下来, 让我们抛开赋值的这些乱七八糟的约定. 可以发现, 我们的赋值的操作让我们能够将值和(变量)名字联系在一起, 只需要呼唤名字, 就能够调用原来的值. 或者是更新, 替换, 修改原来的值. 比如:

```
string = "I'm the King of the "
string = string + "Nuts"
string
```

<sup>69</sup>这个类型不是像 `String`, `Integer` 之类的值的类型, 而是全局变量, 类变量, 实例变量, 局部变量等等的变量类型.

<sup>70</sup>我在这里只会简单定性地介绍, 暂时不会深究这些变量的具体实现. 建议你先是知道有这么个东西, 然后在之后我们会从一些比较基本的方向来介绍.

<sup>71</sup>和 `local_variables` 的方法一样, 我们可以通过 `global_variables` 这个方法来得到我们进程中的全局变量. 比如说在一个全新的 `irb` 里面, `global_variables` 方法的返回值为 `[:$VERBOSE, :$-v, :$-w, :$-W, :$DEBUG, ...]`

<sup>72</sup>等等, 你是不是想问我什么是实例对象? 其实这个就是我们之前一直说的对象啦. 只不过实例对象的这个名字更加注重强调对象是某个类的对象. 这个是一个叫做面向对象编程的思想的概念, 不过我建议您还是

<sup>73</sup>同样的, 你也可以 `instance_variables` 来访问当前的对象的实例变量. 比如 `Human.new("Lucky").instance_variables` 就会返回 `[@name]` 的结果.

<sup>74</sup>类其实也是一个对象, 在 Ruby 里面.

<sup>75</sup>没错, 你没猜错我们还是能够用简简单单的一个 `method` 来访问类的变量, 就决定是你啦: `Human.class_variables`, 这条命令的返回结果就是 `[@common_character]`.

<sup>76</sup>这也就是为什么你会常常看到 Ruby 里面有类似于有 `object.instance_variable_name` 的玩意啦.

<sup>77</sup>具体的话, 我们会在之后再进行一个介绍, 如果可以的话, 我们没准还能实现一个呢.

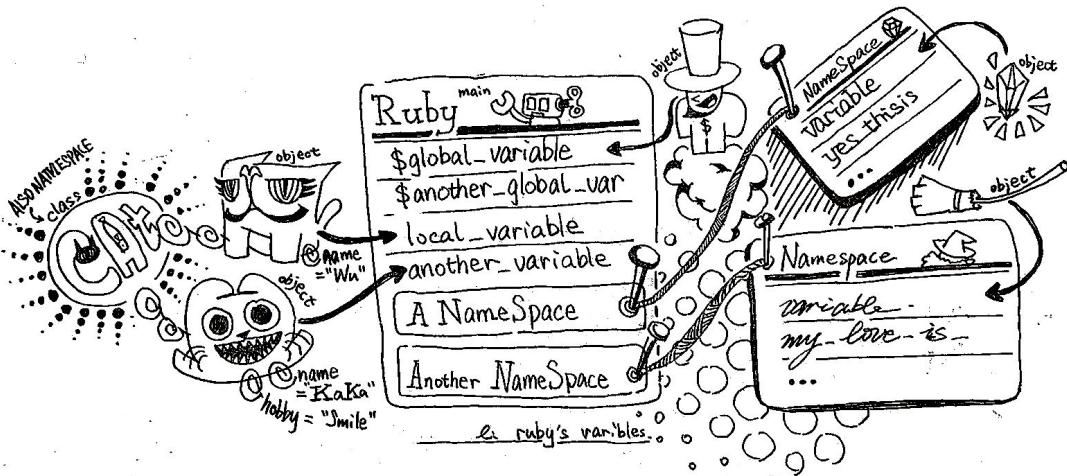


图 1.11: Ruby 的变量们

```

# => "I'm the King of the Nuts"

string.upcase
# => "I'M THE KING OF THE NUTS"
string.downcase
# => "i'm the king of the nuts"

my_money = 100
price_of_cookie = 10
my_money -= price_of_cookie
my_money
# => 90

```

这样我们就得到了存放数值的能力，并且也可以用一个名字来抽象地调用这个值，从而实现更加强大的功能。可惜我比较笨，这里就写一个很普通的利用存放数值的特性交换的例子来结束我们的这一小节吧：

想象一下，一个莫得感情的杀手，为爱杀了人，从此浪迹天涯，流落他乡。一个小女孩委托人，泥沟里翻滚的父母将她的命运拖入了深渊，渴求着复仇的她委托杀手替她报仇。看着小女孩，杀手的心松动了，警醒让他明白这样的后果，人性让他走向了那没有回头路的灭亡之路。所以他把自己的积蓄留在了线人那里。他知道自己终将失去，只是希望至少能够让那个小女孩能够拥有自由的生活。于是他用自己的至宝与小女孩的一无所有交换，

```

leon = "Leon's Treasure"
mathilda_lando = "Nothing"

old_tony = leon
leon = mathilda_lando
mathilda_lando = old_tony

```

上面的代码我们就利用了一个存放数据的概念来做了一个交换，一开始，两个变量手中有着各自的值。Léon 将自己的值交给了中间人 Old Tony，然后拿起了 Mathilda Lando 的值。而 Mathilda Lando 则从中间人那里拿到了

Léon 曾经的值. 于是交换过程结束. 我们利用了一个中间变量来实现了两个变量之间值的转换.<sup>78</sup>

## 1.3 理论法师也要会近战暴击

但甘道夫快步踏上台阶，他盛怒而来，犹如一道白光陡然照进黑暗之地，那些人急忙遮着自己的眼睛往后退开。他抬手就是一击，德内梭尔的剑应声脱手而飞。

《魔戒 III- 王者归来》<sup>79</sup>

作为一名称职的法师，除了法攻之外，物理攻击也是不能缺少的。而作为一名和计算机无关的外行人，除了会一点点理论知识，实操也是不能缺少的。而所谓实操，也并不一定是要写个一万行代码<sup>80</sup>之类的。

在后面的部分，我要来介绍一点点我用过的在编写代码的过程的时候遵守的一些规范和用过的一些工具。虽然只有一点点，因为其实我编写的代码数量真的不太够。

### 1.3.1 六芒星的内角是 $60^\circ$ , 不是 $36^\circ$

神也是人啊！

马猴烧酒《标题还没想好·阿姆斯特朗回旋加速喷气式阿姆斯特朗炮使用手册》

---

<sup>78</sup> 虽然我们还有更多的转换方法，其中 Ruby 提供了一个非常酷的方法：`leon, mathilda_lando = mathilda_lando, leon`，不过这个代码的背后也有中间变量。不过是 Ruby 自动为我们生成的。为了理解这个，我们还有一些路要走（因为我不打算接下去将数值类型），所以放在脚注里面。

<sup>79</sup> 摘自 [甘道夫为什么战斗时不用法术](#)，我对魔戒系列读的不多，大概只看过第一部电影。之后会尝试读完这个系列的。

<sup>80</sup> 虽然这样也确实很重要，我的离散数学老师说，搞计算机的不写个几十万代码都不好意思说自己是搞计算机的。但是好像也不是那么重要，我们的物理班主任说，学好了统计力学，对深度学习那套理论就比只会调包的写代码的强了。说到底，这个就是一个实践和理论的矛盾的区别了。不过这个有什么好争论的呢？小孩子才做选择，成年人应该两个都要。

## 第 2 章 Break Things Down

经过反复的体验, 我认识到: 57 年前悬崖上的那一次使人记忆深刻的教训, 使我在之后的生活中能够正确对待诸如看得太远, 想得太多, 瞻前顾后, 灰心丧气等不利心理因素. 我时常告诫自己: 不要看那些离得很远的岩石, 只管迈出第一步; 站稳脚跟之后, 再迈另一步, 直至达到预期的目标. 这时, 再回头来看我所走过的路程, 我就会为自己所取得的成就感到惊讶和自豪.

莫顿·亨特《悬崖上的一课》<sup>1</sup>

好的, 同学们, 让我们把课本翻到, 呢, 随便哪一页吧. 读一页是一页. 就像是如何击败葫芦娃一样, 千万别一口气打七个, 把那座大山锯开逐个击破就好了.

### 2.1

---

<sup>1</sup>这段话摘自小学课本《走一步再走一步》的原文. 不过都过了这么久了, 我也真的不记得这段课文到底是在哪里出现过了. 笑.

## 附录 A Environment Setting Up

虽然我只能说设置环境实际上是非常枯燥和无聊的一个东西。感觉就像是在和魔鬼做生意一样：魔鬼们说，孩子，搞电脑是多麻烦啊，自己搞电脑是多么丑啊，让我们来帮你美化吧，让我们来帮你设置吧，让我们来帮你...于是我们剩下的就只有接受既得的电脑的操作方式，被迫去适应电脑的逻辑和思路，去享受便利和美观。<sup>1</sup>

所以为了重新得到一切的控制权，我们就要不得不重新拾起麻烦的事情，不过还好，我们不必真的像是最初的程序员一样，需要从零开始，我们要做的，只是从体制化的舒适监狱里面稍微打开一个小洞，就可以利用前人已经挖得差不多的地洞来到自由的世界。哪怕可能在路上会遇到一些不识抬举的路障，但是毕竟是为了自由嘛...

我们的镰刀和锤子就是简单的命令行工具了。

### Terminal

终端，是个只用命令就能够操作电脑的一个电脑的交互式界面。<sup>2</sup> 可以在终端里面输入命令，然后终端就会执行相应的命令，然后得到结果，或者是达到目的。

对终端命令的理解其实非常容易，比如下面这个终端的内容的截取：

```
> mkdir gift
> cd gift
> touch gift.txt
> echo "I'm Lucky. " >> gift.txt
> cat gift.txt
I'm Lucky.
> ls
gift.txt
> ls -l
total 8
-rw-r--r-- 1 liiyiyang staff 12 Mar 11 20:32 gift.txt
```

这里暂时不对上面的命令做解释，但是我们可以简单的观察命令的格式，可以发现，命令的大概模式是这样的：命令名称 + 命令参数。

嗯，有点抽象。那么不妨这样想：你应该见过村妇骂街吧？村东头的王大婶撂下一盆衣服，扯开嗓子大吼：“村西李大姐！来捡你家毛孩子！”这个时候我们可以这样想，王大婶往命令行里面输入了一个命令（或者说程序名称）：村西李大姐，然后又输入了一个参数：毛孩子，这样，系统就会找到“村西的李大姐”这个程序，然后将“毛孩子”这个参数丢给“李大姐”来处理。至于毛孩子的下场如何，就暂时和我们没关系了。

其实我们刚才干的就是一个人为的对计算机终端命令的一个解析的过程。这个过程和真实的计算机的 parser 大概是差不多的。虽然不同的命令对于参数的输入形式是不一样的，就好像李大姐见不得别人喊她儿子为“毛孩子”，除非你叫她儿子为“寿限无寿限无扔屁机前天的小新的内裤新八的人生巴鲁蒙格·费扎利昂爱扎克·休纳德三分之一的纯情之感情的剩下的三分之二是在意倒刺的感情背叛好像知道我的名字我知道他不知道的不在家干鱿鱼鳞鱼干青鱼子粪坑鳞鱼”，并且还要加上修饰符-f，否则她绝对不会反应。于是我们只需要了解有哪些命令，并且它们分别对应的参数是什么形式就好了。

听起来挺麻烦的。因为 Linux 的命令有几百条，但是实际上，常用的估计就不到 5 条，并且我们可以让程序告诉我们它是怎么使用的。

<sup>1</sup>关于这个我记得锤子的设计师罗子雄吐槽过这个：iOS 现在的设计抛弃了原来的拟物风的直观特性，小孩子拿到就会使用到让用户去不得不增加学习成本，去适应新的操作逻辑。

<sup>2</sup>注：我已经默认将 Windows 的 CMD 和 Powershell 自动排除了讨论范围，因为我觉得这两个实在是不太上了台面，假如你是 Windows 用户的话，尝试看看 WSL 吧。

## 网络问题

呃,说实话,有时候我觉得网络上提供的教程不太适用的一个原因是: 我的网络链接有亿点问题,总是因为访问不了某些网站而被迫曲线救国.

不过这个的话就只能各显神通了,我不能多说了.

## A.1 macOS

我的电脑是 m1 的 macOS, 虽然不得不说, 苹果的 m1 真溜, 但是苹果也是个老典狱长了. 不过好在出于历史原因, 苹果还留着一点点类 Unix 的味道. 至少不是无路可走的地步.

### A.1.1 Homebrew

虽然苹果的 Terminal 终端和 Windows 的 CMD 以及 Powershell 比起来真是好用太多了.<sup>3</sup> 但是总是缺了一点东西: 比如一个像主流 Linux 发行版一样好用的包管理器, 或者是(用一个比较离谱的比喻来说就是) 应用市场. 想要使用别人写好的命令行程序的话, 没有包管理器, 就要自己解决依赖, 编译等的问题的话实在是有点麻烦.

所以, 如果安装了一个 macOS 的包管理器的话, 以后的生活就会变得更加美好.

(假设你已经解决了网络的问题)

那么只需要按照官网提供的脚本在终端里面运行就好了:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

当你安装好了以后, 可以使用以下命令来检查 homebrew 的版本来检测是否安装到位:

```
> brew -v
Homebrew 3.3.16
```

假如你想要找什么软件包的话, 可以到[官网](#)上去找.

并且官网上的教程写得很详细, 我就不卖丑了, 就介绍两个命令:

```
> brew install pkg
> brew uninstall pkg
```

分别是安装和卸载叫做 pkg 名字的软件包.

### A.1.2 Ruby

虽然 macOS 自带了 ruby, 但是这个 ruby 用起来不仅不够舒服, 还动不动就会报错, 据说这是因为 macOS 的 ruby 删除了很多的特性, 不过假如你已经安装好了 homebrew, 你可以直接:

```
> brew install ruby
```

这样就可以安装 ruby 了.<sup>4</sup>

### A.1.3 NeoVim

我们都知道 Vim 是一个很厉害的文本编辑器, 然后 Emacs 也是. 不过我用的是 NeoVim, 因为它的 Coc-nvim 插件比较友好, 还有文本补全.

安装方法还是使用 homebrew:

<sup>3</sup>参见[漫画](#)

<sup>4</sup>假如你想安装不同版本的 ruby, 建议安装[rvm](#)

```
> brew install neovim
```

然后在命令行里面输入

```
> nvim filename
```

就可以用 Neovim 来编辑文件名字叫做filename 的文件内容了. 具体的内容之后会写.

## 附录 B A Quick (and Hopefully Painless) Ride Through Ruby (with Cartoon Foxes)



图 B.1: 出现了, 小狐狸们

哈, 没错, 就是这两小只. 糟了, 我的哮喘又发作了, 所以我现在要去拿我的气雾剂了, 一会儿.



图 B.2: foxes in boxes

这章的故事里面有很多的故事没准会让你潸然泪下.<sup>1</sup>

别废话了, 我们现在就快速地看看 Ruby 的大致的特性吧. 就好像是高速吟唱 ABC 字母歌一样.<sup>2</sup>

### B.1 语言以及我所指的语言

我的良心不允许我将 Ruby 叫做一种计算机语言. 这样的话就听起来像是 Ruby 原来只是用于计算机专业的用途, 并且从头到尾就只是设计出来为了对电脑指手画脚发送指令的, 让我们这些 coders 看起来不过就是些在计算机领地里面要求公民身份的外国人一样. 这样的话, 我们岂不就只是个会计算机语言翻译机器了吗?

但是如果你的大脑都开始使用一个语言进行思考, 你开始使用这个语言中的词汇和口语来表达你的思想, 你还会叫这个语言仅仅是计算机语言吗? 想想看吧, 计算机可不会做这样的事. 所以你又怎么能够将 Ruby 叫做是计算机语言呢? 它就是我们自然表达的语言啊.

<sup>1</sup>原句我猜测应该是利用了双关: I'm told that this chapter is best accompanied by a rag. Something you can mop your face with as the sweat pours off your face. 其中 rag 有恶作剧和抹布的两种意思. 但是根据书的标题, 感觉这样翻译比较好一点. 这里借鉴了中文的翻译版本

<sup>2</sup>原句是: Like striking every match in a box as quickly as can be done. (就像是尽可能快地划亮火柴盒里面的所有火柴一样.) 我认为不够能够表达接下来的内容.

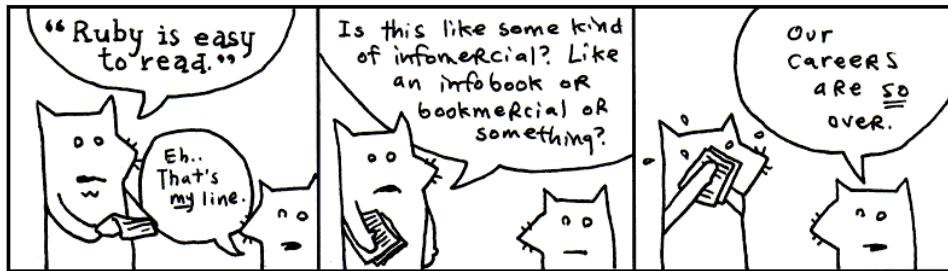


图 B.3: 我们这两只狐狸朋友终于明白了他们一无是处的窘境了

我们可不能再说 Ruby 只是 计算机语言了. 它是 *coderspeak*, 是我们的思维的语言.

试试看将下面的代码读出来:

```
5.times { print "Odelay!" }
```

在中文里面<sup>3</sup>, 句号, 叹号和括号一样的标点符号是无声的, 它们会给单词赋予意义, 帮助读者们在字里行间理解作者想要传递的意义. 所以我们来试试读出上面的代码: 五次打印出 "Odelay!" <sup>4</sup>

没错, 你读出的就正好就是这段小小的 Ruby 代码所做的事情. Beck 的 *mutated Spanish* 里的呼吼也将在他的电脑屏幕里面打印五次.<sup>5</sup>

试试看将下面的代码读出来:

```
exit unless "restaurant".include? "aura"
```

在这个问题里面, 我们做了一点点十分基础的检查 (reality check). 我们的程序将会 **exit**(也就是结束运行并退出)**unless**(除非) 我们的单词 **restaurant**(餐厅) 里面包含 (也就是 **includes**) 单词 **aura**. 这一次用中文来说就是: 退出, 除非餐厅 (**restaurant**) 里面有气氛 (**aura**).<sup>6</sup>

你可曾见过像这样方便地使用问号的编程语言呢? Ruby 使用像感叹号和问号之类的标点符号来增加代码的可读性. 既然我们在上面写的是一个问句, 那为何不让它变得更加清晰易读呢?

试试看将下面的代码读出来:

```
['toast', 'cheese', 'wine'].each { |food| print food.capitalize }
```

尽管这个看起来不是有点难读, 并且和前面的例子相比, 它不是很像一个句子. 但是我还是会鼓励你将它读出来. 因为 Ruby 代码有时候可能读起来像是英语, 它有时候也会读起来像是被缩写了英语, 所以你只要把它稍微转换一下, 就可以这样子: *With the words 'toast', 'cheese', and 'wine': take each food and print it capitalized.* (我有一堆单词 (食物)'toast', 'cheese' 和 'wine', 分别从里面拿出每一个食物, 然后将它用大写的形式输出到屏幕上.)

然后计算机将会彬彬有礼地输出 Toast, Cheese 以及 Wine.

在这个时候, 你也许会好奇这些符号到底是如何相互之间联系在一起的. 估计你现在正在为这些点号和括号到底是干什么用的而烦恼.<sup>7</sup> 别担心, 我将在下一节进行一个介绍.

现在你只要知道 Ruby 基本上是由一个个语句组成的. 虽然严格意义上来说, 这些语句并不是语法正确的英文句子, 它们更像是一堆组合在一起传递特定的想法的单词和标点符号. 这些语句可以形成一本书, 一篇文章甚至如果他们组织得当, 能够形成一部小说, 一部不仅能被人能阅读的小说, 也同时能够被计算机所理解.

<sup>3</sup>原文是" 英语"

<sup>4</sup>Five times print "Odelay!" 或者调换一下顺序, 打印五次"Odelay!"

<sup>5</sup>人类的本质就是复读机

<sup>6</sup>又是一个双关.

<sup>7</sup>原文: Smotchkiss is wondering what the dots and brackets mean. 不清楚这个是什么意思

## B.2 语句的组成部分

就像是臭鼬身上的白色条纹和新娘子的白色婚纱一样富有标志性,<sup>8</sup> 许多 Ruby 的语句在形式上就有很多的标志(visual cues)来帮助你来辨认他们. 比如标点和大写可以让你的大脑在看到这一堆代码的一瞬间就认出它们, 就好像是你的潜意识在大喊: 嘿, 我认识这个家伙! 于是你就能够轻松区分(name-drop)这些组成部分并且融入 Rubyists 的交流之中.

接下去我会在形式上给出 Ruby 里面的不同的语句组成部分的介绍, 现在我不会细致地去解释里面的细节(这些会在这本书的其他部分介绍), 并且你也不必费劲去理解这些解释和定义. 你只需要能够在这章结束的时候, 能够在一个 Ruby 程序中认出这些部分就可以了.<sup>9</sup>

好的, 让我们来关注下面的这些语素吧:<sup>10</sup>

### Variables

任何简单的小写单词在 Ruby 里面就是一种 variable(变量). Variable 可以由字母, 数字和下划线组成.

`x, y, banana2` 或者是 `phone_a_quail` 就是一些例子

Variable 就像是绰号. 还记得曾经所有人都叫你臭彼得(Stinky Pete)的时候吗? 只要你一出现, 总会有人大喊: "嘿, 滚出去, 臭彼得!" 然后所有人都奇迹般地知道了"臭彼得"就是你.<sup>11</sup>

使用 variable, 你就像是给自己经常使用的东西取了一个绰号. 举个例子. 假如说你开了一个孤儿院. 这是一个十分抠门的孤儿院. 当 Warbucks 老爹(Daddy Warbucks)来领养孩子的时候, 我们还强迫他再从我们这里买一个一百二十一点零八美分的泰迪熊 - 因为它们是这些孩子们在这噩梦般的孤儿院里面的精神慰藉了.

```
teddy_bear_fee = 121.08
```

然后, 当你要着铃铛叫他前去收银台(那里有一台运行着 Ruby 的性能强大的收银机)去结账, 然后你要将他的所有的开销都加在一起, 记作 total.

```
total = orphan_fee + teddy_bear_fee + gratuity
```

这些变量的绰号实在是帮大忙了.<sup>12</sup> 并且我确信, 在这肮脏的地下儿童交易里面, 任何的小聪明都是值得一试的.

### Numbers

最常见的一种数的类型是 integer, 它们是带着正号或负号的一系列的数字.

`1, 23, 以及 -10000` 就是一些例子

虽然数字里面不能出现逗号, 但是却可以出现下划线. 所以假如你想要将数字每三位进行分割来增加可读性的话, 可以使用下划线.

<sup>8</sup>原文就是这样, Just like the white stripe down a skunk's back and the winding, white train of a bride, why 先生的比喻十分的有趣, 经过翻译感觉就像是吃麻辣烫前先将佐料过一遍白开水, 原本的味道全没了.

<sup>9</sup>这段有点不太好翻译: Try to focus on the look of each of these parts of speech. The rest of the book will detail the specifics. I give short descriptions for each part of speech, but you don't have to understand the explanation. By the end of this chapter, you should be able to recognize every part of a Ruby program.

<sup>10</sup>在这里我用了语素这个该死的语法名称, 只是因为我词汇缺乏而已, 请见谅. 并且在下面的翻译中, 我不会将英文的叫法给翻译了, 因为这个我觉得是一种专有名词, 翻译了总觉得有点怪怪的.

<sup>11</sup>呃, 虽然我也有绰号, 但是却是一个有意思的绰号. 不知道 why 经历了一个怎么样的学校生活, 这个故事太惨了. 后面的例子我就不一一吐槽了, 但是我真的应该是没有乱翻译. 就是, 这些故事有点悲惨.

<sup>12</sup>上面的公式, 假如把里面的英文翻译一下的话就是: 总计 = 孤儿的领养费用 + 泰迪熊的价格 + 小费, 是不是非常好理解, 比起 121.08 这样不明所以的数字起来, 简直就是小学问题了.



图 B.4: 这帮家伙嘲笑我的例子

```
population = 12_000_000_000
```

在 Ruby 里面, 小数 (decimal numbers) 被称为 *floats*(浮点数). Floats 类型的 (也就是小数) 有两种形式: 带小数点的或者是用科学记数法来表示的.

3.14, -808.08以及 $12.043\text{e-}04$ 就是一些例子.

## Strings

Stings(字符串) 是一堆被引号包围的任意字符 (字母啊, 数字啊, 标点符号啊都行). 无论是单引号还是双引号都能够用来形成 Strings.

"sealab", '2021', 或者是 "These cartoons are hilarious!" 就是一些例子.

当你将这些符号字母用引号包在一起之后, 它们就被储存在一个 string 里面.

想象一下这个场景: 一堆名流的晚宴上, 熙熙攘攘的谈话声混在一起, 场面十分的聒噪. 你是一个记者, 想要从这堆噪音中捕捉到名流的话语. "我比你们更加精明," Avril Lavigne 说到, "并且我知道经商的秘诀 – 就是你该做些什么以及怎么做到它." <sup>13</sup>

```
avril_quote = "I'm a lot wiser. Now I know
what the business is like -- what you have
to do and how to work it."
```

就像我们之前把一个数储存在 **teddy\_bear\_fee** variable 里面一样, 现在我们把一组字符 (也就是一个 string) 储存在了 **avril\_quote** variable 里面. 然后你这个记者把这段名言发送给了印刷人员. 这位印刷人员恰好使用的是 Ruby 来进行他们的印刷工作:

```
print oprah_quote
print avril_quote
print ashlee_simpson_debacle
```

## Symbols

Symbols(符号) 看起来就像是变量一样. 它们都包含字母, 数字, 还有下划线. 但是 symbols 是以一个冒号开头的.

:a, :b, 或者 :ponce\_de\_leon 就是一些例子.

<sup>13</sup>原文是英文, 并且我对代码里面的英文保留了, 代码里面的英文可以参考这里的翻译.

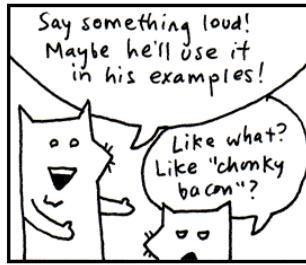


图 B.5: 他们想要帮我想例子

Symbols 相较于 strings 更加的轻便. 假如你想要使用像 string 一样的标记, 但是又不想将它们用来打印的话, 使用 symbol 会比较方便.

或者你可以理解为 symbol 对于电脑来说更加容易处理. symbol 前面的冒号就像是插在椰果上的吸管, 让计算机能够轻松地吸取处理它, 啊, 真是甜蜜而又轻松啊.<sup>14</sup>



图 B.6: Chunky bacon!!(烟熏肉! )

## Constants

Constants(常量) 有点像是 variables, 但是 constant 是用大写字母开头的. 假如 variables 像是 Ruby 里面的名词 (nouns), 那么 constants 就像是专有名词 (proper nouns).

`Time`, `Array`, 或者 `Bunny_Lake_is_Missing` 就是一些例子.

在英语中, 专有名词都是大写字母开头的. 就像是帝国大厦 (Empire State Building) 一般. 你不能够移动帝国大厦. 你也不能定义它为什么其他的东西.<sup>15</sup> 专有名词大概就是像那样的东西. 它们表示的是一些比较特别的东西, 一般人们不想要它随便地变化.

同样的, constant 也不应该在被他们声明了之后被改变.

```
EmpireStateBuilding = "350 5th Avenue, NYC, NY"
```

假如我们尝试去修改这个 constant, Ruby 会皱着眉毛抱怨我们.<sup>16</sup>

<sup>14</sup>原文是: It's like an antacid. The colon indicates the bubbles trickling up from your computer's stomach as it digests the symbol. Ah. Sweet, sweet relief. 大意就是说 symbol 对于计算机来说比较容易处理, 为了保留原来的比较有趣的比喻, 所以我换了一个比喻. 因为翻译不过来. 这里追加一个解释, 因为在计算机里面, 符号和字符串还是有一点点不一样的, 对于字符串, 计算机是对每一个字符进行一个比较后才能说两个字符串是一样的, 而对于符号, 计算机内部是将其转换为一个特殊的数字, 这个数字比较方便比较, 储存和比较的效率比较高.

<sup>15</sup>假如为了地方特色化的话, 可以变成人民大会堂或者是天安门广场一样的东西.

<sup>16</sup>实际上还是可以改的, 毕竟 Ruby 很随意. 假如是别的严格的语言, 就会像是去星巴克点小杯一样的坑爹故事一样了. 因为你不合规矩, 所以就会被骂了. 但是 Ruby 就像是个宽容的同辈的朋友, 会容忍你的任性.

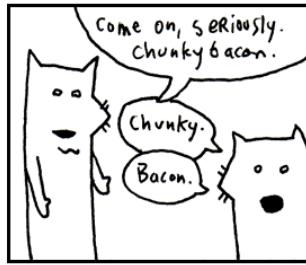


图 B.7: 上啊, chunky bacon

## Methods

假如 variables 和 constants 像是名词, 那么 methods 就像是动词 (verbs). Methods 通常跟在 variable 或 constant 的后面, 用一个点号来分割. 你应该在生活中早就见过类似的东西了:<sup>17</sup>

```
front_door.open
```

在上面的例子里面, method 是 **open**. 在上面的那一行语句里面, 它就像是一个动词, 一个操作. 在某些情况下, 你能够看到多个动词串联在一起使用.

```
front_door.open.close
```

上面的语句里面, 我们控制计算机把前门 (front door) 打开了又立刻关上了.

```
front_door.is_open?
```

上面的语句也同样是一个操作 (action). 我们控制计算机去检查门 (front door) 有没有打开. 本来这个 method 应该叫做 **Door.test\_to\_see\_if\_its\_open**, 但是 **is\_open?** 这个名字更加的简洁和适当. 在 Ruby 里面, 感叹号和问号都可以被用于 method 名字里面.<sup>18</sup>

## Method arguments

一个 method(方法) 可能需要一些信息输入 (method argument) 才能够执行它的操作. 比如说, 假如我们想要让计算机给我们的门涂上一层油漆, 我们就需要告诉它该用什么颜色的油漆.<sup>19</sup>

```
front_door.paint( 3, :red )
```

上面的语句将前门 (front door) 上了三重红色的漆.

想象一下, 在 method 的背后有一个神奇的机械和隐秘的管道 (将我们的参数传递给 method, 或者说计算机来执行). 我们的圆括号就是那个圆圆的, 湿漉漉的管道内壁; 我们的逗号就像是每一个参数 (argument) 长的脚, 这些脚贴在管道上, 因为最后一个参数将它的脚悄悄地藏了起来, 所以你看不见它们.<sup>20</sup>

就像是一台机器里面有许多的管道一样, methods 和参数也是可以被串在一起使用的.<sup>21</sup>

```
front_door.paint( 3, :red ).dry( 30 ).close()
```

<sup>17</sup>原文: Methods are usually attached to the end of variables and constants by a dot. You've already seen methods at work. 嗯, 翻译的感觉差不多.

<sup>18</sup>有助于让 method 的名字更加好懂.

<sup>19</sup>不让你也不想计算机随便给你的门涂上奇怪的颜色吧.

<sup>20</sup>这一段比较难翻译, 原文是: Think of it as an inner tube the method is pulling along, containing its extra instructions. The parentheses form the wet, round edges of the inner tube. The commas are the feet of each argument, sticking over the edge. The last argument has its feet tucked under so they don't show. 大意就是 method 的参数之间用逗号相互隔开, 通过括号传递给 method.

<sup>21</sup>Like a boat pulling many inner tubes, methods with arguments can be chained.

上面的例子里面我们为门涂了三层红漆, 然后干燥了 30 分钟, 然后关上了门. 尽管最后一个 `method` 没有任何的参数, 如果你喜欢的话也可以在后面加上括号. 虽然一般来说, 在机器里面放一条空管道是挺没用的, 所以这对括号常被忽略.

还有一些 `methods`(比如`print`) 属于 `kernel methods` (`kernel` 内核). 这些方法可以在 Ruby 的各个区域被调用. 真是因为它们很普遍 (`common`), 所以你不必加上点号.

```
print "See, no dot."
```

## Class methods

正如上面介绍的 `methods` 一样 (上面的 `methods` 也叫做 `instance methods`), 而 `class methods`(类方法) 虽然也跟在 `variable` 和 `constant` 后面, 但是并不是使用点号分隔, 而是两个冒号.

```
Door::new( :oak )
```

正如上面的例子展示的一样, 一个`new` class method (叫做 `new` 的类方法) 是用来创造一些东西的. 在上面的例子里面, 我们请求 Ruby 为我们制作一个新的 oak(橡树) 门.<sup>22</sup> 当然, Ruby 需要知道如何去做一扇门 – 肯定要有一堆木料, 一堆伐木工人, 以及那些双人锯.



图 B.8: 可真是一堆的 chunky bacon 呢

## Global variables

那些用美元符号开头的 `variable` 就是 `global variable`(全局变量).

`$x, $1, $chunky`, 以及`$CHunKY_bACOn`就是一些例子.

大部分的变量都有着临时或局域的特性.<sup>23</sup> 你的代码的一个个部分就像是一间间小房子. 走进这些小房子里面, 你会发现它们各自有着自己的 `variables`. 比如说在一个房子里面, 你可能会看到代表着 Archie 的 `dad`(爸爸) 的 `variable`, Archie 是一个到处旅行的推销商人, 同时也是一个头骨收藏爱好者. 而在另外一座房子里面, `variable dad` 可能代表的是 Peter, 一个专业驯狮员但同时又深爱着法兰绒制的衣服. 总而言之, 每一座房子里面都有自己独特的意义的 `dad`.

但是有了 `global variable`, 你可以保证它在任何小房子里面都指的是同一个值. 这个美元符号就非常的有灵性. 毕竟是全球通用的令人发狂的符号嘛.<sup>24</sup> 你可以试试随便敲开一家美国家庭的门口, 然后递给他们一沓美金. 我能保证你在打开 Peter, 也就是那个非常喜欢法兰绒衣服的驯兽师的门, 然后递给他一沓现金的时候, 你不会得到和其他人一样的反应.

`Global variable` 可以在程序的各个地方使用. 他们永远不会离开你的视线.

<sup>22</sup>其实写成`Door.new`也是可以创建一个新的实例对象了, 这里就有点让人难懂了, 不过应该不是什么大问题.

<sup>23</sup>原文: Most variables are rather temporary in nature. 这个意思是大部分的变量只能在一定的范围里面生效, 然而这样的范围实际上是临时的, 会随着程序的进行而打开或关闭. 于是变量就会生效或失效, 体现了变量的临时, 或者说局域性的特点.

或者用计算机运行的思路去理解的话, 普通的变量是储存在计算机运行的时候开辟的临时的内存空间里面 (比如栈), 然后全局变量储存在特定的地址里面, 因为临时的空间会关闭和重复利用, 所以放在里面的变量都是类似于一次性的东西. 但是特定的地址除非你故意去修改, 不然一般是不会变的. (个人理解, 不太专业.)

<sup>24</sup>原文是: Every American home respects the value of the dollar. We're crazy for the stuff.

## Instance variables

用艾特符 (@) 开头的 variable 就是 instance variable(实例变量).

`@x, @y, @only_the_chunkiest_cut_of_bacon_I_have_ever_seen`就是一些例子.

这些变量通常被用于定义一个东西的某些特性 (attributes). 举个例子, 你可能会希望告诉 Ruby `front_door`, 也就是前门的特性, 比如说是它的宽度`@width`. 这个属性是储存在`front_door`里面的. Instance variable 可以用来定义在 Ruby 里面一个对象的特性 (characteristics).

你可以吧这个 at(艾特符) 想象为对 attribute(特性) 的标志.

## Class variables

在 variable 前面放两个符号的就是 class variables(类变量).

`@@x, @@y, @@i_will_take_your_chunky_bacon_and_raise_you_two`就是一些例子.<sup>25</sup>

类似于 instance variables, class variables 也是用来定义对象的特性 (attribute) 的. 不过不是仅仅用来定义一个 Ruby 里面独立的对象的特性, class variables 能够定义在 Ruby 一堆互相关联的对象的同样的特性.<sup>26</sup> 假如说 instance variable 只是给一个`front_door` (前门) 设置了属性, 那么 class variable 就像是为所有的属于 `Door`(门) 的一类东西都设置了一个属性.

你可以将前缀的两个 at 号理解为标记所有 (attribute all) 的含义. 并且你可以想象这样的一个场景: 一大群用 Ruby 控制的 AT-ATs<sup>27</sup> 从星球大战中蜂拥而出. 当你改变了一个 class variable, 这个时候你做的不仅是一个小小的改变, 而是对这一个群体整体属性进行了改变.



图 B.9: 哇呜! Chunky bacon 的点子终于用上了.

## Blocks

被一对花括号包裹起来的代码就是一个 block(代码块).

```
2.times {
    print "Yes, I've used chunky bacon in my examples, but never again!"
}
```

有了 block, 你就可以将一串指令打包在一起, 然后它们就能够你的程序里面相互传递.<sup>28</sup> 这一对花括号就像是一对蟹钳, 紧紧地将里面的代码钳住. 所以无论何时你看见了这一对花括号, 请记得把里面的代码当作是一个整体.

<sup>25</sup>这里和上面的例子里面都有和小狐狸的提议相呼应的地方. 可有意思了.

<sup>26</sup>原文: class variables give an attribute to many related objects in Ruby.

<sup>27</sup>来自星球大战电影, 全称是: All Terrain Armored Transport, 也就是步行机.

<sup>28</sup>原文是: With blocks, you can group a set of instructions together so that they can be passed around your program. 可以从中看出运行过程的抽象, 将指令(或者说是函数?)当作了数据来看.

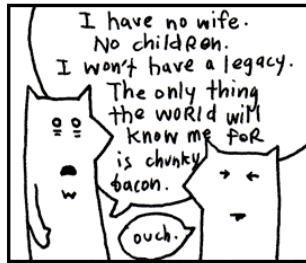


图 B.10: 然后是悲惨的现实

这就有点像是那些在商场里面销售那些印有 Hello Kitty 的文具大礼盒. 透过华丽的透明包装纸可以看见里面塞满了成套的小铅笔和小纸片, 这些漂亮的小玩意儿在光照下闪闪发光, 让你忍不住想要伸手把它们拿起来, 然后在纸上试试. <sup>29</sup> 不过我们的 block 却不需要那么多华丽的包装罢了.

你也可以把那对花括号换成 **do** 和 **end**. 这样在你的代码比较长的时候 (比如多于一行) 就会看起来漂亮一些.

```
loop do
  print "Much better."
  print "Ah. More space!"
  print "My back was killin' me in those crab pincers."
end
```

## Block arguments

Block arguments(代码块参数) 的形式是管道符 (**pipe characters**) <sup>30</sup> 包裹着的一组用逗号分隔的 variable.

`|x|, |x, y|` 以及 `|up, down, all_round|` 就是一些例子

Block arguments 一般用在 block 的开头:

```
{ |x, y| x + y}
```

在上面的例子里面, `|x, y|` 就是输入的参数 (arguments). 我们小小的一片代码就跟在参数的后面: `x + y` 将两个参数加在一起.

我比较倾向于将管道符看作是一个管道 (tunnel) 的象征. 它们看起来就像是滑滑梯一样, variables 从上面滑下来. (`x`展开双臂, 想像雄鹰展翅一般冲了下来, 而`y`则优雅地交叉着腿滑了下来.) 这个滑梯就像是一个连接着周围世界和 block 的一条神奇通道. <sup>31</sup>

Variables 就是通过这个滑滑梯 (或者说是管道) 进入到 block 里面的.

## Ranges

两个用括号包围, 并且用省略号 (虽然大部分情况下, 这个省略号是用两个点来表示的) 分隔的 variable 就是一个 range 的主要形式.

<sup>29</sup>原文: It's like one of those little Hello Kitty boxes they sell at the mall that's stuffed with tiny pencils and microscopic paper, all crammed into a glittery transparent case that can be concealed in your palm for covert stationery operations. 我应该改变了很多的原文的意思, 但是这个长难句实在有点难以分析. (笑) 所以我选择意译.

<sup>30</sup>呃, 其实也可以叫得土一点, 就叫竖杠就行了. 但是这个"竖杠" 在命令行操作里面干的事情确实和管道很像, 就是将参数 (输出) 传送到另外的一个程序里面, 就像一个管道一样. 而在 Ruby 里面, 它干的事情也和参数传递有关. 所以翻译成管道符我觉得比较好.

<sup>31</sup>奇蛋物语里面的游乐场的滑滑梯给了我翻译的灵感. 原文大概应该也是这个意思: I like to think of the pipe characters as representing a tunnel. They give the appearance of a chute that the variables are sliding down. (An x goes down spread eagle, while the y neatly crosses her legs.) This chute acts as a passageway between blocks and the world around them.

- (1..3) 就是一个 range, 代表着从 1 到 3 的数字.
- ('a'..'z') 也是一个 range, 代表着小写的从 a 到 z 的一组字母.

你不妨将它想象成一个被压缩在一起拿着的手风琴. (但是你当然也可以特立独行, 故意拖着不收起来的手风琴在路上走, 绝对会是街上的一道风景. 但是人有时候还是需要一些自我反省的, 所以我建议你还是小心地把手风琴收起来.) 那组括号就是手风琴的两个小把手. 然后那些点号就是合上手风琴的链子 (chain), 让那些褶皱紧紧的合着.

通常, 我们只用两个点号. 但是如果一个 range 里面用了三个点号, 这就表明最后的那个值将会被剔除.

- (1...5) 代表 0 到 4 的数字.

所以当你看到三个点号的时候, 想象一下一个微微打开的手风琴. 恰好让一个小小的音符逃了出来, 对于这最后一个音符, 我们不妨就让它随风而去吧.<sup>32</sup>

## Arrays

一列用逗号分隔并且被方括号包围的东西就是 array(数组).

- [1, 2, 3] 就是一个装着数字的 array.
- ['coat', 'mittens', 'snowboard'] 是一个装着字符串的 range.

你不妨想象这样的一个场景, array 就像是一只毛毛虫被钉在了你的代码里面. 那两个方括号就像是阻止毛毛虫移动的钉子, 这样你就可以知道哪里是它的头, 哪里是它的尾. 然后逗号就像是它的小脚, 在它身体的节与节之间摇摆.

从前有一只毛毛虫长着用逗号做的小腿, 于是他不得不在每一个字面上的间隔上做一次停顿. 为此其他的毛毛虫都非常地尊重他, 所以他变得非常有威严. 噢, 还有一个喜欢给那些不幸的小虫子送新鲜树叶的臭名昭著的慈善家.<sup>33</sup>

是的, 一个 array 就是一堆东西的集合, 但是 array 同时也会将自己里面的东西按照一定的顺序进行储存.

## Hash

一个 hash(哈希) 就像是一本被花括号 (curly braces) 包在一起的字典. 没错, 就像是字典一样 – 字典将单词和他们的定义配对 – Ruby 里面用一个箭头 (=>) 来将两个东西配对起来. 这个箭头, 正如你所见, 就是一个等号和一个大于号组成的.

{'a' => 'aardvark', 'b' => 'badger'} 就是一个例子.

我们这个时候不妨将这对花括号想象成一本书的封皮 (book symbols), 里面的逗号就像是每一页的边角, 见过书中间 (书脊) 折起来的部分吗? 他们代表着打开和折叠这本字典的历史.

翻开我们想象的字典, 将它平摊在桌面上, 每一页都是一个定义, 然后我们的逗号代表着页脚, 然后我们就可以翻到下一页. 在每一页上,<sup>34</sup>

<sup>32</sup>原文更有诗意: When you see that third dot, imagine opening the accordion slightly. Just enough to let one note from its chamber. The note is that end value. We'll let the sky eat it. 俺没文化, 翻译得不够好.

<sup>33</sup>这段不是很好翻译, 在翻译上请教了别人, 原文: Once there was a caterpillar who had commas for legs. Which meant he had to allow a literary pause after each step. The other caterpillars really respected him for it and he came to have quite a commanding presence. Oh, and talk about a philanthropist! He was notorious for giving fresh leaves to those less-fortunate.

在这里我给出一个比较随便的讲法: array 就是一组有顺序排列的东西. 就像是毛毛虫的身体一样, 一节一节的, 每一节里面都放着一个东西, 毛毛虫有头有尾, 所以类似的, 这些一节一节的都是有顺序的.

<sup>34</sup>这里不妨这样想, 左边一面是单词, 中间就是我们的"箭头", 右边一面是定义, 然后书的边角是逗号

## 译后记

怎么说呢, 感觉 why 先生的语言实在是妙, 在书中有各种各样的双关, 明喻和暗喻等等, 翻译起来的难度比较大. 然后发现网上已经有人给了一个翻译了, 虽然没有完全翻译, 但是这个版本的翻译应该是比较标准的.

既然有了比较标准的翻译了, 所以我就决定放飞自我了, 做一个比较随意的翻译了. 但是在编程部分我一定会认真翻译, 并且尽可能地去做注记的. 帮助你能够看懂. 虽然原书就已经非常容易理解了.

假如你觉得不太好的话, 建议还是去看原书吧.