目录

1 前言部分

1.1 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

1.2 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。

1.3 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 |

列出类/module 的常量	Module#constants
获取常量值	Module#const$_{get}$
设置常量值	Module#const$_{set}$
删除常量	Module#remove$_{const}$
列出类变量	Module#class$_{variables}$
获取类变量值	Module#class$_{variableget}$
设置类变量值	Module#class$_{variableget}$’
删除类变量	Module#remove$_{classvariables}$
定义类方法	Module#define$_{method}$
删除类方法	Module#remove$_{method}$
解除类方法定义	Module#undef$_{method}$
给类方法赋予别名	Module#alias$_{method}$
包含模块	Module#include
获取父类	class#superclass

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

1.4 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

1.5 DONE 前言 <2016-08-30 二 14:10>

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

2 DONE 第一章我为什么开发 Ruby

2.1 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

[1]厚厚是

情。

2.1.1 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。

2.1.2 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。

2.1.3 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。

[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。————编者注可以使用多重继承的编程语言，不受单一继承的不自然的限制。例如，只提供单一继承的 Smalltalk 语言，它的类库因为单一继承而显得很不自然。p/图 2-13.png p/图 2-13.bb

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。

«««< Updated upstream 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。

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){
```

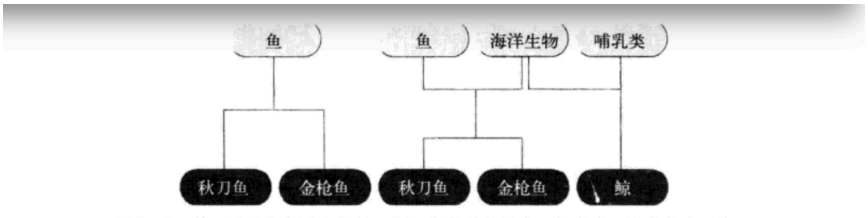


图 2-13 单一继承和多重继承的区别。在多重继承中，每个类可以有多个父类

```
        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”。

2.1.4 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”。

2.1.5 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。

2.1.6 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。===== 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。

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

3 第二章面向对象

3.1 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？

3.1.1 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
```

3.1.2 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

[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)。正如上一节中说明的，如果把继承作为抽离出程序的共通部分的一个抽象化手段来考虑，那么从一个类中抽象化（抽出）的部分只能有一，这个假定会给编程带来很大的限制。因此，多重继承的思想就这样产生了。单一继承和多重继承的区别仅仅是父类的数量不同。多重继承完全是单一继承的超集，可以简单地看做是单一继承的一个自然延伸（图 2-13）。

[4]jddjjd

果有 C 编译器，也可以从源程序来安装 Ruby。

3.1.3 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。

---

[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。但最后一句话严格来说并不完全正确。结构化和抽象化，意味着把共 ceep 部分提取出来生成父类的自底向上的方法。如果继承是这样诞生的话，那么最初，有多个父类的多重继承。

[6]单一继承 (single inheritance) 是指只能有一个父类 (super class) 的继承，也称为单纯继承。有多个父类的继承称为多重继承 (multiple inheritance)。

[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。上一节讲解了面向对象编程的三大原则 (多态性、数据抽象和继承) 中的继承。如前所述，人们一次能够把握并记忆的概念是有限的，为解决这一问题，就需要用到抽出类中相似部分的方法 (继承)。继承是随着程序的结构化和抽象化自然进化而来的一种方式。

3.1.4 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。

3.1.5 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。

3.1.6 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。

3.1.7 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
```

3.1.8 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
```

3.2 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]。

3.2.1 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

[7] ddd

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。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。

3.2.2 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。

3.2.3 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

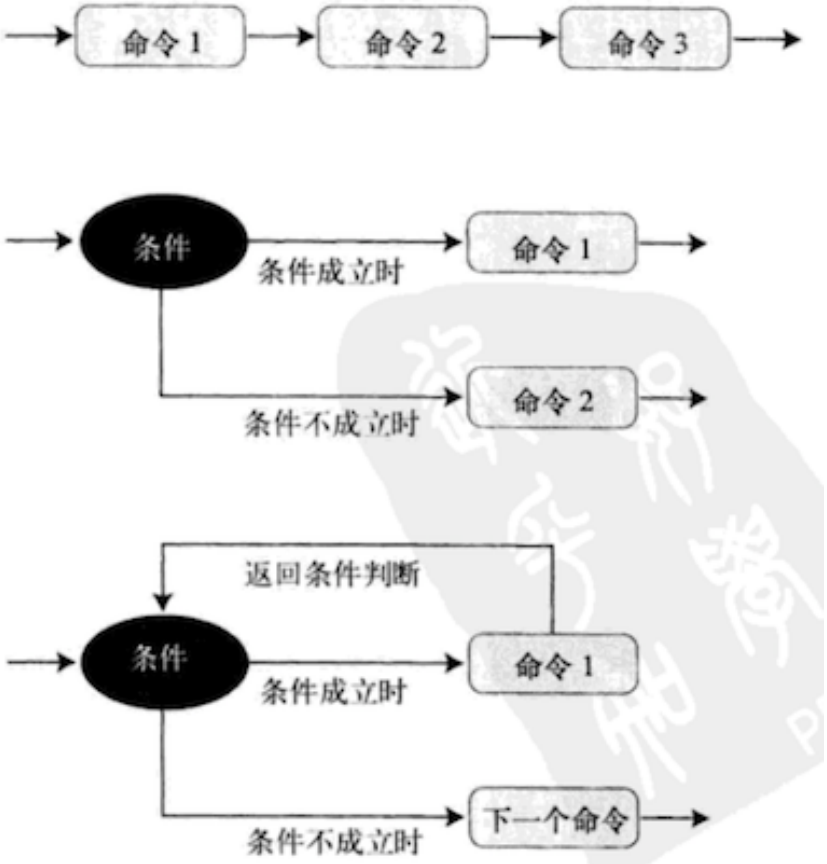


图 2-7 顺序、分支和循环的处理方法

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。

3.2.4 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

[8] b22

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。

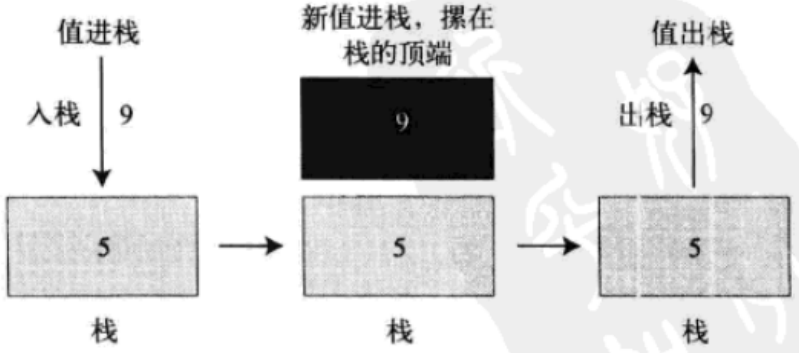


图 2-8 栈的构造

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
```

[9]队列是和栈相似的数据结构，是先入先出山的。

[10]a223

```
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
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。

3.2.5 2.2.5 雏形

出现在程序中的对象，通常具有相同的动作。以交通仿真程序为例，程序中有表示车和信号的对象。虽然同样的对象具有相同的性质，但是位置、颜色等状态各有不同。

从抽象的原则来说，多个相同事物出现时，应该组合在一起。这就是 DRY 原则 (即 Don’t Repeat Yourself)。

我们已经看到，程序的重复是一切问题的根源。重复的程序在需要修改的时候，所涉及的范围就会更广，费用也就更高。当多个重复的地方都需要修改时，哪怕是漏掉其中之一，程序也将无法正常工作。所以重复降低了程序的可靠性。

进一步说，重复的程序是冗余的。人们解读程序、理解程序意图的成本也会增加。让我们再看看代码重复的图 2-10 和没有代码重复的图 2-9，显然图 2-9 的程序更容易理解。请记住，计算机是不管程序是否难以阅读，是否有重复的。然而，开发人员要阅读和理解大量的程序，所以程序的可读性直接关系到生产力。重复冗长的程序会降低生产力。复制和粘贴程序会导致重复，应该尽量避免。

让我们再回到对象的话题上。同样的对象大量存在的时候，为了避免重复，可以采用两种方法来管理对象。

一种是原型。用原始对象的副本来作为新的相同的对象。Self、Io 等编程语言采用了原型。有名的编程语言用原型的比较少，很意外的是，JavaScript 也是用的原型。另外一种是模板。比方说我们要浇注东西的时候，往模板里注入液体材料就能浇注出相同的东西。这种模板在面向对象编程语言中称为类 (class)。同样类型的对象分别属于同样的类，操作方法和属性可以共享。

跟原型不同，面向对象编程语言的类和对象有明显的区别，就像做点心的模具和点心有区别一样整数的类和 1 这个对象、狗类和名字是 poochy 这条狗也都是有区别的。为了清晰的表明类和对象的不同，对象又常常被称作实例 (instance)。叫法虽有不同，但实例和对象是一样的。

在 Ruby 面向对象编程语言[11]中，类用关键字 class 来声明。图 2-9 中的栈，就是 Stack 类。Stack 类的定义如图 2-11 所示。

class 后面是类名。在图 2-11 中，class 后面就是 Stack。Ruby 规定类名称的第一个字母必须大写。类定义的最后用 end。在 Stack 这个类中，定义了 initialize、push、pop 这三个方法。

图 2-9 的程序第二行调用了 intialize 这个初始化方法。

stack = Stack.new

```
class Stack
  def initialize
    @stack=[]
    @sp=0
  end

  def push(value)
    @stack[@sp]=value
    @sp+=1
  end

  def pop
    return nil if @sp==0
    @sp-=1
    return @stack[@sp]
  end
end
```

图 2-11 Stack 类的实现

[11]Ruby 也可以用于原型 (prototype) 面向对象编程

每次生成 Stack 对象的时候，都要调用 initialize 这个初始化方法。

在图 2-11 的初始化方法中，@stack（实际保存栈数据的数组）和 @sp（数组下标）这两个变量被初始化。在 Ruby 中以“@”开头的变量用来保存每个对象中分别独立存在的值，也称为实例变量。如果你创建了多个栈对象，那么每个对象里面都分别有自己独立的 @stack 和 @sp 这两个变量。

push 和 pop 是操作栈的方法。在图 2-10 中不过是罗列了对栈的操作步骤罢了。

图 2-11 的 initialize 是对类定义的操作对象的内部数据进行初始化的“方法”。

为了简化说明，图 2-11 的例子中没有检查数据的范围。事实上，程序中需要检查下标是否为负值等。

3.2.6 2.2.6 找出相似的部分来继承

随着软件规模的扩大，用到的类的个数也随之增加，其中也会有很多性质相似的类。这就违背了我们之前强调多次的 DRY 原则。程序会变得重复而且不容易理解。修改程序的代价也会变高，生产力则会降低。所以，如果有把这些相似的部分汇总到一起的方法就好了。

继承就是这种方法。具体来说，继子就是在保持即有类的性质的基础上生成新类的方法。原来的类称为父类，新生成的类称为子类。子类继承父类所有的方法，如果需要也可以增加新的方法。子类也可以根据需要重写从父类继承的方法。

图 2-12 演示了 FixedStack 这个类，它继承了图 2-11 中的 Stack 类。类名后面的“<Stack”指的是父类。它说明了 FixedStack 是 Stack 的子类，继承了 Stack 类的方法和属性。

FixedStack 类重写了 initialize 和 push 这两个方法。这两个方法都调用了 super 方法，这表明在子类的方法中也调用了父类的具有相同名字的方法。比如在 FixedStack 类的 initialize 方法中也调用父类 Stack 的 initialize 方法。利用这种方式，我们可以只改变子类方法的动作，而不会对父类方法产生任何影响。

initialize 方法在对象初始化时被调用。如果像下面的程序一样，在调用

initizlize 方法时传入参数 10，那么栈对象的实例变量 @limit 就会被设置为 10，它是栈中元素个数的上限。

stack=FixedStack.new(10)

在图 2-12 中，程序末尾追加了并不把栈顶元素弹出栈而只是引用栈顶元素的方法 top。top 在父类中交没有定义，这是一个在子类中追加方法的例子。

```
class FixedStack<Stack
  def initialize(limit)
    super()
    @limit=limit
  end

  def push(val)
    if @sp>=@limit
     puts "over limit"
     return
    end
    super(val)
   end

  def top
    return @stack[-1]
  end
end
```

图 2-12 用类来继承图 2-11 中类的例子。

像图 2-12 这样，利用现有的类派生新类的方法称为“差分编程法”(difference programming)。通过抽象把共通的部分提取出来生成父类，与利用已有的类来生成新类，是同一方法的两种不同的表现形式。前者称为自底向上法，后者称为自顶向下法。

Ruby 跟多数编程语言一样，一个子类只能有一个父类，这称为“单一继

承”。从自顶向下的方法来看，通过扩展一个类来生成新的类也是很自然的。

但是，从用自底向上的方法提取共通部分的角度来看，一个子类只能有一个父类的限制是太严格了。其实，在 C++、Lisp 等编程语言中，一个子类可以有多个父类，这称为“多重继承”。

3.3 2.3 多重继承的缺点

上一节讲解了面向对象编程的三大原则（多态性、数据抽象和继承）中的继承。如前所述，人们一次能够把握并记忆的概念是有限的，为解决这一问题，就需要用到抽出类中相似部分的方法（继承）。继承是随着程序的结构化和抽象化自然进化而来的一种方式。

但最后一句话严格来说并不完全正确。结构化和抽象化，意味着把共ceep 部分提取出来生成父类的自底向上的方法。如果继承是这样诞生的话，那么最初，有多个父类的多重继承[6]就会成为主流。

但实际上，最初引入继承的 Simula 编程语言，只提供单一继承。同样，在随后的很多面向对象编程语言中也都是这样的。因此我认为，继承的原本目的实际上是逐步细化。

3.3.1 2.3.1 为什么需要多重继承

单一继承只能有一个父类。有时候，大家会觉得这样的制约过于严格了。在现实中，一个公司职员同时也可能是一位父亲，一个程序员同时也可能是一位作家。

正如上一节中说明的，如果把继承作为抽离出程序的共通部分的一个抽象化手段来考虑，那么从一个类中抽象化（抽出）的部分只能有一，这个假定会给编程带来很大的限制。因此，多重继承的思想就这样产生了。单一继承和多重继承的区别仅仅是父类的数量不同。多重继承完全是单一继承的超集，可以简单地看做是单一继承的一个自然延伸（图 2-13）。

可以使用多重继承的编程语言，不受单一继承的不自然的限制。例如，只提供单一继承的 Smalltalk 语言，它的类库因为单一继承而显得很不自然。

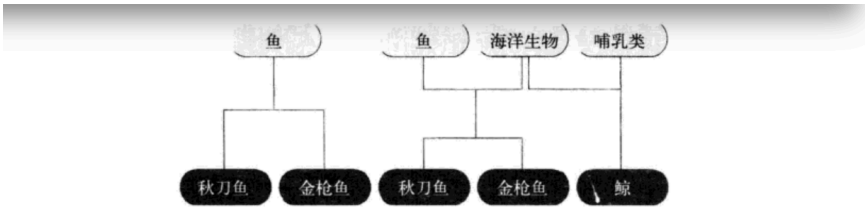


图 2-13 单一继承和多重继承的区别。在多重继承中，每个类可以有多个父类

p/图 2-13.png p/图 2-13.bb

Smalltalk 语言中定义输入输出的 Stream 类有 3 个子类。其中，ReadStream 是输入类，WriteStream 是输出类，ReadWriteStream 是输入输出类。ReadWriteStream 具有 ReadStream 和 WriteStream 两个类的功能，但是由于 Smalltalk 是单一继承的，所以 ReadWriteStream 不能同时从这两个类继承。

结果是 ReadWriteStream 继承了 WriteStream 这个类，然后再把 ReadStream 的程序复制过来，从而实现 ReadStream 的功能（参见图 2-14）。从程序维护的观点来看，程序复制是必须禁止的。由于单一继承的限制而导致的程序复制是我们不愿意看到的。

从另外的角度来看，如果有多重继承的话，那么很自然地从 ReadStream 和 WriteStream 继承就可以生成 ReadWriteStream（参见图 2-15）。

3.3.2 2.3.2 多重继承和单一继承不可分离

经过对多重继承和单一继承这样一比较，单一继承的特点就很明显了。

- 继承关系单纯

单一继承的继承关系是单纯的树结构，这样有利有弊。类之间的关系单纯就不会发生混乱，实现起来也比较简单。但是，如刚才的 Smalltalk 的 Stream 一样，不能通过继承关系来共享程序代码，导致了最后要复制程序。

对需要指定算式和变量类型的 Jave 这样的静态编程语言来说，单一继承还有一个缺点，我们将在后面说明。p/图 2-14.png p/图 2-14.bb

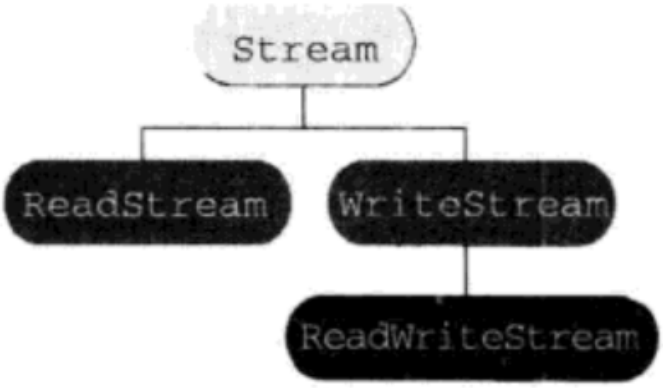


图2-14 单一继承的问题。ReadWriteStream 只能有一个父类，即 WriteStream，而不能同时继承 ReadStream

p/图 2-

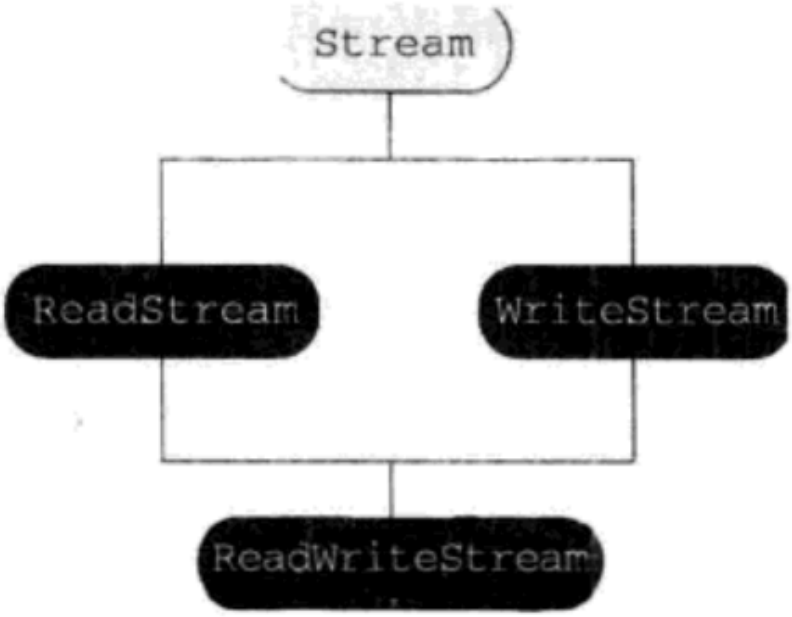


图 2-15 用多重继承的解决方法。和图 2-14 不同，ReadWriteStream 类可以继承两个父类

15.png p/图 2-15.bb

多重继承的特点正好相反。多重继承有以下两个优点：

- 很自然地做到了单一继承的扩展；
- 可以继承多个类的功能。

单一继承可以实现的功能，多重继承都可以实现。但是，类之间的关系会变得复杂。这是多重继承的一个缺点。

3.3.3 2.3.3 goto 语句和多重继承比较相似

前面我们讲到了结构化编程，说明了与其用 goto 语句在程序中跳来跳去，还不如用分支或者循环来控制程序的流程。分支和循环可以用 goto 语

句来实现，单纯的分支和循环组合起来不能直接实现的控制也可以用 goto 语句来实现。goto 语句具有更强的控制力。

goto 语句的控制能力虽然很强，但是我们也不推荐使用。因为用 goto 语句的程序不是一目了然的，结构不容易理解。这样的流程复杂的程序被称为“意大利面条程序”。

多重继承也存在同样的问题。多重继承是单一继承的扩展，单一继承可以实现的功能它都可以实现。用单一继承不实现的功能，多重继承也可以实现。

但是，如果允许从多个类继承，类的关系就会变得复杂。哪个类继承了哪个类的功能就不容易理解，出现问题时，是哪个类导致的问题也不容易判明。

这样混合起来发展的继承称为“意大利面条继承”。当期也不能说所有的多重继承都是意大利面条继承，但是使用时格外小心是必要的。多重继承会导致下列 3 个问题。

. 结构复杂化

如果是单一继承，一个类的父类是什么，父类的父类又是什么，都很明确，因为只有单一的继承关系。然而如果是多重继承的话，一个类有多个父类，这些父类又有自己的父类，那么类之间的关系就很复杂了。

. 优先顺序模糊

具有复杂的父类的类，它们的优先关系一下子很难辨认清楚。比如图 2-16 中的层次关系，D 继承父类方法的顺序是 D、B、A、C、Object 还是 D、B、C、A、Object, 或者是其他的顺序，很不明确。确定不了究竟是哪一个。相比之下，单一继承中类的优先顺序是明确了然的。

. 功能冲突

因为多重继承有多个父类，所以当不同父类中有相同的方法时就会产生冲突。比如在图 2-16 中，当类 B 和类 C 有相同的方法时，D 继承的是哪个方法就不明确了。存在两种可能性。

3.3.4 2.3.4 解决多重继承的问题

上面说明了多重继承的问题。但是像 Smalltalk 的 Stream 的例子一样, 如果没有多重继承的话，有些问题还真是难以解决。

再进一步看，继承做为抽象化的手段，是需要实现多重继承功能的。在抽取类的共通功能的时候, 如果一个类只允许抽出一个功能, 那么限制就太多了。p/图 2-16.png p/图 2-16.bb

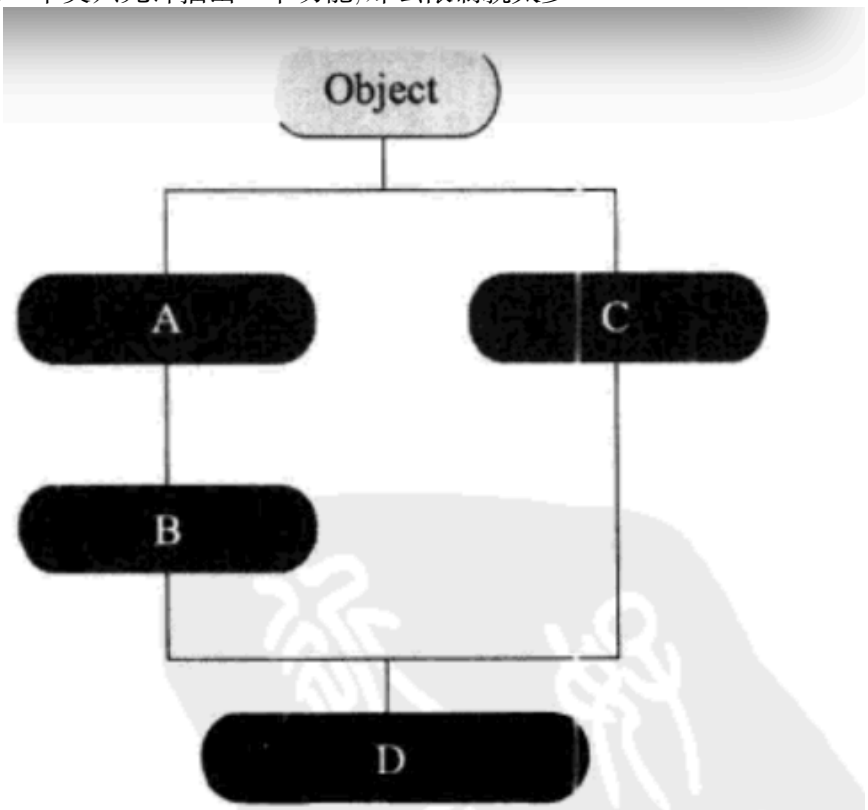


图 2-16 多重继承的优先顺序，方法调用的优先顺序不明确

既想利用多重继承的优点，又要回避它可能会带来的问题，那我们就需要寻找解决问题的方法。结构化编程解决 goto 问题的原则是，用 3 种有限制功能的控制语句来代替自由度太高的 goto 语句。这 3 种控制语句虽然有

限制，但是用它们的组合可以实现任意算法。像这样引入有限制的多重继承应该是一个好的方法。

没错，受限制的多重继承，这个解决或者改善多重继承问题的方法出现了，它在 Jave 编程语言中被称为接口（interface），在 Lisp 或者 Ruby 中是 Mix-in。下面我们看看这些功能是如何克服上述缺点的。

3.3.5 静态语言和动态语言的区别

我们从 Java 的接口说起

在说明接口之前，首先讲一下像 Java 这样的面向对象编程语言和多重继承。

从在的方面来看，编程语言可以分为静态语言和动态语言两种。像 Java 这样规定变量和算式类型的语言称为静态语言。

在静态语言中，不能给变量赋不同类型的值，因为那样会导致编译错误。由于在编译时已经排除了类型不匹配的错误，所以在执行时就不会再发生这样错误了。不通过执行就可以发现类型不匹配这样的错误是静态语言的一个优点。

String str;

str="abc"; //没有问题

str=2; //编译错误

面向对象编程语言大都用类来指定变量类型。上面例子用的就是 String 这个类。但是在使用面向对象编程语言时，像上面的例子那样，只能将特定类的对象（该类的实例）赋给变量的限制的确又太严格了，因为这样的话就没有多态性了。如果只能给一个变量赋值同类对象，就不可能根据对象的类自动选择合适的处理方式（多态性）。

3.3.6 2.3.6 静态语言的特点

为解决这一问题，静态类型面向对象编程语言被设计成这样，当给一个类变量赋值时，既可以用这个类的对象来赋值，也可以用这个类的子类对象来赋值。这样就可以实现多态性。

请看图 2-17 中和程序。这是一个用 Java 风格的静态语言来定义多边形类的例子。最后出现的 poly 是 polygon 类的一个变量，所以通过 poly 应该可以调用 polygon 类的方法（比如“面积”方法）。但实际上，poly 这个变量的值是 polygon 子类 Rectangle 的对象，所以通过 poly 调用的就是 Rectangle 的方法。当然，如果调用的方法只在 polygon 中定义而没有在 Rectangle 中定义，那就会调用 Polygon 中定义的方法。

但是反过来说，在程序中 poly 就是 Polygon 类的变量，即使它的值明明是 Rectangle 类对象，用 poly 这个变量也不能调用 Rectangle 类中固有的方法（比如“边长”）。

```
//多边形类
class Polygon
{
    float 面积 ( ) {……}
    int 顶点数 ( ) {……}
    ……
};

//矩形类（继承多边形类）
class rentangle extends Polygon
{
     float 面积 ( ) {……}//再定义面积计算方法
     int 边长 ( ) {……} //矩形类特有的方法
     ……
} ;

Polygon poly;
poly=new Rectangle();
```

图 2-17 父子关系的类的示例，变量不能调用子类特有的“边长”方法

换个说法就是，变量只是实际赋值对象的一个小观测窗口。即使作为变量值的对象有很多方法，但在使用这个变量来调用方法时，只能调用该变量

类型“知道”的方法。

如果变量 poly 调用“边长”方法的话，静态语言会毫不留情地报告编译错误。

而像 Ruby 这样没有类型定义的动态编程语言，是在程序执行时才来试着调用对象的方法，在实际对象没有可被物方法时程序才会报错。

3.3.7 2.3.7 动态语言的特点

动态语言允许调用没有继承关系的方法。比如说 Ruby 中定义了顺序取出某个元素的方法 each，数据和哈希表中都实现了这个方法。

```
obj.each {|x|
    print x
}
```

在静态语言中只能调用有继承关系的方法，数据、哈希表和字符串都能调用的方法只能是在它们共同的父类（恐怕就是 Object）中定义。

这是单一继承的一个缺点，以后会详细说明。

在静态语言中，如果要调用类层次中平行类的方法，那么必须要有一个可以表现这些对象的类型。如果没有这个类型，可调用的方法是非常有限的。由此我们看到静态语言中某种形式的多重继承是不可少的。

3.3.8 2.3.8 静态语言和动态语言的比较

静态语言和动态语言各有利弊。静态语言即使不通过执行也可以检查出类型是否匹配。在一定程度上，程序的一些逻辑错误可以被自动检测出来。

但是，逐个来定义算式和变量的类型又会使程序变得冗长。只有包含继承关系的类才会具有多态性。相对于动态语言来说，静态语言就显得限制过多，灵活性差。

动态语言则正好相反。程序中有没有错误只有执行了才会知道。从可靠性来看也许会让你感觉有些不安。程序中没有类型定义，这样程序会变得很简洁，但别人看起来或许会有点难懂。

但是，只有方法名一样，这些对象都可以以相同的方式去处理。也就是说不需要深层次探索类也可以开发程序。这样生产效率就会大大提高[12]。

3.3.9 2.3.9 继承的两种含义

像 java 这样的静态面向对象编程语言的变量，具有限制调用方法的功能。但实际上限制的是类有什么样的方法，而不是这个类是怎么实现的。

到现在为止我们一起都在讨论继承，其实包含两种含义。一种是“类都有哪些方法”，也就是说这个类都把持些什么操作，即规格的继承。

另外一种是, “类中都用了什么数据结构和什么算法”，也就是实现的继承。

静态语言中，这两者的区别很重要[13]。Java 就对两者有很明确的区分，实现的继承用 extends 来继承父类，规格和继承用 implements 来指定接口。

类是用来指定对象实现的，而接口只是指定对象的外观（都有哪些方法）。

Java 中，只允许用 extends 继承一个父类（实现的继承），所以类的继承是单一的。类的关系树和类库也就相对简单。

然而，implements 可以指定多个接口（规格的继承）。接口规定了要怎样处理该对象。

举个具体例子说明一下。我们来看看图 2-18 中 java.util.Collection 这个接口的类层次。java.util.Collection 是定义集合的接口，有 2 个接口来继承它，分别是按顺序存放元素的 java.util.List 和没有重复元素的 java.util.Set。也就是说，实现了 java.,util.List 或 java.util.Set 的对象也可以被当做 java.util.Collection 来处理。

---

[12]这种宽松的编程机制称为 Duck Typeing(鸭子类型检测）

[13]动态编程语言中，区分规格的继承和实现的继承意义不大。即使没有继承关系，方法也可以自由地调用。

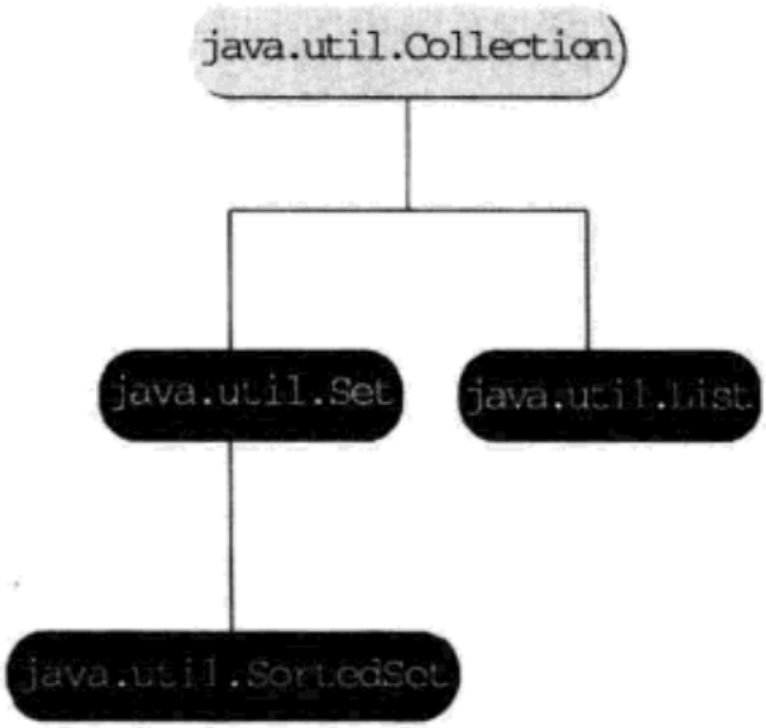


图 2-18 接口的类层次，java.util.Collection 的例子

p/图 2-18.png p/图 2-18.bb

接口对实现没有任何限制。也就是说，接口可以由跟实现的继承没有任何关系的类来实现。也就是说，实现这一接口的类可以继承任何其他类。例如在 java.util 包中，作为 java.util.List 的实现，既提供用数组实现的 java.util.ArrayList，也提供用双向链表实现的 java.util.LinkedList。这些类都直接继承 Object 类。

3.3.10 2.3.10 接口的缺点

关于规格继承和实现继承的区别，很久以前就有论文进行了相关的探讨。但在众多得到广泛运用的编程语言中，Java 是第一个实现这种功能的。这可以说是 Java 对多重继承问题的解答。既实现了静态语言的多重继承性，又避免了多重继承的数据构造的冲突和类层次的复杂性。

但是，我们并不能说接口是解决问题的完美方案。接口也有不能共享实现的缺点。

为了解决多重继承的问题，人们允许了规格和多重继承，但是还是不允许实现多重继承。针对这一点，我们不太好再说什么，但作为用户，就是觉得不方便。Java 推荐的解决共享实现问题的方案是，在单一继承的前提下，使用组合模式（Composite）来调用别的类实现的共通功能。

本来只是为了跨越继承层次来共享代码，现在却需要另外生成一个独立对象，而且每次方法调用都要委派给那个对象，这实在是不太合理，而且执行的效率也不高。

3.3.11 2.3.11 继承实现的方法

和静态语言 Java 不同，动态语言本来就设有继承规格这种概念。动态语言需要解决的就是实现的多重继承。

动态语言是怎么解决这一问题的呢？Lisp、Perl 和 Python 都提供了多重继承功能，这样就不存在单一继承的问题了。在这些语言中，使用多重继承时请千万小心。

3.3.12 2.3.12 从多重继承变形而来的 Mix-in

Ruby 采用了和 Java 及其他动态语言都不同的方法。Ruby 用 Mix-in 模块来解决多重继承的问题。

Mix-in 是降低多重继承复杂性的一个技术，最初是在 List 中开始使用的。实现 Mix-in 并不需要编程语言提供特别的功能。Mix-in 技术按照以下规则来限制多重继承。

- 通常的继承用单一继承

- 第二个以及两个以上的父类必须是 Mix-in 的抽象类

Mix-in 类是具有以下特征的抽象类。

- 不能单独生成实例
- 不能继承普通类

按照这个原则，类的层次具有和单一继承一样的村结构，同时又可以实现功能共享。实现功能共享的方法是把共享的功能放在 Mix-in 类里面，然后把 Mix-in 类插入到树结构里面。相对于 Java 用接口方法解决规格继承的问题，那么 Mix-in 可以说是解决了实现继承的问题。

我们看一个 Mix-in 的具体例子。针对图 2-14、图 2-15 中的 Smalltalk 的 Stream 问题，图 2-19 显示的是用 Mix-in 构建的一个相同结构。

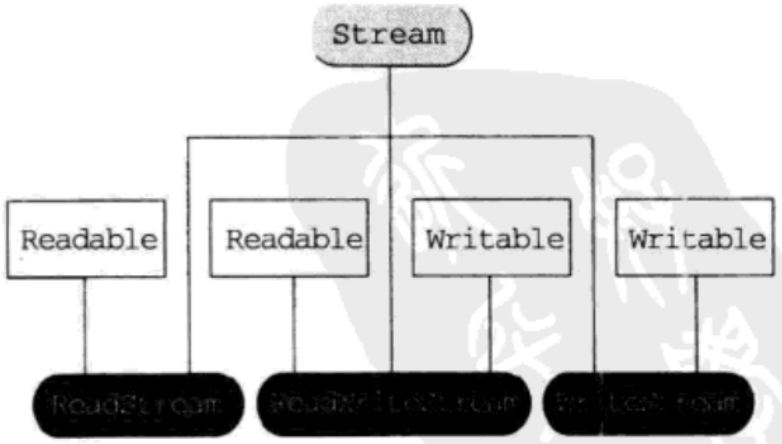


图 2-19 用 Mix-in 实现 Stream 类。既保持了类层次的树结构，又避免了复制程序

p/图 2-19.png p/图 2-19.bb

在使用 Min-in 的类结构中，Stream 只有 3 个子类。在此基础上，实际的输入/输出处理用 Readable(输入) 和 Writable(输出) 这两个 Mix-in 类实现。3 个子类通过继承 Mix-in 类而分别实现了输入、输入以及输入和输出的功能。

从 Stream 的类层次来看，父类是 Stream，负责输入输出的是 ReadStream、WriteStream 和 ReadWriteStream 这 3 个子类，它们形成了非常清晰的树结构。层次很简单，没有变成网状结构。而且，由于 Mix-in 类实现了共通功能，从而避免了复制代码。

和一般的多重继承相比，Mix-in 是使类结构变得简单的优秀技术。使用 Min-in 规则来限制多重继承，实际上也可以说是“驯服”了多重继承。

这和结构化编程用分支和循环来限制随意的 goto 语句是一样的。Mix-in 可以应用于所有的多重继承编程语言中，因此，掌握这个技术是非常有必要的。

3.3.13 2.3.13 积极支持 Mix-in 的 Ruby

和其他直接引入多重继承的编程语言相比，Ruby 具有直接支持 Mix-in 的特点。在 Ruby 中，Mix-in 的单位是模块 (module)。模块具有 Mix-in 的特性，即：

- 不能生成实例；
- 不能从普通类继承。

下面我们看看在 Ruby 中是怎样使用 Mix-in 的。图 2-20 演示了 Ruby 是怎样实现图 2-19 的 Stream 类的定义的。

模块用关键字 module 来定义，这和定义类用关键字 class 相似，但是不能指定它的父类。其中方法等的定义与类也是一样的。

在类中通过 include 可以继承模块中的方法。因为是继承而不是复制，所以当类中有同样的方法时，类中的方法就会优先执行。

关于继承的各方面内容，我们都总结到了表 2-2 中。

表 2-2 与继承有关的内容

用语	内容
单一继承	只能有一个父类，单纯但存在几个问题
多重继承	可以有多个父类，解决了单一继承的问题（面向对象的编程语言需要某种形式

静态语言	区分规格的继承和实现的继承
动态语言	只有实现的继承
规格多重继承的问题	Java 的接口可以解决
实现多重继承的问题	Mix-in 可以解决
Mix-in	所有支持多重继承的语言都可以考虑使用
Ruby 的 Mix-in	强制利用模块，积极解决多重继承的问题

```
# Stream 类， Object的子类
class Stream <Object

    # 这里的定义省略
     …… 

end

#输入用Mix–in
module Readable

   # 定义输入用的方法
   def read
    …… 
   end

#输出用Mix–in
module Writable

   #定义输出用的方法
```

```
  def write(str)
    ……
  end

end

# 输入用Stream,Stream 的子类
class ReadStream < Stream

  #继承输入用的Mix-in
  #Ruby称谓include
  include Writable

end

#输入输出用Stream,Stream 的子类
class ReadWriteStream <Stream

  #继承输入用Mix-in
  include Writable

end
```

图 2-20 用 Ruby 实现图 2-19 的 Stream 类的定义

3.4 2.4 两个误解

本节将说明一下关于对面向对象的误解。

作为一个很早就接触面向对象编程语言的爱好者，我写过关于面向对象的文章，开发了面向对象编程语言 Ruby。我觉得自己为让更多的人都能够熟悉面向对象编程语言作出了贡献。我骄傲的认为，Ruby 比 Smalltalk 更容易上手，比 Java 和 C++ 更容易实现面向对象编程，从而使人们更容易

理解面向对象的概念。

但是，在这个过程中，由于我的不成熟，可能会加深一些人对面向对象的误解。

在这些误解中，有两个是我很在意的。

一个误解是，对象是对现实世界中具体物体的反映，继承是对物体分类的反映。这个观点是错误的。我之前写过的《面向对象编程语言 Ruby》[14]中也用哺乳动物，比如狗和鲸等举过例子，可能也加深了这种误解。

另一个是，多重继承是不好的。这个观点也是错误的。这一误解好像还大都与“但 Mix-in 不错”[15]的误解掺和在一起。

在解释之前我先表明一下正确的观点。关于多重继承，正确的理解应该昌，如果用得不好就会出问题。对 Mix-in 的理解应该是，Mix-in 只不过是实现多重继承的一个技巧而已。

Ruby 只支持 Mix-in 形式的多重继承。这是因为在当时 Mix-in 这种技术还不广为人知，我只是想把多重继承作为一种启蒙，并没有贬低多重继承的意思。这可能造成了有人认为多重继承不好的误解。曾有一个著名的青年学者因为 Ruby 的原因而误认为多重继承和 Mix-in 是不同的概念。这也是我要反省的一点吧。

当然，我也不觉得这些误解全都是我造成的。但是，为了减少这些误解，下面再讲解一下面向对象编程语言和多重继承。

3.4.1 2.4.1 面向对象的编程方法

从历史上看，从 20 世纪 60 年代末期到 70 年代，分别有几个不同领域都发展了面向对象的思想。比如数据抽象的研究、人工智能领域中的知识表现（框架模型）、仿真对象的管理方法（Simula）、并行计算模型（Actor）以及在结构化编程思想影响下而产生的面向对象方法。

---

[14]由松本行弘与石冢圭树合著，ASCII 出版社出版，ISBN 为 4756132545(http://www.ascil.co.books/detail/4-7561/4-7561-3254-5.html)。

[15]Mix-in 是 Ruby 中可以利用的一个抽象类。既具有单一继承的方法构成和优先顺序的明确性，可可以像多重继承一样从多个类继承。Mix-in 类不能用来生成实例，也不能继承普通类。

框架模型是现实世界的模型化。从这个角度来看, “对象是对现实世界中具体事物的反映”这个观点并没有错。

但是不管过去怎样，现在对面向对象最好的理解是，面向对象编程是结构化编程的延伸。

计算机最初出现时，对软件的要求是非常简单的，只是把人完成工作的步骤用汇编或者机器语言表现出来，编程并不是很难的工作。但是随着软件的复杂化，开发就变得越来越复杂。因为这个原因，Edsger Dijkstr[16]提倡把程序控制限制为以下 3 种的组合，使程序变得简单且容易理解。

（1）顺序————程序按照顺序执行。（2）循环————一定的条件成立时程序反复执行。（3）分支————条件满足时执行 A 处理，不满足时执行 B 处理。结构化编程基本上实现了控制流程的结构化。但是程序流程虽然结构化了，要处理的数据却并没有被结构化[17]。面对对象的设计方法是在结构化编程对控制流程实现了结构化后，又加上了对数据的结构化。

众多面向对象的编程思想虽不尽一致，但是无论哪种面向对象编程语言都具有以下的共通功能。

(1) 不需要知道内部的详细处理就可以进行操作（封装、数据抽象）。（2）根据不同的数据类型自动选择适当的方法（多态性）。

可以这样认为，以下两点在面向对象编程语言中是必不可少的。因为不必知道内部结构，所以可以把数据当做黑盒来操作。即使将来数据结构发生变化，对外部也没有影响。黑盒化是模块化折基本原则，面向对象编程语言将每一类数据都当做黑盒处理。

多态性是根据不同的⊠自动选择适当的处理。这就不需要由人来根据不同的数据类型对处理进行分支了。如果没有多态性，那么程序中就会到处是分支处理。这也就意味着，变更和追加数据类型会变得非常困难。

在面向对象分析和面向对象设计领域，有些观点还不尽一致，但如果只谈面向对象编程，就可以认为封装和多态性是提高生产率的技术。

如果把面向对象编程看做是对结构化编程的扩展，那么对象是否是现实

[16]Edsger Wybe Dijkstra 是荷兰的计算机科学家。他提倡结构化编程，发起了减少使用 goto 语法的运动。他也是解决图论中最短路径问题的 Dijkstra 方法的研究者。

[17]20 世纪 70 年代, Michael A.Jackson 在开发的 Jackson Structured Programming(JSP) 中尝试了数据的结构化。但是，JSP 中结构化对象是操作数据的流程而不是数据。

世界中具体物体的反映就不重要了。实际上，面向对象编程语言中的对象，像字符串、数组和范围等，很多都没有现实世界中的具体物体与之对应。即使现实世界中有具体物体与之对应，对象也只是描述现实物体的某一侧面的抽象概念而已。比如猫有颜色、血统等很多属性，而程序中的对象并不需要把这些属性都考虑进去。程序只是处理抽象数据的。

3.4.2 2.4.2 对象的模板 = 类

很多面向对象编程语言都具有类和继承这两个基本特性。利用这两个特性，我们可以高效地把抽象的数据通过类封装起来。

类是对象的模板，相当于对象的雏形。在具有类功能的面向对象编程语言[18]中，对象都是由作为雏形的类来生成的，对象的性质也是由类来决定的。通过类可以把同一类的对象管理起来。

图 2-21 显示的是 Ruby 中对类的定义。我一般情况下是用阿猫阿狗来举例的，但为了避免误解，这次用数据结构的栈来举例。

栈是 LIFO(后入先出）的数据结构。可以按照从后向前的顺序把里面的数据取出来。图 2-21 的程序里定义了两个栈，s1 和 s2，分别对它们放入和取出数据。两个栈都是由相同的类生成的对象，操作方法都一样。但是，它们的数据都是独立的，交互对两个栈进行操作也不会破坏彼此的数据。

我们可以把面向对象编程语言的类看做是结构化编程语言的结构体或记录的扩展。不同的是，类里面不仅有被称为成员或字段的数据，而且还有对这些“数据块”进行操作的方法。

对于只是把数据组织在一起的结构体，我们能做的只是取出或者更新成员变量的值。而类中定义有成员函数（也称为方法），可以调用这些方法来处理类的对象。

例程能够把一系列的处理步骤组织在一起，把处理的内容黑盒化，是个很有用的工具，而类则是把数据⊠盒化的工具。由于对类内部数据的操作都是通过类的方法来实现的，所以内容数据结构即使在以后发生变化，对外部

[18]在有些编程语言中，类并不是必需的。相对于基于类的面向对象语言来说，那些类不是必需的面向对象语言称为基于原型的语言。有代表性的基于原型的语言有 Self.Ajax 背后的 JavaScript 也是基于原型的语言。

也没有影响。这和例程把处理⊠盒化之后，内部算法变化对外部没有影响是同样的道理。

像这样不用考虑内部处理的黑盒化也被称为抽象化。是降低程序复杂度的有效方法。

3.4.3 2.4.3 利用模块的手段 = 继承

类以数据为核心，把与之相关的处理也都集中到一起。这样，模块之间一些具有共通性质的内容就会重复出现，从而违背了禁止重复的 DRY 原则[19]。

避免重复的方法是继承。那些具有相同性质的类可以从拥有共通性质的类中“继承”这些共通的部分。不单是可以继承，还可以替换，追加其中不同的部分，从而生成新的类。

人这个角度来看，类是模块，继承就是利用模块的方法。继承的思想好像有其现实的知识基础，但是把它看做纯粹的模块利用方法则更恰当。

因为继承只不过是抽象的功能利用方法，所以不必把对继承的理解束缚在“继承是对现实事物的分类的反映”。实际上这样的想法反而是非常妨碍了我们对继承的理解。

关于继承，规格继承和实现继承的区别也是非常重要的话题。规格就是从部看到的类的功能，这样的继承是规格继承。实现继承是批继承功能的实现方法。

传统面向对象编程语言是一下子把规格和实现都继承下来，在最近的编程语言中，有的是把这两种继承分开了。比如 Java 里的接口就是规格继承，而在 Sather 编程语言中，规格继承和实现继承被完全分离开了。

3.4.4 2.4.4 多重继承不好吗

在早期的面向对象编程语言中，基功能被继承的类（基类或者父类）被限定为一个，这称为单一继承（或者单纯继承）。

[19]DRY(Don’t Repeat Yourself) 原则就是彻底避免重复。这一原则对提高程序开发的效率和可靠性非常有效。

把单一继承自然地扩展，一个类可以继承多个类的功能，就成为了多重继承。在自然界中也是一样，家长不只有一个。一个程序员同时也可能是一位父亲，同时具有多个角色。所以从单一继承到多重继承是很自然的。

但是，像 Java 和 Smalltalk 这样，不支持多重继承的编程语言还有很多，在程序员中多重继承好像也不是很普及，其中认为“多重继承是不好的东西”的人并不少。

单一继承的类之间的关系是很单纯的树结构。但是对多重继承而言，类之间的关系却是复杂的网状结构。

正因为如此，多重继承在一部分开发者当中的评价并不好，但是考虑到程序的生产力，多重继承不是必要的。

对于像 Java 或者 C++ 这样需要指定变量类型的静态语言来说，父类类型的变量可以用子类的对象来赋值。如果用子类以外的对象来赋值的话，就会发生编译错误。所以可以说这既实现了多态性，又实现了对变量类型的检查，是一个很好的想法[20]。

结果是，静态语言中可以实现多态性的只是限于拥有共父类的对象。

但是，把对象统一处理的观点可能不止一个。比如对于字符串类，如果着眼于能够比较大小这一性质的话，我们有时想把它与数值等统一处理。这时我们可能会创建一个能够比较大小的父类，让数值和字符串来继承这个父类。

而在同一程序中别地地方，考虑到字符串类是字符的序列，为能实现把其中的元素按照顺序取出的操作，又想把它与列表等统一处理。这就需要给字符串和列表定义一个共通的父类。但是单一继承只能有一个父类，不可能同时实现比较大小和按顺序访问这两个要求。

所以，为了解决这个问题，多重继承在静态编程语言中是必要的。实际上，静态面向对象编程语言的代表 C++ 和 Eiffel[21]都支持多重继承。Java 也可以通过接口来支持规格和多重继承。

---

[20]子类对象拥有父类所有属性，可以当做父类对象来处理，这种状态称为 LSP(Liscov Substitution Principle)。

[21]Eiffel 是在 20 世纪 80 年代后期由 Bertrand Meyer 设计的面向对象编程语言。其主要特点是静态类型与多重继承，严密的规格与“基于契约的设计”。Eiffel 在国外金融等领域中得到了实际应用。

3.4.5 2.4.5 动态编程语言也需要多重继承

动态编程语言没有类型检查，从这方面来说没有理由用多重继承。那么动态编程语言真的不需要多重继承吗？

肯定不是这样的。

无论从类型上考虑结果如何，从模块的角度来看，单一继承也有很多不便性。比如“一个文件只能利用一个库”这样的限制就让我们感到很不自由。

当然，实现的共享可以通过多个对象的组合（composition）和委托（delegate）[22] 来做到，Java 中就推荐这种方法。

但是如果把类当做模块来看的话，多重继承相当于语言功能支持模块组合。有了多重继承，同样的处理可以简单地记述，可以促进实现的共享。从 DRY 原则的角度来看，今后的面向对象语言也应该支持多重继承。

3.4.6 2.4.6 驯服多重继承的方法

多重继承因为有多个父类，所以可能引发下面两个问题。

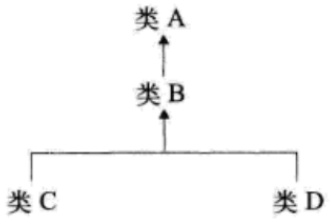
（1）类关系复杂化。

（2）继承功能名字重复。

最初的问题起因是类的关系从简单的树结构变成了复杂的网状结构。单一继承时，子类和父类、父类和它的父类……之间的关系是一条直线。

多重继承时，类之间的关系变成由一个类作为顶点的有向图。如图 2-22 所示，优先级不能被简单地确定。图 2-22 左边显示的是单一继承的例子。类 C 和父类之间的优先级是 C-B-A，简单而明确。右边显示的是多重继承的例子。类 5 的父类有类 1、类 2、类 3 和类 4，但是父类之间的优先级并不明确。

[22] 组合是把多个对象合成一个对象来处理。委托是把对一个对象的方法调用委派给别的对象。



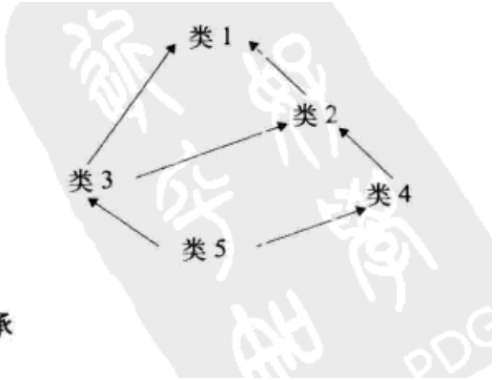


图 2-22 单一继承和多重继承

p/图 2-22.png p/图 2-22.bb

解决优先级问题需要巧妙的设计, 设计好的话, 就不会有（或难以发生）问题。多重继承确实容易让类之间的关系变得复杂。不管怎么说，和单一继承相比，这个一个很显眼的缺点。但是如果能够进行巧妙和适当的设计，大部分场合这个问题是可以避免的。

多重继承设计的一个有效的技巧是 Mix-in。Ruby 也利用了这个技巧。

用 Mix-in 做多重继承设计时，从第 2 个父类开始的类要满足以下条件。

（1）不能单独生成实例的抽象类。

（2）不能继承 Mix-in 以外的类。

满足这两个条件的类称为 MIx-in 类。正是因为这些限制，Mix-in 类可以说是功能模块。通过 Mix-in 类的功能和一般类的组合，继承关系既单纯，又可以享受多重继承的优点。

有这么多的限制，对于 Mix-in 的实用性，恐怕有人会抱有怀疑的态度。

其实一般的继承是可以变换成基于 Mix-in 的关系的。请看图 2-23 和图 2-24。图 2-23 是 Window 类的多重继承关系。在一般的 Window 类上，加上标题栏就是 Titledwindow 类，加上边框就是 FramedWindow 类，既有标题又有边框的是 TitledFramedWindow 类。TitledFramed-Window 类分别继承了 TitledWindow 类和 FramedWindow 类。

而在图 2-24 中，用 Mix-in 实现了相同的功能。标题功能和边框功能分别被做成两个 Mix-in 类，这样 TitledWindow 类、FramedWindow 类和 TitledFramedWindow 类就成为了 3 个独立的类。

这样，通过对功能的分离，多重继承就可以由单一继承加上 Mix-in 类来实现。利用 Mix-in 就可以同时享有单一继承的单纯性和多重继承的共有

性。

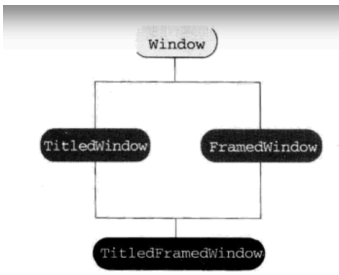


图 2-23 多重继承的例子，Window 类的继承关系

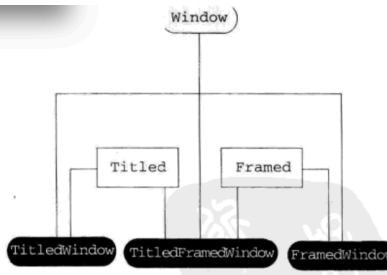


图 2-24 图 2-23 的 Mix-in 版

p/图 2-23-24.png p/图 2-23-24.bb

另外一个比较麻烦的问题是名字重复。多重继承编程语言都有自己的对应方法，大致上分为以下 3 种。

(1) 给父类定义优先级

重复的时候使用优先级高的父类属性。Common Lisp Object System(CLOS) 提供的这个功能在继承数据类型时很有效。

(2) 把重复的名字替换掉

Eiffel 使用的就是这种方法。在模块继承时用这种方法很有效，其缺点是写程序时很复杂。

(3) 指定使用类的名字

C++ 用的是这种方法。这也是在继承模块时有效的方法。缺点是本来不需要指定类名的情况现在却要指定。

从对应方法可以明白地看到各种语言的特点。Lisp 重视数据类型的继承，Eiffel 和 C++ 重视模块的继承。

3.4.7 Ruby 中多重继承的实现方法

当初设计 Ruby 的时候，Mix-in 并不广为人知。我认为 Mix-in 是解决多重继承问题的非常好的方法。所以，出于启蒙的目的，我特意在 Ruby 中强制采用了 Mix-in，而没有使用普通的多重继承。

不知道是否因为这个原因，Mix-in 的知名度提高了。但是也像前面说过的一样，甚至一些计算机科学的研究者也产生了对多重继承的误解。

我们比较一下在 Ruby 中使用和不使用 Mix-in 的区别。图 2-25 是使用 Mix-in 的 Ruby 程序。用 module 定义的是 Mix-in 类。

图 2-25 的 LockingMixin 可以对任意的类提供 lock 功能。在这里，给 Printer 类增加了 lock 功能。在 spool 方法中调用了 lock 方法。

图 2-26 是没有 Mix-in 的 Ruby 程序。这里增加了实现 Lock 功能的对象初始化，添加了 Lock 方法，还要定义很多方法的委托调用。比较起来，Mix-in 程序就很简洁。

```
module LockingMixin

  def lock
    ... ...
  end
  def unlock
    ... ...
  end
end

class Printer<Device

  include LockingMixin
  def spool(text)
    lock
    ... ...
    unlock
  end
end
```

图 2-25 利用 Mix-in 的 Ruby 程序

```
class LockingMixin

  def lock
    ... ...
  end
```

```
  def unlock
    ……
  end
end

class Printer<Device

  def initialize
    @lock=Lock.new
  end
  def lock
    @lock.lock
  end
  def unlock
    @lock.unlock
  end
  def spool(text)
    @lock.lock
    ……
    @lock.unlock
  end
end
```

图 2-26 不用 Mix-in 来实现图 2-25 的功能

3.4.8 2.4.8 Java 实现多重继承的方法

Java 采用了单一继承。但是为了满足静态编程语言对多重继承的需要, Java 采用了规格的多重继承, 即接口。如果使用接口, 即使对没有继承关系的不同种类的对象也可以做共通的处理。

但是接口只能实现规格的多重继承, 实现的多重继承在 Java 中是不允许的。这种设计原则多少会让人感觉到不方便吧。

因为不允许实现的多重继承，如果要共通实现的话，一般要像图 2-26 所示的程序一样使用委托的方法。图 2-27 是使用委托来共通实现的 Java 程序例子。它把从接口调用的方法，都明确地委托给实现共通功能的对象。本来在多重继承中可以自动实现的，现在要通过手工来实现。虽然有些麻烦，但这也算 Java 的风格吧。

图 2-28 没有用委托的方法，而是把实现共通功能的对象作为成员变量来使用。这样操作对象并不需要直接实现接口，而只是作为属性保存一个实现共通功能的对象，在程序中直接调用该属性的方法。

没有了委托的方法，这些部分就变得简单明了，但是在调用共通功能的时候，每次都要引用属性加上.lock，会让人觉得不怎么漂亮。

本节回顾了多重继承，要点有以下 5 个。

(1) 多重继承并不可怕。

(2) 今后面向对象编程语言必须有某种形式的多重继承。

(3) 类既有类型的一面又有模块的一面。

(4) C++、Eiffel 等语言积极利用了类的模块的一面。

(5) 使用 Mix-in 可以避免多重继承的类关系变复杂。

正确地使用多重继承是提高程序效率的有效方法。如果本节的说明能够减少对多重继承的误解，那我就感到很幸运了[23]。

```
interface LockingMixin {
  void lock();
  void unlock();
}

class Lock {
```

[23]本节内容参考了《面向对象入门》一书，该书由 Bertrand Meyer 著，二木厚吉审校，酒顺子译，Ascii 出版，ISBN4756100503。书名是“入门”，但根本不是面向初学者的，而是面向中高级读者的书。例题都是用 (大家不太熟悉的) Eiffel 语言写成的，遗憾的是 Eiffel 本身的版本也很老，但即使有这些缺点，这本书还是非常有价值的。例题以外的内容一点也不过时。如果想要进一步深入理解面向对象编程的话，这本书可以说是最好的。还有，翔泳社重新翻译了这本书，分两册出版：《面向对象入门 (第 2 版): 原则. 概念》，《面向对象入门 (第 2 版): 方法论. 实践》。

```
  void lock(){…… ; };
  void unlock(){…… ; };
}

class Printer implements LockingMixin{
  final Lock lock=new Lock();
  void lock() {lock.lock();}
  void unlock() {lock.lock();}
  void spool(TextData text){
    this.lock();
    ……
    this.unlock();
  }
}
```

图 2-27 用委托来实现多重继承的 Java 程序

```
interface LockingMixin {
  void lock();
  void unlock();
}

class Lock implements LockingMixin{
  void lock(){};
  void unlock(){};
}

class Printer{
  final Lock lock=new Lock();
  void spool(TextData text){
    this.lock.lock();
    ……
```

```
        this.lock.unlock();
    }
}
```

图 2-28 不用委托来实现多重继承的 Java 程序

3.4.9 2.5 Duck Typeing 诞生之前

在编程世界中，经常提到静态 (static) 与动态 (dynamic) 这样的词汇。静态是指程序执行之前，从代码中就可以知道一切。程序静态的部分包括变量、方法的名称和类型以及控制程序的结构等等。

相对于静态，动态是指在程序执行之前有些地方是不知道的。程序动态的部分包括变量的值、执行时间和使用的内存等等。

如果知道育种使用的算法和输入值，虽然有时候不执行也可以知道输出的结果，但是现实中这种单纯的情况很少。通常情况下，程序本来就是不被执行就不知道结果的，所以从一定程度上说程序都具有动态特性。因此，严格地说，静态和动态之间的界限是微妙的。

3.4.10 2.5.1 为什么需要类型

在程序中具有动态或静态特性的东西很多，这里以类型为重点，讲解一下静态类型和动态类型。

编程语言中的类型指的是数据的种类。例如整数和字符串都是数据的类型。从硬件的角度来看，计算机可以处理的类型只有二进制。在计算机可以直接操作的汇编语言中，数据类型都是整数[24]，其他类型的数据都用整数来表现。

例如表现字符串的时候，是给每个字符都编上号，这些整数编号排列起来构成了字符串。内存中的地址 (位置) 也是用整数来表现的。数据、对象等复杂数据也是一样的。

但是这种处理方法是很低级的，它要求人要理解、记忆用整数来表达所有类型数据的方法。不小心出一点关错程序就不能运行。

[24] 只处理二进制数的说法只是一个概念。实际上 CPU 可以直接处理浮点小数等整数以外的类型。

这样的话程序员的负担就太大了，所以编程语言就进化了。被称为世界上最初的编程语言的 FORTRAN(Formaula Translator, 公式变换机), 引入了变量和算式的类型。在程序中，变量只能用整数赋值，数组只能是浮点的数组等，可以指定数据类型。这是静态数据类型的开始。这种对数据类型的定义称为类型定义。图 2-29 是 C 语言的类型定义，它是静态语言的代表。

假设在程序中把字符串赋值给整数型的变量，那么根据程序的定义，编译器知道监事会语句中值的类型（字符串）和变量的类型（整数），所以能够检查出这种类型不匹配的错误。静态的类型不用执行程序就可以通过机器检测到这种人为错误，可以说是一项伟大的发明。

3.4.11 2.5.2 动态的类型是从 Lisp 中诞生的

在 FORTRAN 出现数年之后诞生的 Lisp 编程语言（Lisp Processor, 列表处理机），对于数据类型的问题采取了另一种解决方法。在 1958 年刚出现时，Lisp 只支持列表 (list) 和原子 (atom) 这两个数据类型。

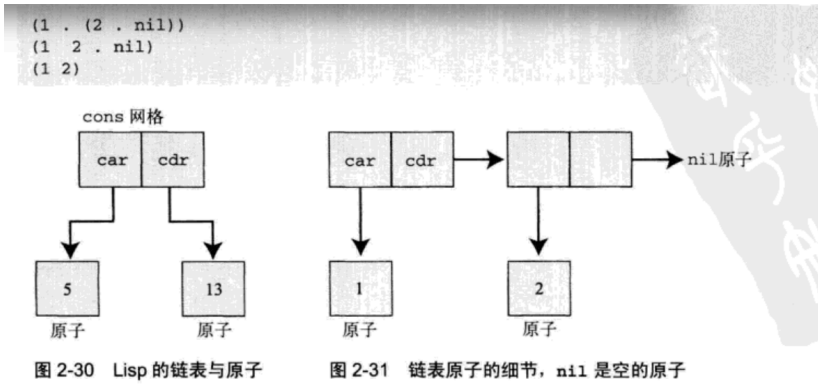
列表是可以由两个引用的节点（node）构成的单向列表, 比如像（5 13）这样的数据。原子比较难说明，简单地说就是指 list 以外的数据类型，比如数据和字符串都是原子（参见图 2-30）。

List 的每个节点，历史上称为 cons 单元，可以引用其他的 cons 单元或原子。cons 单元可以有两个引用，因为历史的原因，前面的称为 car，后面的称为 cdr（参见图 2-31）。

在 Lisp 中程序和数据如果不能用文字来表现的话会很麻烦。所以 Lisp 用字符串来表现名为 S 式的列表。S 式是用以下规则把列表转换成字符串的。

- cons 单元中，car 的值和 cdr 的值用点连接，再用括号括起来。
- cdr 如果是列表的话，省略括号。
- 末尾 cdr 如果是 nil，那么省略.nil。

按顺序应用这些规则的话，图 2-31 的列表可以用下面的字符串来表示。从第一行开始按照顺序进行简化，就可以得到第三行的结果。



p/图 2-31-32.png p/图 2-31-32.bb

List 的数据用列表，程序也用列表，所有的东西都用列表来表示。Lisp 用 S 式表示的列表构造本身就是程序。图 2-32 是 Lisp 的计算阶乘的程序。

List 的列表中的各个 cons 单元是指向 cons 单元还是原子，事前是不知道的。列表中 cons 单元和原子混杂存在，这从本质上就可以说是多态的数据结构。以这种数据结构为基础的 Lisp 采取的战略是数据要记录与自己数据类型有关的信息。这样的数据类型称为动态类型。

Lisp 程序中，如果用只能处理原子数据的方法来处理 cons 单元数据，执行时的数据类型检查就会报告错误。因为执行时有类型检查，所以一旦发现有不正确的处理，程序就会停止执行。但是不执行程序的话是无法知道哪儿有错误的。

3.4.12 2.5.3 动态类型在面向对象中发展起来了

编程语言的数据类型分为两类，一类是起源于 FORTRAN 的指定了变量或算式数据类型的静态类型，另一类是起源于 Lisp 的动态类型。静态类型从 FORTRAN 开始，通过 COBOL、ALGO 被很多编程语言采用。

C 语言的原型是没有数据类型定义的 BCPL 语言，该语言受到了 ALGOL 语言的影响采用了静态类型。

在很长的时间里，只有 Lisp 和受其影响的语言（比如 LOGO）才采用了动态类型，但是以“某件事情”为契机，动态类型开始得到广泛接受。

那个“某件事情”，就是面向对象编程。最初的面向对象编程语言 Simula，受到了 ALGOL 语言的很大影响，像整数等这些基本的数据类型都采用了静态类型。但是对新引入的对象，不论它是哪个类的对象，全都用 Ref 这种

类型来表现。

不管是什么类的对象，它们的静态类型都是 Ref 型，没有任何区别。但是 Ref 型数据知道自己是什么类的对象，所以 Ref 可以说是动态类型。

仔细想想，对象保存着有关自己种类的信息，某个变量可以用各种类型的数据来赋值，这两点是多态这一面向对象重要特性的必要条件。因为如果变量类型和赋值数据的类型必须是完全一致的静态类型的话，程序执行时就不可能根据数据类型的不同来自动选择合适的处理方法。如果没有从 Lisp 中起源的动态类型，面向对象可能也不会存在吧。

继承了起源于 Simula 的面向的思想，Smalltalk 像 Lisp 一样，全面采用了动态类型。虽然从外部来看，Smalltalk 和 Lisp 完全不一样，但从内部构造来看，它们像双胞胎一样。另外，Lisp 也增加了面向对象的功能，独立地发展到了现在。

无论如何，从 20 世纪 70 年代到 80 年代，面向对象编程是由动态类型语言 Smalltalk 和 Lisp(及它们的各种方言) 所支撑的。

3.4.13 2.5.4 动态类型和静态类型的邂逅

20 世纪 80 年代, 面向对象编程语言和主流是包含动态数据类型的语言。但是在 21 世纪的今天，使用最广泛的面向对象编程语言是具有静态数据类型的 Java 和 C++。在面向对象编程语言的历史上，究竟发生了什么呢？

在 20 世纪 80 年代初期，受到 Simula 的影响，一个面向对象的编程语言诞生了，这就是 C++。C++ 包含了 C 的所有功能，并增加了由 Simula 发展而来的面向对象功能[25]。

在 Simula 中，除对象以外的数据类型都是静态类型。受到 Simula 很大影响的 C++，因为引入了一个原则，对象也采用了静态类型。

这个原则就是子类对象可以看成是父类对象。具体来说，如果 String (字符串) 类是 Object （对象）类的子类的话，那么 String 类的所有对象都可以看做是 Object 类的对象 (参见图 2-33)。

[25]C++ 没有受到 Smalltalk 的影响，而是受到 Simula 的直接影响，所以很多用语都不同于 Smalltalk。比如，父类称为基类 (base class)，子类称为派生类 (derived class)。这反映了 C++ 的作者 Bjarne Stroustrup 曾是 Simula 用户。

class Object{ .....; } ; class String:public Object{ .....; } ; //看成是 Object String *str=new String(); Object *obj=str;

图 2-33 C++ 对象与静态类型的关系

根据这个原则，在编译时就可以知道变量或算式的类型，又可以根据执行时的数据类型看上去选择合适的处理，从而同时具备了静态类型的优点和动态类型的多态性。

因为在编译时就知道了变量和算式类型，所以可以在执行前就发现类型不匹配错误。这是一个很大的优点。另外，根据类型信息在编译时大胆进行优化，可以提高程序执行速度。

因为有这样的优点，C++ 与 20 世纪 90 年代受到 C++ 影响诞生的 Java，以及 C# 等采用静态数据类型的面向对象编程语言，都得到了广泛的使用。

3.4.14 2.5.5 静态类型的优点

现在静态类型的面向对象编程语言被广泛使用。首先，我们比较一下静态类型和动态类型的优缺点。

静态类型最大的优点是在编译时能够发现类型不匹配的错误。当然，在编译中是不可能发现程序中所有问题的，但是由于很多问题都是由类型不匹配引发的，所以虽然不能发现全部总题，但这种自动发现问题的功能对我们的帮助还是很大的。

与其相反，动态类型的编程语言至多只能发现程序语法错误。

程序中如果明确指定了数据类型，那么时可以用到的信息就很多。利用这种信息可以在编译时对程序估优化，提高程序执行速度。

数据类型信息不只是对编译器有用。我们在看程序的时候，“这个参数是什么类型”的信息对我们理解程序也是有很大帮助的。集成开发环境 (IDE) 也可以利用这些信息来自动补充完整方法名。这些功能的实现都得益于可以利用的类型信息。

最后，变量和算式分别有自己的类型，这使得我们能够在一开始就认真考虑这些变量应该扮演什么样的角色。我们在编写程序时就要考虑数据类型，虽然要考虑的东西变多了，但是也不能简单了说这是坏事。显而易见，

这是我们开发的、可靠性高的程序所必需的。

从以上几点来看，静态类型似乎全部是优点。其它它也有几个缺点，或者说是问题。

其中一个问题是，若不指定类型就写不了程序。当然，指定类型是静态类型编程语言的特征之一。但是说到底，数据类型只是一些辅助信息，并不是程序本质。当我们想把精力集中到程序处理的实际问题时，却要一个个考虑数据类型的定义，这就很烦琐的。并且，有时会让人觉得，有的类型声明仅仅是为了满足编译器的要求。程序规模也因为数据类型的定义而变大，重要的部分反而容易被忽视。

另外一个是灵活性的问题。静态类型本身限制了给某个变量只能赋值某种类型的对象，这种限制可能成为妨碍将来变化的枷锁。前面学过的多重继承和接口会产生令人费解的继承关系，这时怎样设定适当的类型就变得比较困难了。

总结一下，用静态类型编程语言的人通过定义类型，把更多的信息传达了出来，这算是给编译器的将来读程序的人减轻负担的一种方法吧。

3.4.15 2.5.6 动态类型的优点

前面介绍了静态类型，那么动态类型又怎样呢？动态类型编程语言的最大优点是源代码变得简洁。编程语言的进化使我们可以用更简单的程序来传达给计算机更多信息。如果不用指定与程序本质无关的数据类型，程序也完全可以正确执行，也可以检测出来错误的话，这不是一种很好的想法吗？

得益于简洁，我们在编写程序的时候就不用考虑数据类型这些无关本质的部分了，而是可以集中于程序处理的本质部分，编写简洁程序的话，也可以提高生产力。

另一方面，有人会担心，简洁和程序虽然让我们在编程的时候变得简单了，但是因为没有类型信息，以后读起来是不是就变得难以理解呢？虽然写的时候容易了，但难读的程序也是不可取的。对于这样的担心，我的回答是，简洁的程序更突出了程序处理的实质，理解起来反而变得简单了。实际上，动态类型的编程语言（例如 Ruby）的程序规模和静态类型相比，程序行数相差数倍的情况并不少见。很多人都感觉到动态类型的程序更好理解。

对于简洁程序的另外一个担心是，动态类型语言是否运行缓慢呢？事实上是这样的。同样的处理，在大多数情况下，静态类型编程语言运行得要快些。

这是因为动态类型程序执行时要做类型检查。另外，静态类型的编程语言大都通过编译把程序源代码转换成可以直接执行的形式，而动态类型的编程语言大多是边解释源代码（转换成内部形式）边执行，这种编译型处理和解释型处理的区别也是影响程序执行速度的原因之一。说起程序，很多时候执行速度并不是很重要的，随着计算机性能的提高，执行速度就更不是什么严重的问题了。

动态类型编程语言的另外一个特点是灵活性。动态类型语言的程序不用指定变量的数据类型，所以即使开发时没有考虑到的数据类型也可以轻松地处理。这种灵活性的关键是我们下面要讲的 Duck Typing 概念。

动态类型编程语言的最大缺点是不执行就检测不出错误。和静态类型的自动错误检测相比，这算是它的不足吧。

3.4.16 2.5.7 只关心行为的 Duck Typing

表达动态类型灵活性的概念是 Duck Typing。下面是来自西方的一句格言。

It it walks like a duck and quacks like a duck, it must be a duck（走起路来像鸭子，叫起来也像鸭子，那么它就是鸭子）。

从这里可以导出这样的规则：如果行为像鸭子，那么不管它是什么，就把它看做鸭子。根本不考虑一个对象属于什么类，只关心它有什么样的行为（它有哪些方法），这就是 Duck Typing。提出 Duck Typing 这个概念的是大名鼎鼎的专家程序员 Dave Thomas。

我们来看一个 Duck Typing 的具体例子。假设有一个例程 $\log_{puts}()$, 向文件输出日志消息。假定这个方法有两个参数（输出对象和要输出的消息）。如果是静态类型编程语言，比如 C++，程序会是像下面这样的。

void $\log_{puts}$(ostream out, char* msg);

$\log_{puts}()$ 例程向输出 out 里输出时刻和消息。调用这个例程如下所示。

$\log_{puts}$(cout,"message");

cout(C++ 的标准输出设备）上会输出如下日志。

2005-06-16 16：23：53 message

现在，如果我们想把 log 输出到字符串而不是文件的话，那该怎么办呢？

因为指定输出对象的 out 参数的类型已经定义成 ostream，无法简单地变更，结果我们要么把 $\log s_{puts}()$ 这个例程全部复制一遍，另外新增一个以字符串为输出对象的例程，要么先输出到临时文件，然后再把它读到字符串里，没有别的办法。

那么，如果使用 Duck Typing 的话，会变成怎样灵活的代码呢？如下所示。

$\log_{puts}$(out,msg)

因为是动态类型，所以程序中不用指定参数的类型，下面的调用会和 C++ 一样向 STDOUT(Ruby 的标准输出设备）输出同样的日志。

$\log_{puts}$(STARTED,"message")

好了，和刚才一样，现在我们想把信息输出到字符串，有了 Duck Typing 就简单多了。任何对象，如果它拥有和输出对象（标准输出设备）相同的方法，那么就可以用它作为输出对象。

Ruby 字符串有和文件一样的输入输出类 StringIO。图 2-34 演示了用 StringIO 来实现输入输出。

```
require 'stringio'
out = StringIO()
log_puts(out,"message")
puts out.string
```

图 2-34 Duck Typing 的例子，使用 Ruby 的 StringIO 类

StringIO 类和 STDOUT 的类（IO）没有继承关系。但是，StringIO 类中有 IO 类的所有方法。所以，几乎在所有的情况下，StringIO 可以像 IO 一样地来使用。

如果用静态语言实现相同功能，需要首先定义一个具有 log 输出功能的类（在 Java 中是接口），然后将它定义为 $\log_{puts\ 第一个参数的类型}$。像这个例子，如果输出对象的类型是编程语言中既有的类型，那就需要重新定义另外一个对象来表达输出对象。即使是在刚开始编写程序的时候就采用这种机制

也很费事，而如果是在中途才开始引入的话，程序到处都将需要大规模修改。

使用静态类型语言，程序员通过类型定义提供了大量的信息，错误可以尽早检测出来，程序确保可以执行。其代价是，如果类型设计的前提发生了变化，为保证各种类型的一致性，所有关联的部分都要修改。动态类型语言因为开始就不需要定义数据类型，所以适应类型变化的能力比较强。

去么，动态类型语言用 Duck Typing 的概念设计时要遵循什么原则呢？基本原则只有一个，最低限度是只要掌握下面这个基本原则应该就没有问题了。

3.4.17 2.5.8 避免明确的类型检查

有时需要在程序中检查参数的数据类型。例如图 2-35，如果希望处理对象是字符串，自然就会想在不是 String 类对象的时候，抛出异常，报告错误。

```
if not obj.kind_of? (String)
raise TypeError, "not a string"
end
```

图 2-35 进行明确类型检查的例子，在变量不是 string 类的时候，抛出异常。

```
if not obj.repond_to?("to_str")
raise TypeError, "not a string"
end
```

图 2-36 用方法来检查数据类型，只接受有 to_{str} 方法的对象。

但是如果要用 Duckk Typing 的概念来实现程序的话，怎么也要忍着点，不要把程序写成这样。如果以类为基准进行数据类型检查的话，就会像静态编程语言一样失去灵活性。无论如何都想检查的时候，也不要检查对象是否属于某个类，而是要检查对象是否有某个方法（参见图 2-36）。

其实即使不检查方法，如果处理对象不是程序所期待的对象，也肯定会出现找不到方法错误。

3.4.18 克服动态类型的缺点

动态类型的缺点主要有三个，即在执行时才能发现错误、读程序时可用到的线索少，以及运行速度慢。

首先，执行时才能发现错误这一点可以用完备的单元测试来解决。如果能严格实行完备的单元测试的话，即使没有编译时的错误检查，程序的可靠性也不会降低。

其次,读程序时可用到的线索少这一点可以通过完整的文档来解决。Java 有 JavaDoc 技术，Ruby 也有 RDoc 技术，可以在源代码中同时写文档，减轻维护文档的负担。

最后，运行速度慢这一点，随着计算机性能的提高已经不再重要，现在的程序开发中，程序的灵活性和生产力更为重要。

3.4.19 动态编程语言

现在我们对程序开发生产力的要求越来越高。也就是说，要在更短的时间内开发出更多的功能。

开发周期短，就要求我们在开发过程中不断探求最合适的开发方法。这又被称为“射击移动的目标”。像以前那样，一开始把所有的情况都考虑到，在确定了需求之后再进行开发的方式已经越来越行不通了。尽快着手开发，快速应对需求变更的开发方式变得越来越重要。

在这种快速开发模式中，Duck Typing 所代表的执行时的灵活性就非常有用。Ruby、Python、Perl 和 PHP 等优秀的动态类型编程语言，因为它们在执行时所具有的灵活性而越来越受到人们关注。

3.5 2.6 元编程

“元”一词来源于希腊语中表示“.....之间, ...之后，超越.....”的前缀“meta”，有“超越”和“高阶”等意思。在 Ruby 和其他一些面向对象编程语言中，类的类称为元类，支撑别的对象的类对象称为元对象。

元编程是对程序进行编程的意思。也话会让人感觉没什么用。初看起来，的确有些让人摸不着头脑，下面就来一窥元编程的威力吧。

3.5.1 2.6.1 元编程

首先我们看一个 Ruby 元编程的例子。这是一个动态生成方法的示例。

在 Ruby 类中内嵌的 attr-accessor 方法模块可以动态生成访问实例变量的方法（参见图 2-37）。在图 2-37 简短的程序中，给 Person 类自动生成了 name 方法和 age 方法，也可以用它们来赋值。

重要的是，attr-accessor 并不是 Ruby 中的一个语句，而是 Module 类提供的一个方法，也就是说，如果你愿意的话，也可以自己来定义具有类似功能的方法。

class Person

$\text{attr}_{\text{accessor}}$:name, :age

end

图 2-37 元编程的例子。Ruby 使用 $\text{arrt}_{\text{accessor 生成访问实例变量的方法}}$（这里是 name 和 age）

$\text{attr}_{\text{accessor 内部进行如下处理。}}$

（1）对所有的参数作以下的处理。

（2）生成与参数名同名的方法。用该方法可以访问“@ 参数名”这个实例变量的值。

（3）生成参数名后加“=”的方法。该方法有一个参数，它把参数的值赋给“@ 参数名”这个实例变量。

以上步骤看起来很简单，但是在 C 或者 C++ 语言中很难实现的。因为在 C++ 等程序执行时，是不能动态地给类增加方法的。Ruby 可以像图 2-37 这样简单地实现这一功能。实际上 attr-accessor 是用 C 语言写成的，如果用 Ruby 自己来写这个方法的话，会像图 2-38 的程序那样。

```
class Module

  def attr_accessor(*syms)
    syms.each do |sym|
    class_eval %{
      def #{sym}
       @#{sym}
```

```
        end
        def #{sym}=(val)
          @#{sym}=val
        end
      }
    end
  end

end
```

图 2-38 用 Ruby 自己来实现 $\text{attr}_{\text{accessor}}$

$\text{class}_{\text{eval 方法接受字符串参数}}$，在类的上下文中对字符串进行处理。在图 2-38 中，从%{到} 之间的字符串作为参数传递给 $\text{class}_{\text{eval 方法}}$。字符串中 #{和} 之间是可以替换的标识符，会被展开成 sym 参数所代表的方法名，每个循环定义两个方法。因此，下面的调用会在被调用的对象类中生成两个方法：一个方法是 name，用来访问实例变量 @name 的值；一个方法是 name=，用来给实例变量 @name 赋值。

$\text{attr}_{\text{accessor}}$: name

不支持元编程语言实现这样的功能是很麻烦的。要么需要扩展语言的语法，要么用宏定义等预处理的方法来实现。无论怎样，在普通语言中这都会很麻烦。

3.5.2 2.6.2 反射

下面说明一下元编程的反射（reflection）功能。relection 这个英语单词是反射、反省的意思。在编程语言中它是指在程序执行时取出程序的信息或者改变程序信息。

表 2-3 列出了 Ruby 的反射功能，包括取得变量或方法取得或变更值等，很丰富。比如实现 Mix-in 的 include 并不是 Ruby 的语法，而是通过方法来实现的。所以说 Ruby 彻底实现了对程序的动态操作。

现在我们来看一下这些功能到底都能实现些什么。

3.5.3 2.6.3 元编程的例子

首先看一下反射的例子。

有时我们需要把对一个对象的调用委派给另外一个对象。Ruby 用 Delegator 这个库实现了委托功能。Delegator 对象中包含有方法委托的对象, 把方法调用委派给委托的对象。它实现了设计模式中 Proxy 模式的基础部分。要使用 Delegator 功能, 可以用 SimpleDelegator 这个类。

requir 'delegator'

d=SimpleDelegator.nex(a)

只用这两句就可以实现, 调用对象 d 的方法时可以转变为调用对象 a 的方法。仅仅是委派的话也没有什么让人高兴的, 实际上我们可以给这个对象增加特异方法[26]来改变它的部分行为, 这就大大扩展了它的应用范围。

这种处理在 Java 中如何实现呢？在静态类型编程语言 Jave 中, 因为需要匹配类型, 所以要另外生成一个 Delegator 类, 专门对应 a 的类型来传送每个方法调用。a 的方法如果很多的话, 这将是很烦琐的工作。恐怕需要用专门的工具来自动生成才行。

而 Ruby 通过动态类型的反射功能第一个实现了 Delegator。

表 2-3 Ruby 的反射功能

[26]所谓特异方法, 是指类中没有定义而只存在于实例 (对象) 中的方法。Ruby 以外的其他语言也会用到。

功能	方法名
列出类/module 的方法	Module#instance$_{methods}$
列出对象对方法	Object#methods
列出对象的实例变量	Object#instance$_{variables}$
列出全局变量	global$_{variables}$
列出局部变量	local$_{variables}$
列出类/module 的常量	Module#constants
获取常量值	Module#const$_{get}$
设置常量值	Module#const$_{set}$
删除常量	Module#remove$_{const}$
列出类变量	Module#class$_{variables}$
获取类变量值	Module#class$_{variableget}$
设置类变量值	Module#class$_{variableget}$’
删除类变量	Module#remove$_{classvariables}$
定义类方法	Module#define$_{method}$
删除类方法	Module#remove$_{method}$
解除类方法定义	Module#undef$_{method}$
给类方法赋予别名	Module#alias$_{method}$
包含模块	Module#include
获取父类	class#superclass
获取包含的类	Module#included$_{modules}$
获取类的继承关系	Module#ancestores
给继承设置钩子处理	Class#inherited
给包含设置钩子处理	Module#include
给方法定义设置钩子处理	Module#method$_{added}$
给特别方法定义设置苑处理	Module#singleton$_{methodadded}$
给未定义的方法设置钩子处理	method$_{missing}$
解释字符串	eval
在对象的上下文中解释字符串	Object#instance$_{eval}$
在类的上下文中解释字符串	Module#class$_{eval}$
检查是否有定义	defined *

3.5.4

3.5.5 2.6.4 使用反射功能

让我们来看看 Ruby 是怎样用反射功能来实现 Delegator 这个类的。图 2-39 是 Simple-Delegatro 类的部分代码。为了容易理解，例子中程序被在幅度简化了。

图 2-39 的程序分为 4 个部分。首先说明最重要的。图 2-39C 是 Delegator 的核心部分。Ruby 在调用方法时，如果对象不知道这个方法，就会首先调用 $\text{method}_{\text{missing 这个方法}}$。$\text{method}_{\text{missing 的第一个参数是被调用的方法的名字}}$，剩下的是传给方法的参数。$\text{method}_{\text{missing 的默认实现是进行异常处理的}}$，但通过重载，也可以处理未知的方法。接着说明下面的两个处理。

（1）被委派的对象如果不知道这个方法（$\text{respond}_{\text{to}}$?），默认的实现会被调用（super），发生错误。

（2）知道的话，用——$\text{send}_{\text{来调用委派对象对方法。}}$

——$\text{send}_{\text{是调用委派对象方法的方法。}}$ 这个方法的别名是 send，由于 send 容易重名，所以用了 $\text{send\_\_}_{\text{。}}$

SimpleDelegator 剩下的部分比较容易实现。就像刚才说明的，SimpleDelegator 是通过 $\text{method}_{\text{missing 来委派方法的。}}$ 实际上 Object 类有 40 多个方法。SimpleDelegator 是 Object 的子类，是知道这 40 多个方法，也就不能委派。为解决这一问题，（a）中用 $\text{instance}_{\text{mothods 获取方法的列表，}}$除了几个必要的方法 (——id——、$\text{object}_{\text{id}}$、——send——、$\text{respond}_{\text{to}}$?) 以外，取消了其他方法的定义。

(b) 用于设定 SimpleDelegator 的委派对象。（d）是为了让 $\text{respond}_{\text{to}}$? 可以正常执行，首先用 super 检查自己的方法，然后检查委派对象的方法。

3.5.6 2.6.5 分布式 Ruby 的实现

Delegator 将被调用的方法直接委派到其他对象，这一功能在很多领域都有应用。作为一个例子，我们介绍一下 dRuby(Distributed Ruby，分布式 Ruby)。

dRuby 是通过网络来调用方法的库。dRuby 可以生成服务器上存在的

远程对象（Proxy），Proxy 的方法调用可以通过网络执行。

调用的方法在服务器上的远程对象中执行，执行结果可以通过网络返回。这和 Java 的 RMI(Remote Method Invocation）功能比较相似。但是，利用 Ruby 的元编程功能，不用明确定义接口，也可以通过网络调用任意对象的方法。

C++ 和 Java 的远程调用是用 IDL(Interface Definition Language）等语言来定义接口的，自动生成的存根（stub）必须编译和连接。和这些相比，Ruby 的元编程更简单。

DRuby 的最初版本只有 200 多行程序，这也体现了元编程的力量。但是现在 DRuby 作为 Ruby 的标准库，已经有 2000 多行的规模了。

3.5.7 2.6.6 数据库的应用

在数据库领域，元编程也很有用。

Web 应用程序框架 Ruby on Rails（也称为 Rails 或 RoR）[27]中也应用了元编程。具体地说，在与数据库关联的类库（ActiveRecord）中，利用元编程简单地把数据记录定义为对象。

图 2-40 是 ActiveRecord 定义数据库的例子。然后，数据库中定义了表。图 2-41 中演示了对应 users 表的 User 类。

```
class User <ActiveRecord::Base
has_one :profile
has_many:item
end
class Profile < ActiveRecord::Base belongs_to :user
end
class Item < ActiveRecord::Base
belongs_to:user
end
```

[27]在 Ruby on Rails 中，仅仅从数据库定义就可以自动生成相关的程序和配置文件，非常便利。详情请参阅 Rubyist Magazine(http://jp.rubyist.net/magazine/?0004_RubyOnRails) 的介绍资料。

图 2-40 用 ActiveRecord 定义的记录

CREATE TABEL 'users'(

'id' int(11) NOT NULL auto$_{\text{increment}}$,

'login' varchar(80) default NULL,

'password' varchar(40) default NULL,

PRIMARY KEY('id')

)TYPE=MYISAM;

图 2-41 图 2-40 用到的 users 表

从图 2-40 中几行代码可以看出，ActiveRecord 进行如下的处理。

（1）类 User 是和以类名的复数形式为名字的表（users）关联在一起的。

（2）定义了从表的全名空间（schema）访问记录的方法。

（3）用 has$_{\text{one}}$、belongs$_{\text{to}}$ 等关联定义，提供了访问关联对象的方法。

之所以能够实现这些处理，都是由于元编程功能让我们可以获取类名，在执行时增加方法。元编程的功能使得 Rails 被称赞为生产效率高的 Web 应用程序框架。

当然 Rails 不是万能的，也不能说比别的应用程序框架都好。但是，Rails 最大程度地灵活运用了 Ruby 语言的优点，从而确实提高了生产效率。

用 Rails，一眨眼的功夫就可以生成一个 Web 应用程序，给人印象颇深。

3.5.8 2.6.7 输出 XML

最后介绍一下输出 XML 文件的类库，由 Jim Weirich 开发的 XmlMarkup。图 2-42 是一个输出 XML（参见图 2-43）的简单程序。

```
require 'builder/xmlmarkup'

xm=builder::XmlMarkup.new(:indent=>2)
puts xm.html{

  xm.head{
    xm.title("History")
```

```
    }
    xm.body{
      xm.h1("Header")
      xm.p{
        xm.text!("paragraph with")
        xm.a("a Link","href"=>"http://onestrpback.org")}
    }
}
```

图 2-42 XmlMarkup 输出的例子

```
<html>
  <head>
    <title>History</title>
  </head>
  <body>
    <h1>Header</h1>
    <p>
     paragraph with <a
href="http://onestepback.org">a
Link</a>
    </p>
  </body>
</html>
```

图 2-43 图 2-42 的输出内容

Builder::XmlMarkup 和 Delegator 同样用到了 $\text{method}_{\text{missing 的技巧}}$，通过调用方法从而输出了有标签的 XML。

没有标签的文本必须用 text! 命令输出。手工写 XML 是很麻烦的，利用 Ruby 块功能则能很方便地处理。

3.5.9 2.6.8 元编程和小编程语言

到目前为止我们介绍了元编程功能，如果是你，会怎么来利用它呢？

Glenn Vanderburg[28]把它灵活运用到了 DSL（领域特定的语言）领域。DSL 是针对特定领域强化了功能的小规模编程语言。DSL 是很早就有的想法，最近，因为通过 DSL 用户可以强化应用程序的功能或者定制一些功能，所以 DSL 再次得到了关注。DSL 主要需要列于表 2-4 的一些功能。

表 2-4 小语言需要必备的功能

必要的功能	Ruby 的功能
类型（Type）	O
字面量（Literal）	O
表达式（Expression）	O
运算符（Operator）	O
语句 (Statement)	O
控制结构（Control Structure）	O
声明（Declaration）	O（与现实相关）
上下文相关（Context Dependence）	O(与现实相关）
单位（Unit）	O(与现实相关）
词汇（Large Vocabulary）	O(与现实相关）
层次数据（Hierarchy）	O(与现实相关）

Ruby 本来就具备从类型到结构这些功能。许多 DSL 小语言往往缺乏其中一些功能，Ruby 反而更好使。还有学过的 Ruby 可以自用块自己定义实现控制结构的方法，这也是它的优点之一。

剩下的从声明到层次数据等其他的功能，Ruby 也都可以利用自身的功能来实现。元编程实现这些功能起到了很大的作用。

3.5.10 声明的实现

我们介绍一下表 2-4 中声明的实现方法。

之前我们介绍了使用 $\text{attr}_{\text{accessor 进行元编程的例子}}$。在 Ruby 中 $\text{attr}_{\text{accessor 只是一个方法}}$，但是我们也可以把它看做声明。另外，ActiveRecord 中的 $\text{has}_{\text{many 方法}}$，也可以看做声明，这样的傻孩子还有很多。

[28]请参考 Glenn Vanderburg 的 / Metaprogramming Ruby Domain-Specific Language for Programmers /(http://www.vanderburg.org/Speaking/Stuff/oscon05.pfd)。

Ruby 的方法可以读取或改变程序自身的状态, 利用普通的方法调用, 可以实现其他编程语言中声明所完成的工作。

从外部来看，Ruby 的方法调用可以省略括号，还可以使用 foo 这样的符号来表示名字，这些都使 Ruby 程序看起来像在使用声明一样。

3.5.11 2.6.10 上下文相关的实现

下面讲一下上下文相关。上下文相关是指有些定义只是在一定范围内有效。我们看一下图 2-44 的例子。

这个例子中，name、password 等方法只是在 $\text{add}_{\text{user 块中才有效}}$。也就是说，在 $\text{add}_{\text{user 的}}$ 外部是看不到这些方法的。

在 Ruby 这一层次上，它把块的范围内的方法调用对象换成了 self。具体说来，像图 2-45 的例子，使用了 $\text{instance}_{\text{eval 方法}}$。

```
add_user{

  name "Charles"
  password "hello123"
  privilege normal

}
```

图 2-44 上下文相关的程序示例

```
def add_user(&block)

  u=User.new
  # User定义了name、password
  # privilege方法
  u.instance_eval(&block) if block

end
```

图 2-45 替换图 2-44 程序上下文的处理

$instance_{eval}$ 方法接受块作为参数，把调用对象转换成 self 来执行块。结果，对图 2-44 中块范围内的代码而言，默认的调用对象变成了 User 类的对象 u。所以，在不指定调用对象而调用方法（如 name 等）时，就会调用 User 类的方法。

3.5.12 2.6.11 单位的实现

在一般的编程语言中，数值是用标量值来定义的，但这只是数本身。数值的单位需程序员来管理。

然而，DSL 想要处理的不单是数，大都想要处理量。因此我们扩展了一些面向 DSL 的库，以便能处理单位。

例如在 Ruby on Rails 中,Numeric 类和 Timer 类增加了处理时间单位（基本单位是秒）的方法。比如下面的时间

3.years + 13.days + 2.hours

表示“3 年 13 天又 2 小时”，以秒为单位就是整数 95803200。另外像下面这样：

$4.months.from_{now.monday}$

表示“4 个月后的星期一”。

我写这本书的时间是

Mon Dec 12 00:00:00 JST 2005

这里只是举了一些与时间想着的例子。因为 Ruby 中可以给再有类自由地追加新方法，所以单位处理功能是很容易实现的。

3.5.13 2.6.12 词汇的实现

DSL 是针对特定领域的，所以在这个特定领域需要什么处理，需要用词汇来表现。针对特定领域，如果用 Ruby 来定义需要的类和方法，那么就可以认为 Ruby 是这个领域的专用语言。这些类和方法可以称为这个领域的词汇。

借用著名的程序员 Dave Thomas 的话，“所有应用程序的开发过程都可以说是设计语言的过程”。从这个观点来看，开发应用程序就是针对应用程序的总是领域定义各种词汇，最后用这些词汇来描述解决总是的方法。

Ruby 的方法调用和块等具有丰富的表现力，用户可以很自然地用它们来定义词汇。另外如果一开始决定不了用的词汇，那么像 Delegator、XmlMarkup 一样，可以用 $\text{method}_{\text{missing 这个}}$ 方法来动态地追加和利用词汇。

3.5.14 2.6.13 层次数据的实现

最后说明一下表 2-4 最后的层次数据。前面的 XmlMarkup 就是层次数据，我们可以再看一下图 2-42。程序本身看起来只不过是以块为参数嵌套调用方法，而外观和功能都漂亮地实现了数据的层次化。

3.5.15 2.6.14 适合 DSL 的语言，不适合 DSL 的语言

总的来看，Ruby 是非常适合 DSL 的语言。

首先，调用方法的时候可以省略括号，这些表现的多样性使得程序看起来像是在使用声明一样。DSL 的必要功能很多都用数据构造、设定等声明来定义，所以 Ruby 对声明的支持是很重要的。

其次，元编程功能可以获取并更新程序的信息，所以不必使用预处理和宏就可以实现 DSL 必要的功能。像这样不用对语言进行扩展就可以实现 DSL 的方法也称为“语言内 DSL”。

适合语言内 DSL 的编程语言不只是 Ruby。对 Ruby 影响很大的 Lisp 和 Smalltalk 也和 Ruby 一样适合 DSL。特别是 Lisp, 原则上固定语法只有 S 式这种数据结构表现形式，几乎是可以构成任何语言内语言。

Ruby 可以用 eval 处理字符串来生成程序，Lisp 则可以用宏处理列表来实现对程序的处理。所以有人把 Lisp 称为 Programmable Programming Language（可编程的编程语言）

Smalltalk 虽然不像 Lisp 那样极端，但是它的动态性并不比 Ruby 差，另外还具有元编程的功能。Smalltalk 的控制结构本来就是用块来实现的，所以语法扩展也是自然天成的。

反之，有的编程语言，如果不用别的方法就不容易实现 DSL。比如 C++、Java 和 C# 等语言就不能像 Ruby、Lisp 和 Smalltalk 那样实现 DSL。

但这并不是说这些语言中不能使用 DSL 方法，比如自动生成程序代码。首先定义好 DSL 用的小语言，然后编译成 C++、Java 和 C# 等目标语言。

编译器经常用到 Ruby 这种具有优秀文本处理功能的语言。Code Generation in Action[29]一书讲解了这个问题。

另外一个实现 DSL 方法的例子是解释器。比如开发应用程序时从设计到实现是很复杂的，可以利用固定的语法和类库函数来读取程序。

具体的方法是用 XML、DOM(Document Object Model) 等 XML 处理类库来解释小语言语法。这是 Java 应用程序的配置文件采用 XML 的原因之一。通过 XML 文件，不用每次编译 Java 程序就可以改变配置，定制程序的行为。

这种用法可以把 XML 称为 Java 界的 DSL，或者是 Java 应用程序的脚本语言。

[29]Code Generation in Action,Jack Herrington 著，ISBN 1-930-11097-9。主要讲解 Java 程序的代码生成。代码生成程序都是用 Ruby 写的，所以也可以说它是并未明说的 Ruby 书。