目录

松本行弘的程序世界

本书为“Ruby”之父经典力作，展现了大师级的程序思考方式。作者凭借对编程本质的深刻认识和对各种技术优缺点的掌握，阐述了 Ruby 的设计理念，并由此延伸，带领读者了解编程的本质，一窥程序设计的奥秘。本书不是为了介绍某种特定的技术，而是从宏观的角度讨论与编程相关的各种技术。书中第 1 章介绍了作者对编程问题的新思考和新看法，剩下的内容出处《日经 Linux》杂志于 2005 年 5 月到 2009 年 4 月连载的“松本编程模式讲坛”，其中真正涉及“模式”的内容并不多，大量篇幅都用于介绍技术内幕和背景分析等内容，使读者真正了解相关技术的立足点。另外，书中还包含许多以 Ruby、Lisp、Smalltalk、Erlang、JavaScript 等动态语言所写成的范例。

- Ruby 之父佳作，进入不同凡响的程序世界
- 深入剖析程序设计的道与术
- 举一反三，触类旁通

内容提要

本书是探索程序设计思想和方法的经典之作。作者从全局的角度，利用大量的程序示例及图表，深刻阐述了 Rub 编程语言的设计理念，并以独特的视角考察了与编程相关的各种技术。阅读本书不仅可以深入了解编程领域各个要素之间的关系，而且能够学到大师的思考方法。本书面向各层次程序设计人员和编程爱好者，也可以供相关技术人员参考。

推荐序

在流行的编程语言中，Ruby 比较另类，这是因为大多数编程语言的首要着眼点在于为解决特定的问题领域而设计语言，而 Ruby 的首要着眼点在于“人性化”，让程序员充分享受编程的乐趣。由于组织国内的 Ruby 会议，我曾两次邀请松本行弘来中国。他是一位性格平和、对生活充满热爱的人，在演讲吕也一再传递 code for fun 的宗旨，即编程语言不应该是冷冰冰地给机器阅读的执行的指令，而应该昌让程序员编程的工作过程变成一种充满乐趣和享受的过程。而且，松本先生发明 Ruby 语言也是因为他对创造一种人性化的面向对象脚本语言的热爱。

程序员社区经常拿另外一个主流的面向对象脚本语言 Python 来和 Ruby 做对比。从全球范围来看，Python 的社区更大，应用更广泛，但 Ruby 的语法相对 Python 来说更强大和宽松，给程序员发挥的自由度更大，可以基于 Ruby 创建各个领域的 DSL，比方说 Ruby on Rails 就是一个基于 Ruby 的 Web 快速开发领域的 DSL。

总之，Ruby 语言的这种“人性化”以及给程序员很大编程自由度的气质奠定了整个 Ruby 社区的气质：热爱生活的程序员，所求编程的自由度，带点非主流的极客色彩。也下因为如此，Ruby 和基于 Ruby 的 Rails 得到了硅谷许许多多创业公司青睐，有名者如 Twitter、Groupon、Hulu、github 等。而这种气质也鲜明地体现在 Rails 框架的创建者 David Heinemeier Hansson 及其所在的 37signals 公司身上。37signals 的 20 多位员工遍全球，每天只上班四天，David Heinemeier Hansson 本人不是一位保时捷车队的职业赛车手。

当然，Ruby 并非只在非主流程序员社区中流行，随着全球 IT 产业进行云计算时代，Ruby 也发挥着越来越大的作用。著名的 SAAS 厂商 salesforce 在 2010 年底以 2.1 亿美元收购了 PAAS 厂商 Heroku，并且在 2011 年 7 月聘请松本行担任 Heroku 首席架构师，开拓 Ruby 在云计算领域的应用。Heroku 本身就是一个完全采用 Ruby 架构的 PAAS 平台，同样支持 Ruby 的 PAAS 厂商还有 EngineYard、VMware 等。随着这些云计算厂商的努力，Ruby 必然在未来得到越来越广泛的应用。我之前阅读了本书的部分章节，这本书实际上是松本行弘从一个编程语言设计者的角度去看待各种各样的流行编程语言，分析它们有哪些特点，以及 Ruby 编程语言是如何取舍的。Ruby 语言的设计本身大量参考了一个更古老而著名的面向对象编程方法的开山之作 Smalltalk，而且从函数式编程语言鼻祖 Lisp“偷师学艺”了不少好东西。程序员社区有个著名的说法：任何现代编程语言都脱胎于 Smalltalk 和

Lisp, 都与这两个编程语言有着似曾相识的特性，自 Smalltalk 和 Lisp 诞生以来，编程语言领域可谓大势已定了。因此，集这两种编程语言诸多特点于一身的 Ruby 语言很值得编程爱好者去学习，而看看 Ruby 设计师是怎么设计 Ruby 语言的，则可以让人高屋建领地理解一些主流的编程语言。

范凯

中文版序

从年轻的时候开始，我就对编程语言有着极为浓厚的兴趣。比起“使用计算机干什么”这一问题，我总是一门心思想着“如何将自己的意图传达给计算机”。从这个意义上说，我认为自己是个“怪人”。但是，想选择一个能让自己的工作变得轻松的编程语言，想编写一种让人用起来感到快乐的编程语言，一直是我梦寐以求的，这种迫切的心情怕不输于任何人。虽说是有点自卖自夸，但是 Ruby 确实给我带来了“快乐”，这一结果让我感到很满足。

让我感到惊奇的是，有很多人，包括那些没有我这么“怪”的人，都对这种快乐有着共鸣。Ruby 自 1995 年在互联网上公布以来，着实让世界各地的程序员都认识了它，共享着这种快乐，提高了软件开发的生产力。完全出乎我意料的是，世界各地的人，不管是东方还是西方，都极为欣赏 Ruby。在刚开始开发 Ruby 的时候，我想都没有想到过有这样的结果，程序员的感觉会超越人种、国籍、文化，有如此之多的共通之处。

现在，为世界各地的程序员所广泛接受的 Ruby，正带来一种新的文化。已经有越来越多的开发人员，在实践中果敢地施行着 Ruby 语言及其社区所追求的“对高生产力的追求”、“富有柔性的软件开发”、“对程序员人性的尊重”、“鼓起勇气挑战新技术”等原则。在 Ruby 以前，这些想法也都很好，却一直实践不起来。我相信，Ruby 的卓越之处，不仅在于语言能力，而且更重要的是引领了这种文化的践行。

本书在解说编程中的技术与原则时，不局限于表面现象，而是努力挖掘其历史根源，提示其本质。虽然很多章节都以 Ruby 为题材，但这些原则对于 Ruby 以外的语言也行之有效。衷心希望大家能够实践本书中所讲述的各项原则，成为一个更好的开发人员。

松本行弘

2011 年 4 月 18 日

DONE 前言 *<2016-08-30 二 14:10>* CLOSED: *[2016-08-30 二 15:30]*
本书的目的不是深入讲解哪种特定的技术，也没有全面讨论我所开发的编程语言 Ruby，而是从全局角度考察了与编程相关的各种技术。读者千万不要以为拿着这本书，就可以按图索骥地解决实际问题了。实际上，最好反它看成是一幅粗略勾勒出了编程世界诸要素之间关系的“世界地图”。

每种技术、思想都有其特定的目的、渊源和发展进步的过程。本书试图换一个角度重新考察各种技术。如果你看过后能够感觉到“啊，原来是这样的呀!”或者“噢，原来这个技术的立足点在这里呀!”那么我就深感心慰了。我的愿望就是这些知识能够激发读者学习新技术的求知欲。

本书的第 2 章到第 14 章，是在《日经 Linux》杂志于 2005 年 5 月到 2009 年 4 月连载的“松本编程模式讲坛”基础上编辑修改而成的。但实际上连载与最开始的设想并不一致，真正涉及“模式”的内容并不多，倒是技术内幕、背景分析等内容占了主流。现在想来，大方向并没有错。

除了连载的内容之外，本书还记录了我对编程问题的新思考和新看法。特别是第 1 章“我为什么开发 Ruby”，针对“为什么是 Ruby”这一点，比其他杂志做了更加深入的解说。另外，在每章的末尾增加了一个小专栏。

对于连载的内容，因为要出成一本书，除修改了明显的错误和不合时代的部分内容之外，我力求每一章都独成一体、内容完整，同时也保留了连载时的风貌。通读全书，读者也许会感到有些话题或讲解是重复的，这一点敬请原谅。

我的本职工作是程序员，不能集中大段时间去写书，不过无论如何最后总算是赶出来了。非常感谢我的家人，她们在这么长时间里宽容着我这个情绪不稳的丈夫和父亲。

稿子写完了，书也出来了，想着总算告一段落了吧，而《日经 Linux》又要开始连载“松本行弘技术剖析”了，恐怕还要继续让家里人劳心。

松本行弘

2009 年 4 月于樱花季节过后的松江

1 DONE 第一章我为什么开发 Ruby

1.1 1.1 我为什么开发 Ruby

Ruby 是起源于日本的编程语言。近年来，特别是因为其在 Web 开发方面的效率很高,Ruby 引起了全世界的关注，它的应用范围也扩展到了很多企

业领域。

作为一门编程语言，Ruby 正在被越来越多的人所了解，而作为一介工程师的我，松本行弘，刚开始的时候并没有想过“让全世界的人都来用它”或者“这下子可以大赚一笔了”，一个仅仅是从兴趣开始的项目却在不知不觉中发展成了如今的样子。

当然了，那时开发 Ruby 并不是我的本职工作，纯属个人兴趣，我是把它作为一个自由软件来开发的。但是世事弄人，现在开发 Ruby 竟然变成了我的本职工作了，想想也有些不可思议。

“你为什么开发 Ruby ?”每当有人这样问我的时候，我认为最合适的回答应该就像 Linux 的开发者 Linus Torvalds 对“为什么开发 Linux”的回答一样吧———

“因为它给我带来了快乐。”

当我还是一个高中生，刚刚开始学习编程的时候，不知何故，就对编程语言产生了兴趣。

周围很多喜欢计算机的人，[1]有的是“想开发游戏”，有的是“想用它来做计算”，等等，都是“想用计算机来做些什么”。而我呢，则想弄明白“要用什么编程语言来开发”、“用什么语言开发更快乐”。

高中的时候，我自己并不具备开发一种编程语言所必需的技术知识，而且当时也没有计算机。但是，我看了很多编程语言类的书籍和杂志，知道了“还有像 Lisp 这样优秀的编程语言”、“Smalltalk 是做面向对象设计的”，等等，在这些方面我很着迷。上大学时就自然而然地选修了计算机语言专业。10 年后，我通过开发 Ruby 实现了自己的梦想。

从 1993 年开发 Ruby 到现在已经过去 16 年了。在这么久的时间里，我从未因为设计 Ruby 而感到厌烦。开发编程语言真是一件非常有意思的事情。

1.1.1 1.1.1 编程语言的重要性

为什么会喜欢编程语言？我自己也说不清。至少，我知道编程语言是非常重要的。

最根本的理由是：语言体现了人类思考的本质。在地球上，没有任何超越人类智慧的生物，也只有人类能够使用语言。所以，正是因为语言，才造成了人类和别的生物的区别；正是因为语言，人和人之间才能传递知识和交

[1]当时喜欢计算机的人当然还是少数。

流思想，才能做深入的思考。如果没有了语言人类和别的动物也就不会有太大的区别了。

在语言领域里，有一个 Sapir-Whirf 假说，认为语言可以影响说话者的思想。也就是说，语言的不同，造成了思想的不同。人类的自然语言是不是像这个假说一样，我不是很清楚，但是我觉得计算机语言很符合这个假说。也就是说，程序员由于使用的编程语言不同，他的思考方法和编写出来的代码都会受到编程语言的很大影响。

也可以这么说，如果我们选择了好的编程语言，那么成为好程序员的可能性就会大很多。

20 年来一直被奉为名著的《人月神话》的作者 Frederick. Brooks 说过：一个程序员，不管他使用什么编程语言，他在一定时间里编写的程序行数是一定的。如果真是这样，一个程序员一天可以写 500 行程序，那么不论他用汇编、C, 不是 Ruby，他一天都应该可以写 500 行程序。

但是，汇编的 500 行程序和 Ruby 的 500 行程序所能做的事情是有天壤之别的。程序员根据所选择编程语言的不同，他的开发效率就会有十倍、百倍甚至上千倍的差别。

由于价格降低、性能提高，计算机已经很普及了。现在基本上各个领域都使用了计算机，但如果没有软件，那么计算机这个盒子恐怕一点用都没有了。而软件开发，就是求能够用更少的成本、更短的时间，开发出更多的软件。

需要开发的软件越来越多，开发成本却有限，所以对于开发效率的要求就很高。编程语言就成了解决这个矛盾的重要工具。

1.1.2 1.1.2 Ruby 的原则

Ruby 本来是我因兴趣开发的。因为对多种编程语言都很感兴趣，我广泛对比了各种编程语言，哪些特性好，哪些特性没什么用，等等，通过一一进行比较、选择，最终把一些好的特性吸纳进了 Ruby 编程语言之中。

如果什么特性都不假思索地吸纳，那么这种编程语言只会变成以往编程语言的翻版，从而失去了它作为一种新编程语言的存在价值。

编程语言的设计是很困难的，需要仔细斟酌。值得高兴的是，Ruby 的设计很成功，很多人都对 Ruby 给出了很好的评价。

那么，Ruby 编程语言的设计原则是什么呢？

Ruby 编程语言的设计目标是, 让作为语言设计者的我能够轻松编程, 进

而提高开发效率。

根据这个目标，我制订了以下 3 个设计原则。

- 简洁性
- 扩展性
- 稳定性

关于这些原则，下面分别加以说明。

1.1.3 1.1.3 简洁性

以 Lisp 编程语言为基础而开发的商业软件 Viaweb 被 Yahoo 收购后，Viaweb 的作者 PaulGraham 也成了大富豪。最近他又成了知名的技术专栏作家，写了一篇文章就叫“简洁就是力量”。[2]

他还撰写了很多倡导 Lisp 编程语言的文章。在这些文章中他提到，编程语言在这半个世纪以来是向着简洁化的方向发展的，从程序的简洁程度就可以看出一门编程语言本身的能力。上面提到的 Brooks 也持同样的观点。

随着编程语言的演进，程序员已经可以更简单、更抽象地编程了，这是很大的进步。另外随着计算机性能的提高，以前在编程语言里实现不了的功能，现在也可以做到了。

面向对象编程就是这样的例子。面向对象的思想只是把数据和方法看作一个整体，当作对象来处理，并没有解决以前解决不了的问题。

用面向对象记述的算法也一定可以用非面向对象的方法来实现。而且，面向对象的方法并没有实现任何新的东西，却要在运行时判定要调用的方法，倾向于增大程序的运行开销。即使是实现同样的算法，面向对象和程序往往更慢，过去计算机的执行速度不够快，很难请允许我像这样的“浪费”。

而现在，由于计算机性能大大提高，只要可以提高软件开发效率，浪费一些计算机资源也无所谓了。

再举一些例子。比如内存管理，不用的内存现在可用垃圾收集器自动释放，而不用程序员自己去释放了。变量和表达式的类型检查，在执行时已经可以自动检查，而不用在编译时检查了。

---

[2] Paul Graham 目前是世界知名的天使投资人，其公司 Y Combinator 投资了很多极有前途的创业项目。Paul Graham 曾出版过两本 Lisp 专著，最新著作《黑客与画家》已经由人民邮电出版社出版。————编者注

我们看一个关于斐那契 (Fibonacci) 数的例子。图 1-1 所示为用 Jave 程序来计算斐波那契数。算法有很多种，我们最常用的递归算法来实现。

图 1-2 所示为完全一样的实现方法，它是用 Ruby 编程语言写的，算法完全一样。和 Java 程序相比，可以看到构造完全一样，但是程序更简洁。Ruby 不进行明确的数据类型定义，不必要的声明都可以省略。所以，程序就非常简洁了。

```
class Sample {
 private static int fib (int n){
   if (n<2){
       return n;
    }
    else {
         return fib (n-2) +fib (n-1);
        }
  }
  public static void main(String[]argv){
    System.out.println("bib(6)-"+fib(6));
 }
}
```

图 1-1 计算斐波那契数的 Java 程序

```
def fib(n)
  if n<2
   n
  else
    fib(n-2) +fib(n-1)
  end
end
print "fib(6)=",fib(6),"\n"
```

图 1-2 计算斐波那契数的 Ruby 程序

算法的教科书总是用伪码来描述算法。如果像这样用实际的编程语言来描述算法，那么像类型定义这样的非实质代码就会占很多行，让人不能专心于算法。如果可以反伪码中非实质的东西去掉，只保留描述算法的部分就直

接运行，那么这种编程语言不就是最好的吗？Ruby 的目标就是成为开发效率高、“能直接运行的伪码式语言”。

1.1.4 1.1.4 扩展性

下一个设计原则是“扩展性”。编程语言作为软件开发工具，其最大的特征就是对要实现的功能事先没有限制。“如果想做就可以做到”，这听起来像小孩子说的话，但在编程语言的世界里，真的就是这么一回事。不管在什么领域，做什么处理，只要用一种编程语言编写出了程序，我们就可以说这种编程语言适用于这一领域。而且，涉及领域之广远远超出我们当初要预想。

1999 年，关于 Ruby 的第一本书《面向对象脚本语言 Ruby》出版的时候，我在里面写道，“Ruby 不适合的领域”包括“以数值计算为主的程序”和“数万行的大型程序”。

但在几年后，规模达几万行、几十万行的 Ruby 程序被开发出来了。气象数据分析，乃至生物领域中也用到了 Ruby。现在，美国国家海洋和航天局（NOAA,National Oceanic and Atmospheric Administration）、美国国家航空和航天局（NASA，National Aeronautice and Space Administration）也在不同的系统中运用了 Ruby。

情况就是这样，编程语言开发者事先并不知道这种编程语言会用来开发什么，会在哪些领域中应用。所以，编程语言的扩展性非常重要。

实现扩展性的一个重要方法是抽象化。抽象化是指把数据和要做的处理封装起来，就像一个黑盒子，我们不知道它的内部是怎么实现的，但是可以用它。

以前的编程语言在抽象化方面是很弱的，要做什么处理首先要了解很多编程语言的细节。而很多面向对象或者函数式的现代编程语言，都在抽象化方面做得很好。

Ruby 也不例外。Ruby 从刚开始设计时就用了面向对象的设计方法，数据和处理的抽象化提高了它的开发效率。我在 1993 年设计 Ruby 时，在脚本编程语言中采用面向对象思想的还很少，用类库方式来提供编程语言的就更少了。所以现在 Ruby 的成功，说明当时采用面向对象方法的判断是正确的。

Ruby 的扩展性不仅仅体现在这些方面。

比如 Ruby 以程序块这种明白易懂的形式给程序员提供了相当于 Lisp 高阶函数的特性，使“普通的程序员”也能够通过自定义来实现控制结构的高

阶函数扩展。又比如已有类的扩展特性，虽然有一定的危险性，但是程序却可以非常灵活地扩展。关于这些面向对象、程序块、类扩展特性的内容，后面的章节还会详细介绍。

这些特性的共同特点是，它们都表明了编程语言让程序员最大限度地获得了扩展能力。编程语言不是从安全角度考虑以减少程序员犯错误，而是在程序员自己负责的前提下为他提供最大限度发挥能力的灵活性。我作为 Ruby 的设计者，也是 Ruby 的最初用户，从这种设计的结果可以看出，Ruby 看重的不是明哲保身，而是如何最大限度地发挥程序员自身的能力。

关于扩展性，有一点是不能忽视的，即“不要因为想当然而加入无谓的限制”。比如说，刚开始开发 Unicode 时，开发者想当然地认为 16 们（65535 个字符）就足够容纳世界上所有的文字了；同样，Y2K 问题也是因为想当然地认为用 2 位数表示日期就够了才导致的。从某种角度说，编程的历史就是因为想当然而失败的历史。而 Ruby 对整数范围不做任何限定，尽最大努力排除“想当然”。

1.1.5 1.1.5 稳定性

虽然 Ruby 非常重视扩展性，但是有一个特性，尽管明知道它能带来巨大的扩展性，我却一直将其拒之门外。那就是宏，特别是 Lisp 风格的宏。

宏可以替换掉原有的程序，给原有的程序加入新的功能。如果有了宏，不管的控制结构，还是赋值，都可以随心所欲的进行扩展。事实上，Lisp 编程语言提供的控制结构很大一部分都是用宏来定义的。

所谓 Lisp 流，其语言核心部分仅仅提供极为有限的特性和构造，其余的控制结构都是在编译时通过用宏来组装其核心特性来实现的。这也就意味着，由于有了这种无与伦比的扩展性，只要掌握了 Lisp 基本语法 S 式（从本质上讲就是括号表达式），就可以开发出千奇百怪的语言。Common Lisp 的读取宏提供了在读取 S 式的同时进行语法变换的功能，这就在实际上摆脱了 S 式的束缚，任何语法的语言都可以用 Lisp 来实现。

那么，我为什么拒绝在 Ruby 中引入 LIsp 那样的宏呢？这是因为，如果在编程语言中引入宏的话，活用宏的程序就会像是用完全不同的专用编程语言写出来的一样。比如说 Lisp 就经常有这样的现象，活用宏编写的程序 A 和程序 B，只有很少一部分昌共通的，从语法到词汇都各不相同，完全像是用不同的编程语言写的。

对程序员来说，程序的开发效率固然很重要，但是写出的程序是否具有

很高的可读性也非常重要。从整体来看，程序员读程序的时间可能比写程序的时间还长。读程序包括为理解程序的功能去读，或者是为维护程序去读，或者是为调试程序去读。

编程语言的语法是解读程序的路标。也就是说，我们可以不用追究程序或库提供的类和方法的详细功能，但是，“这里调用了函数”、“这里有判断分支”等基本的“常识”在我们读程序时很重要。

可是一旦引入了宏定义，这一常识就不再适用了。看起来像是方法调用，而实际上可能是控制结构，也可能是赋值，也可能有非常严重的副作用，这就需要我们去查阅每个函数的方法的文档，解读程序就会变得相当困难。

当然了，我知道世界上有很多 Lisp 程序员并不受此之累，他们只是极少数的一部分程序员。

我相信，作为在世界上广泛使用的编程语言，应该有稳定的语法，不能像随风飘荡的灯芯那样闪烁不定。

1.1.6 1.1.6 一切皆因兴趣

当然，Ruby 不是世界上唯一的编程语言，也不能说它是最好的编程语言。各种各样的编程语言可以在不同的领域吕应用，各有所长。我自己以及其他 Ruby 程序员，用 Ruby 开发很高，所以觉得 Ruby“最为得心应手”。当然，用惯了 Python 或者 Lisp 的程序员，也会觉得那些编程语言是最好的。

不管怎么说，编程语言存在的目的是让人用它来开发程序，并且尽量能提高开发效率。这样的话，才能让人在开发中体会到编程的乐趣。

我在海外讲演的时候，和很多人交流过使用 Ruby 的感想，比较有代表性的是 :“用 Ruby 开发很快乐，谢谢!”

是啊，程序开发本来就是一件很快乐、很刺激和很有创造性的事情。想起中学的时候，用功能不强的 BASIC 编程语言开发，当时也是很快乐的。当然，工作中会有很多的限制和困难，编程也并不都是一直快乐的，这也是世之常情。

Ruby 能够提供很高的开发效率，让我们在工作中摆脱很多困难和烦恼，这也是我开发 Ruby 的目的之一吧。

2 第二章面向对象

2.1 2.1 编程和面向对象的关系

所谓编程，就是把工作的方法告诉计算机。但是，计算机是没有思想的，它只会简单地按照我们说的去做。计算机看起来功能很强大，其实它也仅仅只会做高速计算而已。如果告诉它效率很低的方法，它也只是简单机械地去执行。所以，到底是最大程序地发挥计算机的能力，不是扼杀它的能力，都取决于我们编写的程序了。

程序员让计算机完全按照自己的意志行事，可以说是计算机的“主宰”。话虽如此，但世人多认为程序员是在为计算机工作。

不，不只是一般人，很多计算机业内人士也是这样认为的，甚至比例更高。难道因为是工作，所以就无可奈何了吗？

2.1.1 2.1.1 颠倒的构造

如果仔细想想，就会感到很不可思议。为什么程序员非要像计算机的奴隶一样工作呢？我们到底是从什么时候放弃主宰计算机这个念头的呢？

我想，其中的一个原因是“阿尔法综合征”。阿尔法综合征是指在饲养宠物狗的时候，宠物狗误解了一直细心照顾它的评价的地位，反而感觉到它自己是主人，比主人更了不起。

计算机也不是好伺候的。系统设计困难重重，程序有时也会有错误。一旦有规格变更，程序员就要动手改程序，程序有了错误，也需要一个个纠正过来。

所以在诸如此类烦琐的工作中，就会发生所谓的“逆阿尔法综合征”现象，主从关系颠倒，话务员沦为“计算机的奴隶”，说的客气一些，也顶多能算是“计算机的看门狗”。难道这是人性使然？

不，不要轻易放弃。人是万物之灵，比计算机那玩意儿要聪明百倍，当然应该摆脱计算机奴隶的地位，把工作都推给机器来干，自己尽情享受轻松自在。因此，我们的目标就是让程序员夺回主动权！

程序员如果能够充分利用好计算机所具有的高速计算能力和信息处理能力，有可能会从奴隶摇身一变，“像变戏法一样”完成工作，实现翻天覆地的大逆转。

但是，要想赢得这场夺回主动权的战争，“武器”是必须的。那就是本书是讲解的“语言”和“技术”。

Ruby 的安装

读者中恐怕有不少人的初次安装 Ruby，所以这里再介绍一下 Ruby 的安装方法。我在写这本书的时候，Ruby 的版本是 1.9.1。在我平时使用的 Debian GNU/Linux 操作系统中，用下面的方法来安装 Ruby。

$ apt-get install ruby

其他的 Linux 操作系统大多也提供了 Ruby 的开发包。

在 Windows 操作系统中安装 Ruby 时，直接点击安装文件就可以了。从下面的网站可以下载安装程序：http://rubyinstaller.rubyforge.org。

如果从 Ruby 源程序来编译安装的话，可以从下面的网站来下载 Ruby 源程序包（tarball):http://ruby-lang.org。

编译和安装的方法如下。

$ tar zxvf ruby-1.9.1-p0.tar.gz

$ cd ruby-1.9.1-p0

$ ./configure

$ make

$ su

$ make install

2.1.2 2.1.2 主宰计算机的武器

程序员或者将要成为程序员的人，如果成了计算机的奴隶，那是十分不幸的。为了能够主宰计算机，必须以计算机的特性和编程语言作为武器。

编程语言是描述程序的方法。目前有很多种编程语言，有名的有 BASIC、FORTRAN、C、C++、Java、Perl、PHP、Python、Ruby 等。

从数学的角度来看，几乎所有的编程语言都具备“图灵完备”[3]的属性，无论何种编程语言都可以记述等价的程序，但这并不是说选择什么样的编程语言都一样。每种编程语言都有自己的特征、属性，都各有长处和短处、适合的领域和不适合的领域。写程序的难易程度（生产力）也有很大的不同。

有研究表明，开发程序时用的编程语言和生产力并没有关系，不论用什么编程语言，一定时间内程序的开发规模（在一定程度上）是相当的。还有一些研究表明，为了完成同样的任务，程序规模会因为开发时选取的编程语

[3]图灵完备指在可计算性理论中，编程语言或任意其他的逻辑系统具有等同于能用图灵机的计算能力。换言之，此系统可与能用图灵机互相模拟。这个词源于引入图灵机概念的数学家阿兰. 图灵 (Alan Turing)。

言和库而相差数百倍，甚至数千倍。所以如果选用了合适的编程语言，那么你的能力就可能增长数千倍。

但是不论什么都是有代价的。比如效率高的开发环境，在执行时效率往往很低。还有很多领域需要人们想尽办法去提高速度。在这里，因为我们在讨论如何主宰计算机，所以尽可能地选择让人们轻松的编程语言。基于这个观点，本书用 Ruby 语言来讲解。当然，Ruby 是我设计的，讲解起来相对也就容易点。

Ruby 是面向对象的编程语言，具有简洁和一致性。开发 Ruby 的宗旨是用它可以轻松编程。

Ruby 的运行环境多种多样，包括 Linux 及 UNIX 系列操作系统、Windows、MacOS X 等各种平台，很多系统上都有 Ruby 的软件包[4]。当然，如果有 C 编译器，也可以从源程序来安装 Ruby。

2.1.3 2.1.3 怎样写程序

使用编程语言写好程序是有技巧的。在本书中，将会介绍表 2-1 中列出的编程技巧。

表中的编程风格指的是编程的细节，比如变量名的选择方法、函数的写法等。

算法是解决问题的方法。现实中各种算法都已经广为人知了，所以编程时的算法也就是对这些技巧的具体应用。

有很多算法如果单靠自己去想是很想出来的。比方说数组的排序就有很多的算法，如果我们对这些算法根本就不了解，那么要想做出调整排序程序会很困难。算法和特定的数据结构关系很大。所以有一位计算机先驱曾经说过 :“程序就是算法回味数据结构”。[5]

设计模式是指设计软件时，根据以前的设计经验对设计方法进行分类。算法和数据结构从广义上来说也是设计模式的一种分类。有名的分类（设计模式）有 23 种[6]。

开发方法是指开发程序时的设计方法，指包括项目管理在内的整个程序开发工程。小的软件项目可能不是很明显，在大的软件项目中，随着开发人员的增加，整个软件工程的开发方法就很重要。

[4]比如 MacOS X10.5 中标准搭载了 Ruby 1.8.6。

[5]Algorithms + Data Structures = Programs, Niklaus Wirth 著。Wirth 是在 1971 年开发了 Pascal 编程语言的计算机学者。

[6]《设计模式：可复用面向对象软件基础》，Erich Gamma 等著，机械工业出版社出版。

2.1.4 2.1.3 面向对象的编程方法

下面，我们来看看 Ruby 的基本原理————面向对象的设计方法。面向对象的设计方法是 20 世纪 60 年代后期，在诞生于瑞典的 Simula 编程语言中最早开始使用的。Simula 作为一种模拟语言，对于模拟的物体，引入了对象这种概念。比如说对于交通系统的模拟，车和信号就变成了对象。一辆辆车和一个个信号就是一个个对象，而用来定义这些车和信号的，就是类。

此后, 从 20 世纪 70 年代到 80 年代前期, 美国施乐公司的帕洛阿尔托研究中心 (PARC) 开发了 Smalltalk 编程语言。从 Smalltalk-72、Smalltalk-78 到 Smalltalk-80，他们开发完成了整个 Smalltalk 系列。Smalltalk 编程语言对近代面向对象编程语言影响很大，所以把它称为面向对象编程语言之母也不为过。

在这之后, 受 Simula 影响比较大的有 C++ 编程语言, 再以后还有 Jave 编程语言，而现在大多数编程语言使用的教师面向对象的设计方法。

2.1.5 2.1.5 面向对象的难点

面向对象的难点在于，虽然有关于面向对象的说明和例子，但是面向对象具体的实现方法却不是很明确。

面向对象这个词本身是很抽象的，越抽象的东西，人们就越难理解。并且对于面向对象这个概念，如果没有严密的定义，不同的人就会有不同的理解。

这里，我们暂时回避一下“面向对象”的整体概念这一问题，首先集中说明“面向对象编程”。

至于“好像是听明白了，还是不会使”这一点，原因可能在于平易的比喻和实际编程之间差距太大。这里，我们选择 Ruby 这种简单易用的面向对象编程语言，希望能够拉近比喻和实例之间的距离。

另外很重要的一点，面向对象编程语言有很多种类，也有很多技巧。一下子全部理解是很多困难的，我们分别加以说明。

我认为面向对象编程语言中最重要的技术是“多态性”。我们就先从多态性说起吧。

2.1.6 2.1.6 多态性 secret

多态性，英文是 polymorphism，其中词头 poly-表示复数，morph 表示形态，加上词尾-ism, 就是复数形态的意思，我们称它为多态性。

换个说法，多态就是可以把不同种类的东西做相同的东西来处理。

只从字面上分析不容易理解，举例说明一下。

看看图 2-1 所示的 3 个箱子。每个箱子都有不同的盖子。一个是一般的盖子，一个是带锁的盖子，一个是带有彩带的盖子。因为箱子本身非常昂贵，所以每个箱子都有专人管理，如果要从箱子里取东西，要由管理人员去做。

打开 3 个箱子的方法都不同，但如果发出同样的打开箱子的命令，3 个人会用自己的方法来打开自己的箱子。因此，3 个箱子虽然各有不同，但它们同样“都是箱子，可以打开盖子”。这就是多态性的本质。

在编程中，“打开箱子”的命令，我们称之为消息；而打开不同箱子的具体操作，我们称之为方法。

2.1.7 2.1.7 具体的程序

上面例子的程序如图 2-2 所示。

box_{open} 是打开箱子的方法，相当于前面所说的“管理员”。调用 box_{open} 这个方法时，方法会根据参数（箱子和种类）的不同做相应的处理。你只要说“打开箱子”，箱子就真地被打开了。这种“根据对象不同类型而进行适当地处理”就是多态性的基本内容。

但只有图 2-2 还不够。我们来考虑一下如何定义 box_{open} 这个方法吧。如果只是单纯地实现这个方法，也许就会写成图 2-3 的样子。

但是，图 2-3 所示的处理并不能令人满意。如果要增加箱子种类，这个方法中的代码就要重写，而且如果还有其他类似于 box_{open} 、需要根据箱子类型来做不同的处理的方法，那么需要修改的地方就越来越多，追加箱子种类就会变得非常困难。

程序修改得越多，出错的可能性也就越大，结果可能是程序本身根本就动不起来了。

像这样的修改本来就不该直接由人来做。根据数据类型来进行合适的处理（调用合适的方法），本来就应该是编程语言这种工具应该完成的事。只有实现了这一点，才能称为真正的多态。

为此，我们修改一下图 2-2 的程序，来看看真正的多态是如何工作的。

图 2-4 的程序把参数移到了前头，并增加了一个“.”。这行代码可以理解为“给前面式子的值发送 open 消息”。也就是说，它会“根据前面式子的值，调用合适的 open 方法”。这就是利用了多态性的调用方法。

图 2-4 程序中的各种处理方法的定义如图 2-5 所示。

图 2-5 的程序定义了 3 种箱子：box1、box2、box3，表示“打开箱子”的不同方法。

比较图 2-5 和图 2-3 的程序可以看到，程序中不再有直白的条件判断，非常简明了。即使在图 2-5 中程序增加一种新的箱子，比如“横向滑动之后打开箱子”，也不需要对原来的程序做任何修改。不需要修改，当然也就没有因修改而出错的危险。

图 2-2 例子的程序

```
# 用变量 box1 box2 box3 代表3个箱子

box_open(box1) # 表示打开箱子
box_open(box2) # 表示开锁, 打开箱子
box_open(box3) # 表示解开彩带, 打开箱子
```

图 2-3 图 2-2 例子的 box-open 方法的内容

```
def box_open(box)

# 判断 box 类型的方法
if box_type(box)=="plain"
  puts("打开箱子")
elsif box_type(box)=="lock"
 puts("开锁,打开箱子")
elsif box_type(box)=="ribbon"
 puts("解开彩带, 打开箱子")
else
 puts("不知道打开箱子的方法")
end

end
```

2.1.8 2.1.8 多态性的优点

前面说明了多态性，那么它到底有什么好处呢？

首先，各种数据可以统一地处理。多态性可以让程序只关注要处理什么（What），而不是怎么去处理（How）。

其次，是根据对象的不同自动选择最合适的方法，而程序内部则不发生冲突。不管调用有锁的箱子，还是系着彩带的箱子，它们都能自动处理，不用担心调用中会发生错误，这样就会减轻程序员的负担。

再次，如果有新数据需要对应处理的话，通过简单的追加就可以实现了。而不需要改动以前的程序，这就让程序具备了扩展性。

综上所述，多态性提高了开发效率，所以说，面向对象技术最重要的一个概念应该是多态性。

相关的 Ruby 语法

为了让读者能理解本书中的程序例子，这里简单说明一下 Ruby 语法。

首先，以"#" 开始的行是注释行，注释的内容随便是什么都可以。

```
# 这一行是注释行

条件判断用if语句。

if 条件
    处理代码
elsif 条件
    处理代码
else
    处理代码
end
```

具体的程序如图 2-6 所示。

```
if box_type(box)=="plain"
  puts("打开箱子")
elsif box_type(box)=="lock"
  puts("用钥匙打开箱子")
elsif box_type(box)=="ribbon"
  puts("解开彩带，打开箱子")
else
  puts("不知道打开箱子的方法")
end
```

图 2-6 条件判断程序

当第一个条件成立的时候，就执行第一段处理代码；当第二个条件成立的时候，就执行第二段处理代码；而当所有条件都不成立的时候，就执行 else 下面的处理代码。如果处理代码由多条语句并列构成，不需要用“{}”括起来，而是用 elsif 或者 end 等保留词来分隔，这一点也许会让你觉得耳目一新。

在 Ruby 的 if 语句中，elsif 部分可以重复出现任意次。当然也可以是 0 次，这时候 elsif 是可以省略的。else 同样也是可以省略的。

对于“plain”来说，"" 中的是字符串。与数值一样，字符串也是能直接写在程序里的数据。在 Ruby 中，这些数据都是对象，我们将在以后的章节中详细说明。

像 box 这样，以小写英文字母开头的是变量。这个例子中已事先设置好了变量的值。像其他的编程语言一样，变量的赋值语句是

变量 = 值发送

用来初始化变量。

要判断两个表达式的值是否一样，可以使用“==”运算符。

表达式 == 表达式

请注意，在赋值语句中是用一个等号，而判断两个表达式更不相等则是用两个等号。这跟 Java 或 C 等许多语言中的用法也都是一样的。

后面有小括号的语句是方法调用。如

puts(" 打不开箱子")

puts 方法可以把字符串显示在画面上。

最后，使用 def 语句来定义方法。

```
def 方法名 （参数1，……）
    处理代码
end
```

2.2 数据的抽象和继承

多态性、数据抽象和继承被称为面向对象的三原则。这三项原则通常也会有别的称谓。例如，把多态性称为动态绑定，把数据抽象称为信息隐藏或封装，虽然名称不同，但是内容都是相同的。许多人认为这些原则是面向对

象程序设计的重要原则[7]。

2.2.1 2.2.1 面向对象的历史

新接触面向对象的人可能觉得它难以理解。事实上，对于从事面向对象编程有 15 年以上的我来说，有很多概念还是觉得很难理解。

自 20 世纪 60 年代末至今，面向对象的思想已经经过了 40 多年的发展。猛一看这些一步步积累起来的成果，你可能会觉得数量庞大。然而，如果沿着面向对象的发展历史一步步开始去学习的话，那么看起来很难的面向对象概念，实际上比我们想象中的要简单。

首先，我们回顾一下面向对象的发展历史。对不必担心讲解历史过程中提到的一些陌生的词语，后面会详细说明。

Simula 的"发明"

如前所述，面向对象编程思想起源于瑞典 20 世纪 60 年代后期发展起来的模拟编程语言 Simula。以前，表示模拟对象的数据和实际的模拟方法互相独立的，需要分别管理，编程时需要把两者正确地结合起来，程序员的负担是很重的。因此，Simula 引入了数据和处理数据的方法自动结合的抽象数据类型。随后，又增加了类和继承的功能。其实在 20 世纪 60 年代后期现代面向对象编程语言的基本特征 Simula 都已经具备了。

Smalltalk 的发展

Simula 的面向对象编程思想被广泛传播。从 20 世纪 70 年代到 80 年代初，美国施乐公司的帕洛阿尔托研究中心开发了 Smalltalk 编程语言。当时的开发宗旨是"让儿童也可以使用"。在 Lisp 和 LOGO 设计思想的基础上，Smalltalk 又吸取了 Simula 的面向对象思想，且独具一格。不仅如此，它还有一个很好的图形用户界面。这个创新的语言使得世人开始了了解面向对象编程的概念。

Lisp 的发展

另外，位于美国东海岸的麻省理工学院及其周边地区，用 Lisp 语言发展了面向对象的思想。Lisp 和 FORTRAN、COBOL 语言一样，都是最古老的语言。与同时期登场的其他语言不同，Lisp 语言具有非常浓厚的数学背景，所以它本身具有很强的扩展功能。面向对象的特性也是 Lisp 所拥有的。

[7]三原则虽然是非常重要的，但是在面向对象编程中并不是必不可少的。比如有不支持继承的面向对象编程语言（JaveScript）和不支持封装的面向对象编程语言（CLOS,Common Lisp Object System)。

因此，编程语言规格的变更、功能的扩展和实验都很容易进行，由此产生了很多创新的想法。多重继承、混合式和多重方法等，许多重要的面向对象的概念都是从 Lisp 的面向对象功能中诞生的。

和 C 语言的相遇

20 世纪 80 年代，世界上很多地方都在研究面向对象编程思想。AT&T 公司的贝尔实验室在 C 语言中追加了面向对象的功能，开发出了“C with Class”编程语言。开发者是 Biame Stroustrup，他来自距离 Simula 的起源瑞典不远的丹麦。在英国剑桥大学的时候，Stroustrup 就使用 Simula 语言。加入贝尔实验室以后，为了能够把 C 语言的高效率和 Simula 的面向对象功能结合起来，他开发了“C with Class”编程语言。

因为当时 Simula 的处理速度是非常缓慢的，所以在他的研究领域中不能使用。“C with Class”语言就演变成了后来的 C++ 语言。从这些情况来看，C++ 是直接受到了 Simula 语言的影响，而没有受到 Smalltalk 多大影响。

Java 的诞生

强调与 C 语言兼容的 C++ 语言，能够写低级的方法，这是有利有弊的。为了克服低级语言的缺点，在 20 世纪 90 年代 Java 编程语言应运而生。Java 语言放弃了和 C 语言的兼容性，并增加了 Lisp 语言中一些好的功能。此外，通过 Java 虚拟机（JVM），Jave 程序可以不用重新编译而在所有的操作系统中运行。

现在，Java 作为在 20 世纪 90 年代诞生的最成功的语言，被全世界广泛应用。

面向对象编程方法和编程语言一样在不断地演变发展。到了 20 世纪 90 年代，面向对象的方法在软件设计和分析等软件开发的上层领域中流行起来。1994 年，当时主要的面向对象分析和设计方法 Booth、OMT(Object Modeling Technique) 以及 OOSE(Object Oriented Softrware Engineering) 的发明人 Grady Booth、Jim Rumbaugh 和 Ivar Jacobson 合作设计了 UML(Unified Modeling Language)。UML 是用来描述通过面向对象方法设计的软件模型的图示方法，也是利用这种记法进行分析和设计的一种方法论。

UML 提供了很多设计高可靠性快软件的面向对象设计方法。但是，UML 整体上很复杂，用到的概念很多，会让初学者觉得很难掌握。

面向对象的基本概念建立以后，催生了各种编程语言。

2.2.2 2.2.2 复杂性是面向对象的敌人

我们再回到面向对象的重要原则，来了解真正的面向对象编程。

软件开发的最大敌人是复杂性。人类的大脑无法做太复杂的处理，记忆力和理解力也是有限的。

计算机上运行的软件却没有这样的限制，无论多么复杂的计算机软件，无论有多少数据，无论需要多长时间，计算机都可以处理。随着越来越多的数据要用计算机来处理，对软件的要求也越来越高，软件也变得越来越复杂。

虽然计算机的性能年年在提高，但它的处理能力终究是有限的，而人类理解力的局限性给软件生产力带来的限制更大。在计算机性能这么高的今天，人们为了找到迅速开发大规模复杂软件的方法，哪怕牺牲一些性能也在所不惜。

2.2.3 2.2.3 结构化编程

最初对这种复杂的软件开发提出挑战的是“结构化编程”。结构化编程的基本思想是有序地控制流程，即把程序的执行顺序限制为顺序、分支和循环这 3 种，把共通的处理归结为例程（见图 2-7）。p/图 2-7.png p/图 2-7.bb

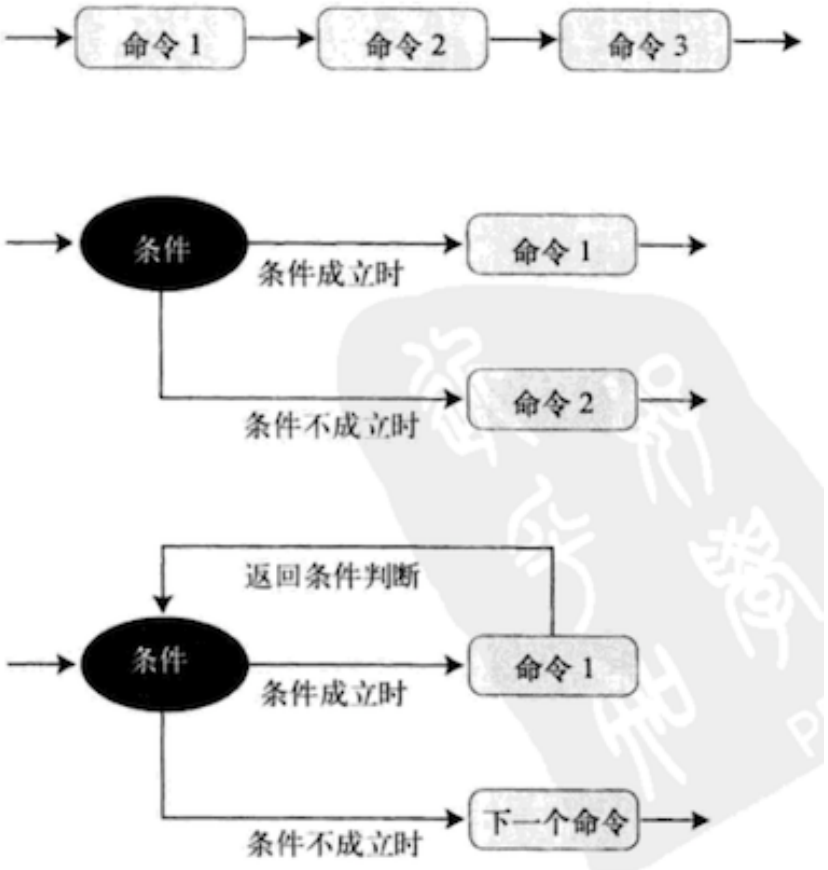

图 2-7 顺序、分支和循环的处理方法

在结构化编程出现之前，可以用 goto 语句来控制程序的流程，执行流可以转移到任何地方。而结构化编程用上文所述的 3 种语句控制程序的流程。这样可以降低程序流程的复杂性，此外，还引入了较为抽象的处理块（例程）的概念，也就是把基本上相同的处理抽象成例程，其中不同的地方由外部传递进来的参数来对应。

结构化编程的“限制”和“抽象化”，是人类处理复杂软件的非常有效的方法。

通过限制大大降低了程序的自由度，减少了各种组合，使得程序不至于太过复杂。但是如果由于降低了程序的自由度而导致程序的实现能力低下，那是我们所不愿看到的。而结构化编程的顺序、分支和循环这可以实现一切算法，虽然降低了程序的复杂性和灵活性，但是程序的实现能力并没有降低。

抽象化的目的是我们只需要知道过程的名字，而并不需要知道过程的内部细节，因此它也被称为“黑盒化”。我们只需要知道“黑盒子”的输入和输出，

而过程的细节是隐藏的。[8]

例如，如果你知道了例程的输入和输出，那么即使不知道处理的内部细节也可以利用这个例程。建立一个由黑盒子组合起来的系统，复杂的结构被黑盒子隐藏起来，这样我们就可以更容易、更好地理解系统的整体结构。

如果把黑盒子内的处理也考虑上，整个系统的复杂性并没有改变。但是如果不考虑黑盒子内部的处理，系统复杂性就可以降低到人类的可控范围内。此外，黑盒子内部的处理无论怎么变化，如果输入和输出不发生变化，那么就对外部没有影响，所以这种扩展特性是我们非常希望获得的。

针对程序控制流的复杂问题，结构化编程采用了限制和抽象化的武器解决问题。结果证明，结构化程序设计是成功的，并且这种方法已经有了稳固的基础。现在几乎所有的编程语言都支持结构化编程，结构化编程已经成为了编程的基本常识。

2.2.4 2.2.4 数据抽象化

然而，程序里面不仅包括控制结构，还包括要处理的数据。结构化编程虽然降低了程序流程的复杂性，但是随着处理数据的增加，程序的复杂性也会上升。面向对象编程就是作为对抗数据复杂性的手段出现的。

前面已经介绍过了，世界上第一个面向对象和编程语言是 Simula。随着仿真处理的数据类型越来越多，分别管理程序处理内容和处理数据对象所带来的复杂性也就越来越高。为了得到正确的结果，必须保持处理和数据的一致性，这在结构化编程中是非常困难的。解决这一问题的方案就是数据抽象技术。

数据抽象是数据和处理方法的结合。对数据内容的处理和操作，必须通过事先定义好的方法来进行。数据和处理方法结合起来成为了黑盒子。

举一个栈的例子。栈是先入后出的数据存储结构[9]。比如往快餐托盘中叠加地摞放食品（见图 2-8）。栈只有两种操作方法：入栈（push），向栈中放入数据；出栈（pop），把最后放入的数据拿出来。

[8]计算器是黑盒子的一个例子。输入数字后，计算结果在液晶屏上显示出来，而内部是怎样计算的我们并不知道。也有可能是里面的小人的打算盘哦。

[9]队列是和栈相似的数据结构，是先入先出的。

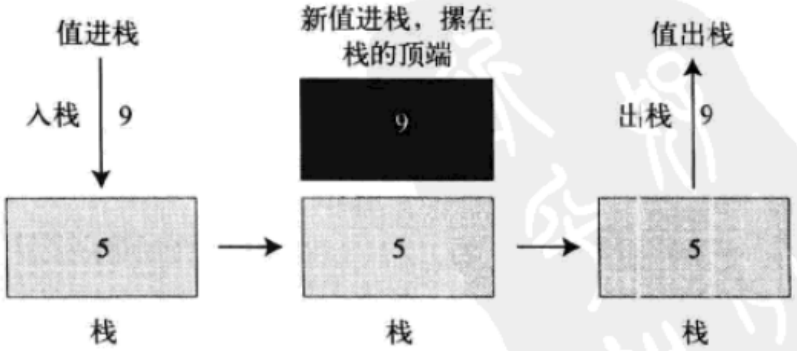

图 2-8 栈的构造

p/图 2-8.png p/图 2-8.bb

我们用 Ruby 来写这个栈[10]。图 2-9 中使用了抽象的数据结构，栈的操作只有 push 和 pop。别的方法是无法访问栈内数据的。图 2-10 中则没有使用抽象的数据结构，而是用数组索引来实现栈的操作。和图 2-9 相比，哪个更简单是显而易见的。

```
#用Stack.new 生成新的栈
stack = Stack.new
#对stack进行push操作
stack.push(5)
stack.push(9)

#用Stack的pop方法取出数据
puts stack.pop() #显示9
puts stack.pop() #显示5
```

图 2-9 用 Ruby 写的栈的操作

```
#用数组实现的栈的操作
stack = []
# 数组的先头位置
sp= 0

stack[sp]= 5
```

[10]在标准的 Ruby 中没有定义栈 (stack) 这种数据结构。如果要执行图 2-9 所示的程序，那么图 2-11 所示的 stack 定义是必须要的。图 2-9 所示只是作为一个例子来说明抽象数据的操作方法。

```
sp +=1
stack[sp]=9
sp +=1


sp -=1
puts stack[sp]
sp -= 1
puts stack[sp]
```

图 2-10 用数组实现图 2-9 的程序

图 2-9 的程序有几点优于图 2-10 的程序。第一，图 2-10 的程序暴露了“数组和下标”这一内部构造，而图 2-9 则把内部构造隐藏到了 stack 这一数据结构里。利用图 2-9 的方法，使用栈的人并不需要关心栈是如何实现的，即使将来因为什么事情而改变了栈的内部实现方式，也不需要对使用栈的程序做任何修改。

另外一点是图 2-9 所示的方法很容易理解。比如数据的 push 操作，在图 2-9 中是：stact.push(5)

在图 2-10 中是：

stack[sp]=5

sp += 1

图 2-9 中可以直接表现 push 这个操作。对数据进行操作的一方，并不需要知道图 2-10 中的处理细节，而只对“要做什么”感兴趣。所以隐藏了处理细节的程序会变得更加明确，实现目的也更清晰。

不仅是操作方法容易理解，抽象数据也是能够对特定的操作产生反应的智能数据。使用抽象数据可以更好地模拟现实世界中各种活生生的实体。

有了数据抽象，程序处理的数据就不再是单纯的数值或文字这些概念性的东西，而变成了人脑容易想象的具体事物。而代码的“抽象化”则是把想象的过程“具体化”了。这种智能数据可以模拟现实世界中的实体，因而被称作“对象”，面向对象编程也由此得名。