Ruby 再入门

Delton Ding

Version 0.1.0, 2019-02-08

目录

简介

一本关于重新思考编程入门的教程。

如何编译这本书?

这本书使用 Asciidoc 标准编写。编译这本书需要有 Ruby 的环境。

使用 `bundle install` 安装工具，并使用 `bundle exec rake build` 进行编译。

编译完成的电子书在 `./build` 目录下。

如何给这本书提供意见/修改?

本书使用 Git 进行维护，并使用 GitHub 对 Git 仓库进行管理。请使用 GitHub 的 Issue 和 Pull Request 系统为本书提供意见和修改。

在线阅读

restartruby.com

下载

- PDF 版本
- EPUB 版本

鸣谢

这本书的写作过程和发行过程离不开下面这些人的帮助，谢谢他们。

写在一切之前

为什么要写这本书?

市面上关于编程入门的书多到无法想象，但是其中令人满意的却很少。事实上，写一本面向初学者的书比写一本面向专家的书更困难。面向专家的书籍，最重要的是专业性和正确性，而面向初学者的书籍在此基础上还要加上通俗性。市面上的一些书籍，在通俗性和正确性上往往二者不能兼顾。

对于初学者来说，给予错误资讯的危险性是极大的。如果你今天拿到一份乐谱准备练琴，练了一个月发现，调号就没看对。等你再去纠正，形成了肌肉记忆的手再想纠正回来就必须付出更多的代价。我虽然对于 Ruby 编程也算颇有经验，但也绝不能说内容 100% 的正确性。我也不敢像 Donald E. Knuth 老先生那样为第一个发现错误的准备 2.56 美金，此后每发现一个翻倍，这样我离破产基本上是不远的。维护这本书正确性的核心，一方面是反复校稿，其次是维护了一套 CI 系统来自动化测试书中代码行为是否和描述一致，但更重要的就是秉持开放的态度，基于 GitHub 进行协作。欢迎大家自由为书籍纠错、 提意见。

至于通俗性，我觉得大家要有对编程难度的一个正确认知。如果你觉得编程对你很难，这里的「难」指的是「问题困难」还是「问题复杂」。「问题困难」往往是一些计算机科学学科强相关的内容，比如算法、数据结构、形式语言；而「问题复杂」常常是工程问题，里面的内容通常是你所熟知的，唯一的问题是你没有办法很好解构这个问题，将复杂的实际问题转换成一系列你熟悉的小问题。

我非常喜欢陶哲轩先生所写的《实分析》，因为他把一个「困难」的数学分析问题讲得很简单易懂，和他比起来我还差太远。好在这本书所涉及的问题，几乎都是后一类问题，很少会涉及前一类问题。所以大可不必妄自菲薄，你一定具备理解这本书内容的前置知识。同时，这本书会在每个章节后通过一系列复杂性逐渐上升的综合练习，让你可以慢慢解构问题，最终能够独当一面完成一个复杂的项目。

什么人适合读这本书?

- 想要了解编程的人
- 想要将编程作为自己工作而入门的人
- 想要学习 Ruby 编程语言的人
- 有一定编程基础，想尝试 Ruby 语言的人
- 有一定 Ruby 基础，想进一步理解 Ruby 的人

什么是 Ruby?

Ruby 是最初由 Matz（Yukihiro Matsumoto）于 1993 年开发，现在作为开源软件开发的语言。它可以在多个平台上运行，并在世界各地使用。尤其适合于网站的开发。

为什么选择 Ruby?

因为我喜欢 Ruby。

Ruby 是一门适合用来作为入门教学的语言。使用这门语言不需要对计算机组成原理有着比较深刻的理解。使用这门语言进行教学，我们不但可以学习形如面向对象等在工程中常用的范式，还可以掌握形如 Lambda 演算、元编程这样非常 Lisp 的特性。站在功利的角度来说，Ruby 不是一门非常大众的语言，虽然可能岗位相比 Java、PHP 少很多，但是相对的竞争的求职者也更少。但是，编程的工具绝不是一招鲜吃遍天的，10 年前所流行的框架放到今天可能多半已经被淘汰了。如果单靠一门技术就想在这个行业活到退休是非常困难的，但是这些框架背后的思想确是共通的。

Ruby 的创始人 Matz 曾经说过：「Ruby 的首要设计目标是让这个世界上每一位程序员都变得充满生产力，享受编程，以及变得开心。」无论你最后会不会选择 Ruby 作为工作，都不妨碍我们用 Ruby 作为教学语言。

怎么阅读这本书?

我把这个书的设计理念定义为「在两座山脊之间寻找山谷」。这两座山脊分别是站在语言学习者角度语法使用的山脊，以及站在语言设计和实现角度对语言特性思考的山脊。

所谓「学而不思则罔，思而不学则殆」。如果只学习基本的语法使用，没有认清很多设计面向的场景，没有理解语言的思考方式，不但写出来的代码非常脏乱差，而且实际使用的时候往往不知从何下手。

本书按知识点的逻辑关系进行排序，因此难度上可能存在跳跃。在一些章节，本书会使用如下的提示来提醒初学者。

> 本章节涉及较为进阶内容，建议第一次接触编程或缺乏编程经验的开发者暂时跳过这一章节。

除了读之外，一定要 **练习**。软件工程是工程学，是由大量的细节组合起来的工程。如果看完书中的内容，不上机练习，不多加尝试，不勤加思考，那么收益必然是零。因为会有大量的问题在没有尝试之前不会发现，等把书看完了，除了掌握了基本概念，剩下的必然都还在云里雾里。

遇到问题怎么办?

如果你从「○家」买了一套家具回家，发现不会安装怎么办？你必然是先要仔仔细细检查安装说明书，如果实在不行，再打电话给「○家」，写代码也是这样。大家的时间通常都是有限而宝贵的，如果遇到什么鸡毛蒜皮的事情，都问别人，很可能会得到一句愤怒的 RTFM 作为回答。这是 Reading the F**king Manual 的缩写，意思是「去读＊＊的手册」。而且这也会影响到自己独立解决程序问题能力的养成，绝对不是好习惯。对于遇到程序问题，一般的解决流程是：

1. 检查开发手册。对于 Ruby 这一手册通常是 Ruby 的 官方文档，对于 Linux/UNIX 系统调用，手册可以通过 `man 调用名称` 和 `info 调用名称` 的系统命令来查询。
2. 使用搜索引擎查找问题。除非你在使用非常先进的技术，通常你不会是第一个遇到这个问题的。许多网站，例如 stack overflow 就大量收录了程序相关的问题。上网找找，有极大的概率有人会遇到和你同样的问题，并且已经有解决方案了。
3. 问问小黄鸭。很多看起来很奇怪的问题往往来源于粗心。放一只橡皮小黄鸭在桌子上，给它一行一行耐心解释你所写的代码，你可能会突然发现问题所在。
4. 询问行业专家或开发者。如果真的不幸全部都没有解决问题，确实有必要找个人问问。在网上问问题是个很好的途径，一方面你可以问到平时不容易接触到的领域专家，同时也可以留下记录帮助更多人。但是注意，一定要把自己问题的触发条件、相关的环境配置、代码、遇到问题的详情好好描述出来。如果就说一句「我做了 xxx，它不工作」，别人就算看到也无从下手。具体如何提问可以参考 stack overflow 的《How do I ask a good question》。

豆知识：`man` 命令和 `info` 命令有什么区别?

`man` 是在相当早年的 UNIX 系统中就已经内建在系统中了。而 `info` 则是 GNU 项目的文档系统。`info` 使用 Texinfo 作为其源文件的形式，提供了更加丰富的文本格式，通常内容也会比 `man` 来得更完整。但在长长的 `info` 页面中通过快捷键导航快速找到自己想要的东西也是一件需要熟悉的事情。

闲聊：如果我觉得 Manual 的缩写 `man` 命令冒犯到了我的性别平等主义怎么办?

如果你是用 zsh 作为自己的 shell，zsh 有非常有用的 `alias` 功能可以来解决这一问题。把这些东西加入到你的 zsh 配置文件中就好：

```
alias man="info"
alias woman="info"
alias lgbt="info"
alias lgbtq="info"
alias lgbtqi="info"
alias lgbtqia="info"
alias lgbtqiap="info"
alias lgbtqiapk="info"
```

第一周：Hello World 再入门

简介

Ruby 的 Hello World 怎么写?

```
puts 'Hello World'
```

学会了，今天也是努力的一天，睡觉吧。

等一等！睡太早啦！

什么是 Hello World?

Hello World 就是在屏幕上显示一行 Hello World 的最简单的程序。这通常是程序员接触一门新语言时所需要写的第一个程序。

为什么要学习 Hello World?

Hello World 程序的意义绝对不单单是打印一行「Hello World」这么简单。其核心是用来确认程序开发环境和运行环境是不是安装妥当，同时也能对语言的开发进行一个初步的认识。砍柴不误磨刀工，那我们花一些时间，仔细探讨 Ruby 的开发、运行环境配置。

让我们一起开心地开始吧。

环境搭建

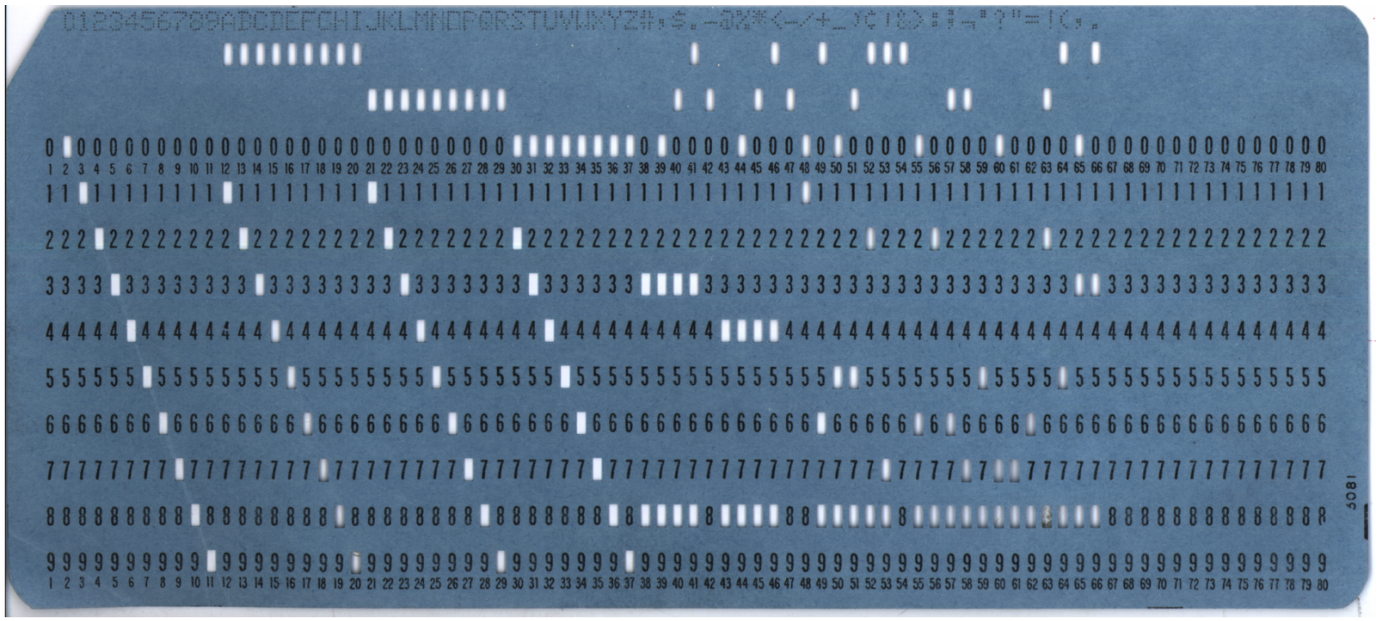
了解 Ruby 解释器

什么是解释器?

开发 Ruby 程序最重要环境的是 Ruby 解释器。「解释器」是在解释什么？解释前和解释后的东西是什么?

了解解释器，我们需要先了解另一个相近的概念「编译器」。在上世纪计算机刚刚出现的时候，程序员编程需要

依赖纸带打孔卡或类似的穿孔纸带。



这里的打孔与不打孔就被作为一系列「开」和「关」信号传给电脑，形成程序交由计算机执行。今天的计算机已经淘汰了打孔纸带，计算机的启动和运行流程也变得越来越复杂，但程序依然是一系列「开」和「关」（1 或 0）的二进制信号。直接编写这些二进制，对于今天越来越复杂的程序开发要求来说，已经变得非常困难，于是我们发明了更接近于自然语言的高级语言。Ruby 就是这样一门高级语言。

高级语言不能直接被计算机执行，但是我们可以先通过一个「编译器」程序，读入这些高级语言，然后翻译成机器可以执行的二进制。「编译器」通常会把整个程序在运行前就完全编译到机器可以执行的二进制。「解释器」会在运行过程中，把程序一行一行直接翻译执行。解释器通常会比编译器执行得更慢，但同样也带来了更大的动态特性，允许更自由地开发方式。而 Ruby 的执行环境，通常就是这样一个「解释器」。

Ruby 解释器有什么?

Ruby 解释器是 Ruby 语言开发的核心。Ruby 是一个开放的语言，任何人都可以为 Ruby 实现自己的解释器。Ruby 的解释器多种多样，常见的仍在维护的解释器有：

- Ruby
    - Ruby MRI (CRuby)
    - JRuby
    - TruffleRuby
    - Rubinius
    - RubyMotion

    - Opal
- mruby
    - mruby
    - mruby/c

这些 Ruby 解释器各有一些差异，支持的语法和执行的性能也并不完全相同。本书所涉及的全部 Ruby 都指官方实现的 Ruby MRI。

MRI 代指 Matz’s Ruby Interpreter，即 Ruby 创始人松本行弘最早实现的 Ruby 解释器，是 Ruby 的官方解释器。虽然在 Ruby 1.9 之后的版本中，官方已经把虚拟机换成了由 Kiochi Sasada 主导的 YARV (Yet another Ruby VM) 解释器。但在 1.9 版本后，YARV 已经被合并到 MRI，此后我们已不特别区分 MRI 和 YARV 了，现在仍称呼这一解释器实现为 Ruby MRI。下文所有的 Ruby 解释器，如无特殊标注，都是 Ruby MRI 解释器的缩写。

在撰写本章节的时候，Ruby 的最新版本是 2.7.0，本书全部的代码都在 2.7.0 中进行过测试。我们会自动化测试本书代码在不同环境下的兼容性，以确保正确性。

安装方法

Windows 用户

在 Windows 上安装 Ruby 最简单的方式是 RubyInstaller。在本书的编写过程中，我们会测试书中所有涉及到的程序在 Windows 上的兼容性。但由于 Ruby 的第三方依赖，特别是一些设计给 *NIX 服务器的依赖程序，可能没有测试在 Windows 上的兼容性，从而可能在使用上会遇到一定的困难。

一些教程不推荐新手在 Windows 上开发 Ruby。但事实上，Ruby 的标准库对于 Windows 的兼容性还是相当良好的。如果没有人在 Windows 上使用 Ruby，那么 Ruby 运行在 Windows 上的问题会变得更多，这是一个恶性循环。但是对于初学者，使用 Linux/macOS 进行开发依然是我个人推荐的。主要问题是，初学者缺乏对环境问题处理的经验，遇到问题往往会不知所措。大多数服务器软件的生产环境更愿意使用自由的 Linux 操作系统，而使用 Ruby 开发服务器应用是最常见的用途，使用和生产环境一致的环境，至少是 *NIX 环境能有效避免问题发生的概率。

关于 PC 用户如何选择和安装 Linux 发行版，本书单独开设附录章节来描述，请参阅《附录一：新手如何安装 Linux 开发版？》。另外，在 Windows 10 中可以使用 Windows Subsystem for Linux (WSL) 来产生一个无缝的 Linux 环境。详情请参阅微软的 WSL 官方文档。

*NIX 用户（Linux、macOS、BSD 用户）

RVM

RVM 是最推荐新手安装 Ruby 的方法。笼统来说，使用 RVM 安装 Ruby 需要三步。如果你是 macOS 用户，你可能需要先安装 XCode 和 XCode Command Line Tools 才能安装 RVM。

XCode 可以从 App Store 直接下载到，安装完成后打开「终端（Terminal）」应用，使用 `xcode-select --install` 安装 XCode Command Line Tools。

对于 Linux 用户，请确认自己的 Terminal 是 login shell 的模式。一些 Linux 发行版自带的 Terminal 应用没有 login，可能没有加载用户的环境变量。

第一步是获取 GPG 公钥，RVM 使用 GPG 密钥系统来确保程序在传输过程中不被篡改。打开终端应用，输入下面的命令即可获取 GPG 密钥。

```
gpg2 --keyserver hkp://keys.gnupg.net --recv-keys
409B6B1796C275462A1703113804BB82D39DC0E3 7D2BAF1CF37B13E2069D6956105BD0E739499BDB
```

第二步则是一键安装脚本，下面的命令会下载 RVM。

```
\curl -sSL https://get.rvm.io | bash -s stable
source ~/.rvm/scripts/rvm # 载入 RVM 环境（新开 Terminal
就不用这么做了，默认自动重新载入的）
```

对于中国大陆地区用户，RVM 的自带镜像源可能下载速度太慢。为此 Ruby China 提供了镜像源，在执行第三步前可以使用

```
echo "ruby_url=https://cache.ruby-china.com/pub/ruby" > ~/.rvm/user/db
```

来切换镜像源。

最后一步就是执行下面命令来安装特定版本的 Ruby。

```
rvm install 2.7.0 # 这里的 2.7.0 可以切换成阅读本文时最新的 Ruby 版本。Ruby
最新版本可以在 https://www.ruby-lang.org/ 确认。
```

安装成功后可以使用

```
ruby -v
```

来检查所安装的 Ruby 版本有没有正确安装成功，如果返回了版本号，那么就是安装成功了。

详细的安装文档可以在 rvm.io 来查询。

snap

Ruby 在 2018 年 11 月 8 日加入了官方的 snap 套件支持。如果你使用 Ubuntu 16.04 的后续版本，你可以一键安装 Ruby。

```
sudo snap install ruby --classic
```

使用 Snap 安装的 Ruby 可能会在环境变量上需要一些额外的配置。建议检查官方的 新闻 来确认细节。

编译自源代码

编译自源代码是较为困难的安装方法，不建议新手使用这一方式。

在电子游戏《尼尔：机械纪元》的 用户协议 中，我们会发现出现了 Ruby License。可见在这款游戏中使用了 Ruby 语言实现了一定的功能。这款游戏首发在 PS4 平台上，而 PS4 的操作系统是一个修改自 FreeBSD 操作系统。所以 Ruby 语言对于 BSD 系的操作系统同样是非常友好的。

但如果你想在一些嵌入式设备上运行 Ruby 或者需要运行在 PS4 上，使用包管理器可能不是一个好主意，因为你不一定具有全局安装的权限或者不想引入额外的复杂度。这时候直接从源代码编译可能就变成了必须。

生日快乐！欢迎自己编译你的蛋糕。 (Photo by Monika Grabkowska on Unsplash)

从源代码编译安装很简单，你可以先从 Ruby 官方网站下载 最新的源代码，在解压后执行：

```
./configure
make
```

进行编译。Ruby 的编译过程中，一些组件是可选的，你需要自己确认这些可选的依赖是否准备妥当。如果编译后需要安装，你可以执行：

```
sudo make install
```

进行安装。

内置包管理器

使用例如 Ubuntu、Debian 内置的 `apt` 或者 CentOS、Fedora 内置的 `dnf` 或类似方法也可以很方便安装 Ruby。但是，大多数操作系统内建的软件源常有版本滞后、缺少组件的问题。如无必要，不推荐新手使用这样的安装方法。但如果你有一个可控、可信、维护良好的软件源，这也是一个不增加复杂度安装 Ruby 的好方法。

如何管理 Ruby 版本

使用 RVM 的一个好处是可以很好管理不同的 Ruby 版本。刚开始写 Ruby 的时候可能不太会意识到这件事情的重要性。但是下面这几种需求在实际开发中可能会发生：

- Ruby 版本更新了，怎么升级？
- 想临时测试某一特定版本 Ruby 的特性，怎么临时切换？
- 如何指定项目的 Ruby 版本，来确保服务器运行环境和开发环境一致？

如何安装新的 Ruby 版本？

在安装之前，你可以先用下面命令检查可以安装的 Ruby 版本：

```
rvm list known
```

如果你想要安装的版本没有出现在这个列表中，你也许需要更新 RVM。更新 RVM 的方法非常简单：

```
rvm get head
```

和你安装第一个 Ruby 版本一样，安装新的 Ruby 版本的方法完全一样，比如我想安装 2.7.0 版本，就可以使用下面命令。

```
rvm install 2.7.0
```

如何切换 Ruby 版本？

如果一个常见的需求是安装新 Ruby 版本后希望把默认 Ruby 版本切换到新安装的版本上，命令则是：

```
rvm --default use 2.7.0
```

如果你只是需要在当前 Shell 环境下临时切换，不需要设置成默认，只要把 `--default` 参数拿掉即可：

```
rvm use 2.7.0
```

切换后，可以用

```
ruby -v
```

确认切换后的 Ruby 版本。

如何设置项目的特定 Ruby 版本?

在项目根目录下放置一个名称为 `Gemfile` 的文件，并在里面写入如下的内容：

```
ruby '2.7.0'
```

RVM 就会在 shell 切换到这个目录下之后自动切换当前的 Ruby 版本。这样的设置还有一个好处，就是著名的 Ruby 托管平台 Heroku 也是使用这一方法来切换 Ruby 版本的。如果之后你需要将自己制作的网站托管到 Heroku 上的话，可以利用这一特性自动设置 Heroku 上的 Ruby 版本。

Gemfile 是 Bundler 提供依赖管理的重要文件，有关于这方面的功能，我们会在「第六周：Ruby 工程化入门」中重点介绍。

由于 Ruby 在设置版本上并没有官方制定的标准。管理项目版本依赖于不同版本管理工具的具体实现。在 RVM 中除了上面的方法，还有 4 种方法可以设置项目的 Ruby 版本。具体可以参考 RVM 的 Typical RVM Project Workflow。其中使用 `.ruby-version` 设置 Ruby 版本的方法可以同时在 RVM 和另一个在 Ruby 上常用的版本管理工具 rbenv 通用。

REPL 和编辑器

REPL

使用 `info irb` 来查看 `irb` 命令的帮助，我们会看到 `irb` 是「交互式 Ruby 壳程序」（Ruby Interactive Ruby Shell）的缩写，所谓的交互式就是 Ruby 程序的 REPL 环境。

REPL 是一个在 Lisp 和受其影响的语言中非常常见的概念，是「读取-求值-输出 循环」（Read-Eval-Print Loop）的缩写。启动 REPL 程序后，你可以立刻运行你的各种代码，而无需新建一个文件编辑后再去执行。这对于测试、调试非常实用。

对于 Ruby 来说，在终端程序中输入 `irb` 按下回车后，Ruby REPL 程序就会启动。

```
❯ irb
2.7.0 :001 >
```

在这个环境里输入 `puts 'Hello World'` 然后回车，就会立刻看到屏幕上打印了 Hello World。Hello World 两侧被「单引号（'）」包围，来告诉电脑这里是一个字符串，而不是代码。**注意：这里的单引号是英文输入法下的傻瓜单引号（'），而不是中文的单引号（ ‘’ ）**

```
❯ irb
2.7.0 :001 > puts 'Hello World'
Hello World
 => nil
2.7.0 :002 >
```

在这个环境里输入 `exit` 然后回车则会退出。

REPL 叫「读取-求值-输出 循环」，我们就来分别看一看这四个步骤：

1. 读取：REPL 程序启动后，就会等待你输入新的代码。
2. 求值：当输入 `puts 'Hello World'` 后，程序就会进行求值。`puts` 方法接收一个字符串（一串字符）作为参数，在屏幕上打印出来并自动换行。然后返回值为「空」。
3. 输出：REPL 会自动把求值结果输出打印到屏幕。由于 `puts` 的返回值是「空」，所以这里打印 `nil`。
4. 循环：完成输出后回到等待读取状态，等待下一条代码的输入。

豆知识：`nil` 还是 `null`?

Ruby 在返回值是「空」时，返回 `nil`。你可能在其它语言中会发现会用 `null` 或者 `NULL` 关键字来表示类似的含义。事实上，它们确实是相同的。这两个词语都是拉丁语词源的。

`nil` 来自于拉丁语 nīl，是 nihil, nihilium 的缩写。其中 ni 是 ne- 词缀的变形，表示「没有 (not)」。而 hilium 是「一点点 (a little, a trifle)」。合起来「没有一点点」，就是「完全没有」的意思。

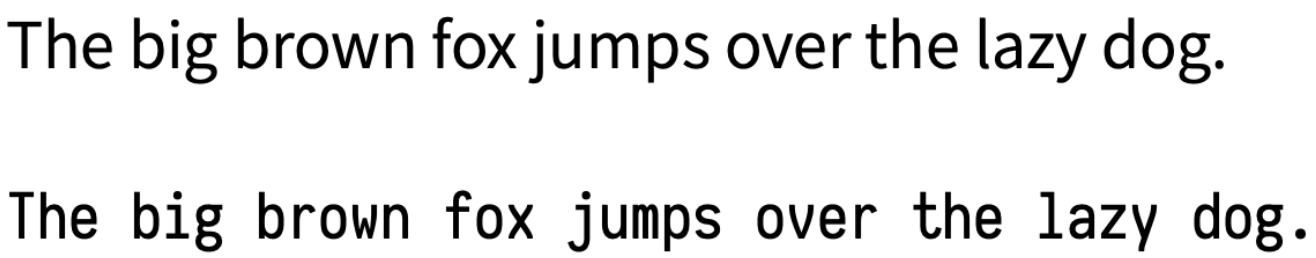
`null` 来自法语 nul，而 nul 来自于拉丁语 nūllus。其中 n 同样是 ne- 词缀的变形，表示「没有 (not)」。而 ūllus 则是「任何（any）」的意思。合起来「任何一个都没有」，同样也是「完全没有」的意思。

编辑器

如果要写一个复杂的程序，使用 REPL 可能还是太过牵强了。我们还是要准备一个编辑器，提前编辑好文件再来执行它。事实上，任何纯文本编辑器，包括 Windows 自带的那个记事本，都可以被用来编辑 Ruby 代码。但是，我们没有必要折磨自己，写代码应该是一件让人开心的事情。使用专门为代码编辑设计的编辑器对于程序开发还是有很多好处的。下面是一些代码编辑器会提供的关键特性：

等宽字体

代码编辑器通常都会默认使用等宽字体作为默认字体，其特点是无论是拉丁字母还是常用的西文符号的字符宽度都是一致的。



在这张图中，上方的思源黑体 **不是** 一款等宽字体，字形的宽度会变化。而下方的 Iosevka 字体是一款等宽字体，字形的宽度是固定的。如果是阅读长篇文章，非等宽字体也许会更加美观好读。但是代码是具备功能性的问题，有时代码的对齐对于调试时的可读性有很大的帮助，所以 **强烈推荐** 使用等宽字体来进行编程。

对于 CJK（中文、日文、韩文）开发者，我个人非常推荐使用 更纱黑体 作为编辑器的字体。更纱黑体是 Iosevka 和思源黑体的结合字体。因为 Iosevka 字体设计的宽度正好是思源黑体中汉字、平假名、片假名和谚文宽度的一半，即使文字混合同样可以对齐。

语法高亮

语法高亮指的是通过不同颜色来标记代码中不塌缩的关键字从而提高代码的可读性。比如下面两段一样的代码：

```
def abs(num)
  return num if num > 0
  -num
end
```

```
def abs(num)
  return num if num > 0
  -num
end
```

显然下面那种读起来更加舒适。花花绿绿是一种硬需求，以至于在 Ruby 2.7 的 `irb` 中，默认也支持了语法高亮。

代码补全

代码补全指的是编辑器能根据你输入一半的关键字来自动推测你的代码，从而给予智能的补全提示。这在一些静态类型语言中会有相当高的准确率，在一些 API 名称长度非常长的语言或框架（例如 Java 或苹果的 Cocoa 框架）中会显得特别实用：

XCode 中 Swift 语言的自动代码补全

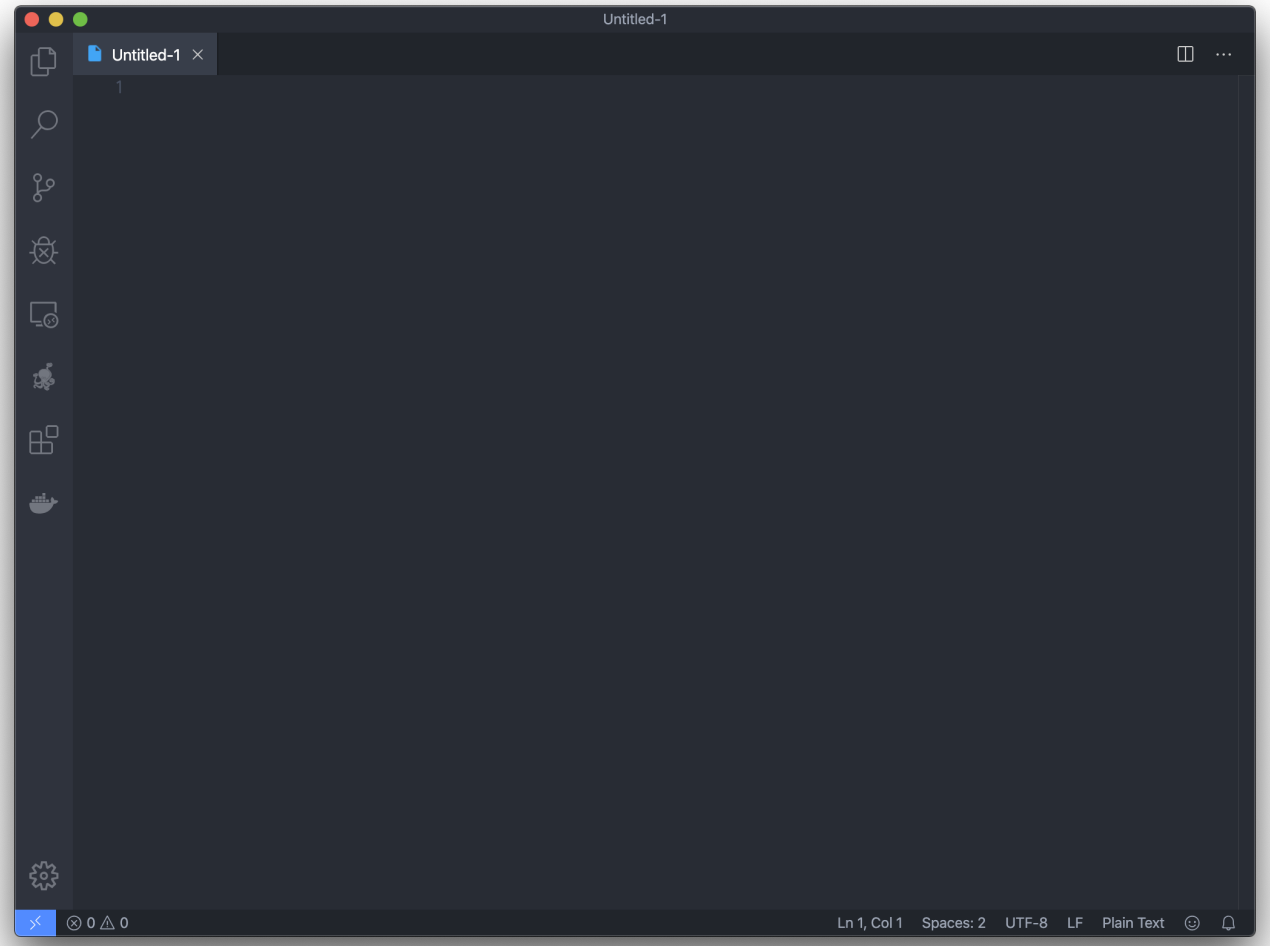
Ruby 是一门动态语言，大多数的代码补全的准确率没有那么好，目前有一些基于人工智能的全新尝试，例如「TabNine」代码补全插件，但还在比较早期的阶段。好在 Ruby 的 API 通常都比较短，也不算是太大的问题。

项目组织管理

> 虽然文件管理在早年的编辑器中不是必需。但在今天工程复杂度越来越大的大前提下变得越来越重要。内置文件、目录的管理，甚至是和 `git`
> 等版本管理工具进行结合对于提升代码开发效率也有着相当的帮助。

编辑器选择

现代的代码编辑器，我个人推荐使用 Visual Studio Code。Visual Studio Code（以下简称 vscode）是微软近年来推出的罕见在口碑上成功的产品。vscode 是一款开源、跨平台的文本编辑器。其本身非常轻量，安装非常方便，同时也有丰富的插件系统支持。只需安装 Ruby 插件，即可满足 Ruby 开发大多数的需求。和其主要竞争者 Atom 比较，vscode 的插件系统实现非常高效，编辑器不容易被插件拖累性能而变得卡顿。



纯命令行（CLI）下的代码编辑器，Vim 和 Emacs 都有其非常坚持的使用者。这两款编辑器仿佛两个宗教，在网上时不时都会引来支持者们疯狂地辩论。但不得不说，这两款编辑器都有着非常高的效率（如果你熟悉各种快捷键和插件配置的话），但这两款编辑器同样都有着极高的学习曲线。如果你考虑把终端作为自己的主要使用应用的话，熟悉其中的一款是非常有效的。但如果你只是临时、极少使用终端上的文本编辑器的话，使用操作傻瓜 nano 也不失为一个好选择。

另一个选项则是 IDE，IDE 是集成开发环境（Integrated Development Environment）的缩写。代码补全、整合的调试开发环境、项目组织管理、版本管理曾经是 IDE 最明显的特点。但现代的文本编辑器像是 vscode 同样支持了这些特性，让编辑器和 IDE 之间的界限愈发模糊。如果想用 IDE 开发 Ruby，目前最好用的工具是由

JetBrains 推出的 RubyMine。RubyMine 对 Ruby
上常见框架都实现了自己的类型补充，使得代码自动补全变得比较可用。同时 JetBrains 祖传的重构（Refactor）工具对于大型项目的开发也是比较高效的工具。

试一试

本文使用 vscode 作为编辑器的例子，其它编辑器同理。创建一个文件，保存命名为 `hello.rb`（`.rb` 是 Ruby 的默认后缀名）。在里面输入如下的代码：

```
puts 'Hello World'
```

打开终端应用（Windows 下是 `cmd`），利用 `cd 路径` 切换至这一目录，使用 `ruby hello.rb` 执行。

```
❯ ruby hello.rb
Hello World
```

你便成功运行了这一 Ruby 文件。尝试改一改这一 Ruby 文件，再多运行几次来体验一下 Ruby 程序的运行吧。

小练习

1. 在屏幕上打印你的名字。
2. 分两行分别打印你的「姓」和「名」。

豆知识：Ruby 中打印输出有哪些常用方法？

在上文中我们使用的是 `puts` 命令来打印的。Ruby 中常用的打印输出到屏幕的方法有：

- `puts`
- `print`
- `p`

`puts` 和 `print` 的差异是显而易见的，`puts` 会自动在打印完成后换行，而 `print` 不会。`p` 则比较复杂，`p foo` 类似于 `puts foo.inspect`，`#inspect` 是 Ruby 中查看某一个对象内部结构的方法。比如：

```
p 'Hello World'
```

打印的是 "Hello World" 而不单单是 Hello World。这一对引号即强调了这个对象是一个字符串，串的内容是 Hello World。另外一个差异是，`puts` 和 `print` 都是返回 `nil` 的，但 `p` 会原样返回，这一特性可以用来非常方便地调试程序故障。

猜一猜运行下面的代码，打印结果是什么？

```
print 'Hello'
puts 'Hello'
p 'Hello'
```

运行一下看看和自己猜的是不是一样，并尝试来解释一下为什么。

变量与基本算数

整数运算

既然计算机叫计算机，那么我们还是从「计算」作为入门的第一步吧。

四则运算在 Ruby 语言中的表达几乎和现实世界算数的表达式一样。小小的不同是，由于乘法 × 和除法的符号 ÷ 在键盘上不方便输入，于是使用了另两个符号 * 和 / 来替代乘和除。

特别注意

除法符号是 / 不是 \。

```
1 + 1 # => 2
1 - 5 # => -4
3 * 7 # => 21
6 / 2 # => 3
```

需要注意的是，整数之间的加减乘除的结果必然还是整数。如下面的代码所见，整数间的除法运算实际上是整除。

```
5 / 2 # => 2
```

如果需要获得整除后的余数，Ruby 提供了模除运算 %：

```
5 % 2 # => 1
```

需要特别注意的是，在 Ruby 中对负数取余数的结果可能和其它语言不一样。Ruby 中余数的符号 **永远和除数一致**。

```
5 % 2 # => 1
-5 % 2 # => 1

5 % -2 # => -1
-5 % -2 # => -1
```

这一设计带来了显而易见的好处，在判断一个数字是不是奇数的时候我们只需要 % 2 检查结果是否为 1 即可。而无需确认其正负。但坏处也是很明显的，如果你尝试从其它编程语言移植程序到 Ruby 上需要格外小心对正负数取模的处理。

Ruby 的整数没有无穷大和未定形式，如果整数除以零会直接抛出程序错误。如果没有处理错误，在程序运行中会直接退出。

```
3 / 0 # => ZeroDivisionError (divided by 0)
```

Ruby 另一个特色是大整数运算。不同于 C、C++ 语言中整数类型拥有固定的内存分配大小（上限和下限），Ruby 的整数类型的大小是会动态调整的。这使得你可以自由处理非常大的数字而无需考虑溢出问题。

例如 C++ 语言中的下面代码：

```
#include <iostream>

int main() {
  std::cout << 1000000000000000000 * 1000000000000000000 << std::endl;
  return 0;
}
```

会因为内存溢出，给出一个不符合数学常识的 687399551400673280 结果。而在 Ruby 中你完全不用担心这一点。

```
1000000000000000000 * 1000000000000000000 # => 1000000000000000000000000000000000000
```

这一特性在幂运算时特别实用。Ruby 的幂运算符号是 **。a**b 相当于 a^b。

```
2**256 # =>
115792089237316195423570985008687907853269984665640564039457584007913129639936
```

局部变量

变量 (variable) 是程序开发的基础概念之一。变量是一种对临时存储的抽象概念。Ruby 中的变量有 4 种类型：

- 局部变量 variable
- 实例变量 @variable
- 类变量 @@variable
- 全局变量 $variable

这里所介绍的变量是局部变量，至于其它变量的时候，我们会在之后章节中再做介绍。

Ruby 的局部变量以小写字母或下划线 _ 开头。Ruby 中的 = 符号和数学中的等于不同，表示「赋值」符号。表示将 = 符号 **右边** 的结果保存到 **左边** 的变量里。数学中等号两边内容可以互换，但在赋值中是 **绝对** 不能的。

在 irb 中试一下下面的代码，来理解局部变量和 = 的使用。

```
a = 1
a + 1 # => 2
a + 3 # => 4
a = 2
a + 1 # => 3
a + 3 # => 5
2 = a # SyntaxError
```

试一试

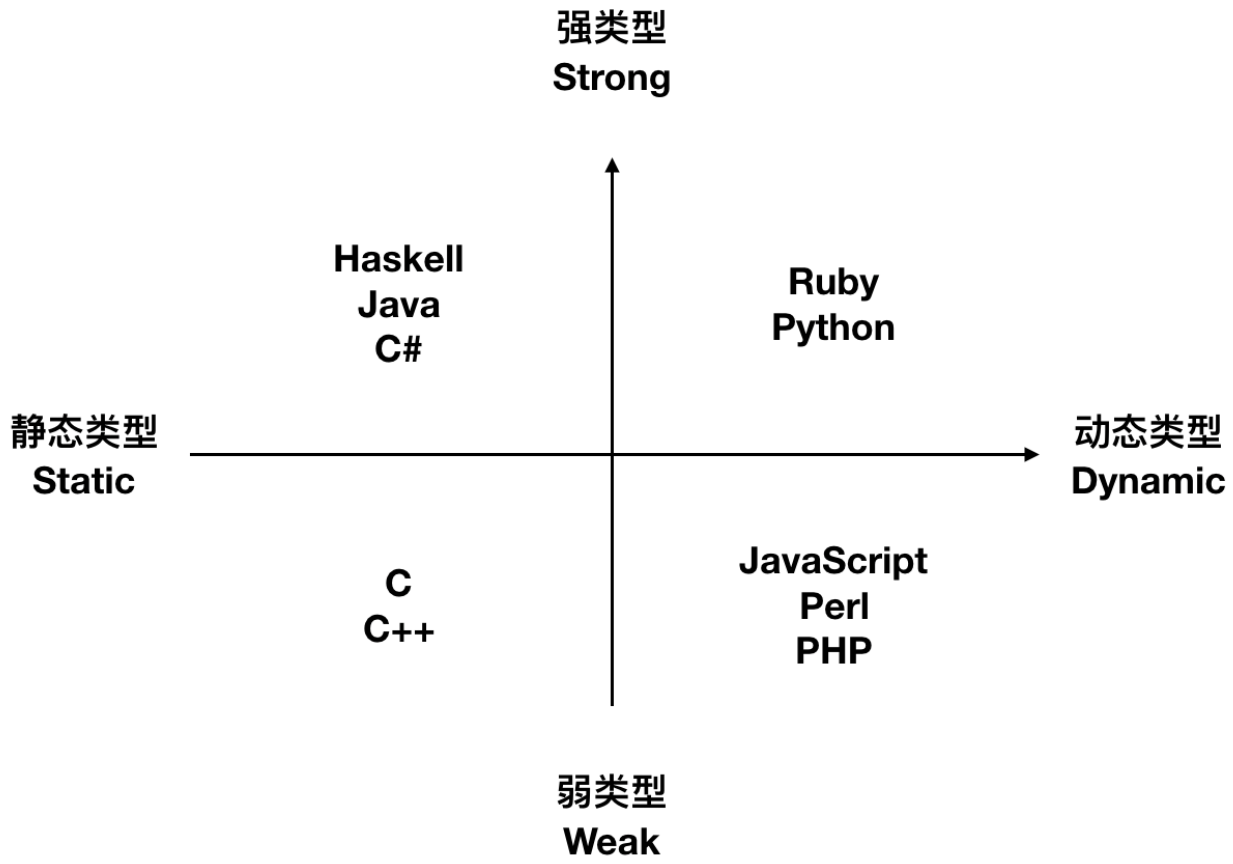
- 创建两个变量 a 和 b，给它们赋两个整数，计算并打印它们的和。
- 创建两个变量 a 和 b，给它们赋两个整数，计算并打印它们的乘法。
- 创建三个变量 a 和 b 和 c，给它们赋两个整数，计算它们的和，将他们的结果写入 c 变量中。

答案见《附录三：参考答案》中的 局部变量 小节

Ruby 的类型系统

类型系统自古以来都是高级语言的一个非常重要的属性。所谓类型，指的是程序语言中变量、常量的数据类型，类型是对数据的约束。比如我们可以检查这一变量是数字还是字符串，从而避免超出我们预料的行为。

基于不同的分类方法可以把不同语言的类型系统分成不同类别。一般我们会说 Ruby 的类型系统是一个动态类型、强类型系统。其中动态类型与静态类型相对，而强类型与弱类型相对。类型的动态与静态、强与弱是一对正交（orthogonal）的概念。



理解这一问题，我们先要理解类型可能的错误，以及错误带来的问题。

一类问题是出现在类似于 C 语言中的内存管理带来的错误。比如说根据某一变量的类型分配了一定空间，但是在写入的时候，占用了大于既定的空间。C 语言不会检查这一类的错误，从而造成程序写入到了其它变量 / 程序本来占用着的空间，进而产生完全未知的行为。这一问题称为「缓冲区溢出」，这会带来未知的程序错误，同时还有可能被黑客巧妙利用从而进一步攻击你的程序。一个典型的案例是 SONY 在 PSP-3000 游戏机上的相册应用存在一个「缓冲区溢出」漏洞，通过一张特殊的图片，黑客实现了对 PSP-3000 的破解，从而运行黑客指定的第三方程序。

另一类问题则是由隐式类型转换带来的问题。这一类问题的翘楚就是 PHP 和 JavaScript。这两门语言在运算所需类型不满足的情况下，比起抛出错误，更愿意尝试一系列自动的类型转换来满足需求，而这一过程会带来一系列的问题。

比如在 PHP 中著名的 Type Juggling 漏洞，使用 == 符号比较两个完全不同的字符串：

```
if ('0e1234' == '0e5678') {
  echo('wat');
}
```

程序会返回 `wat`，纵使字符串 `0e1234` 和 `0e5678` 完全不相同，但是 PHP 发现了这些字符串都是 `0e` 开始的，于是会认为这 **可能** 是 科学记数法 的数字。而无论 0 的多少次方都是 0，所以两者都是 0，因此认为这两个字符串相等。这显然不是预期行为。这一方法常会在一些特殊情况引发更严重的问题，比如检查用户登录密码时，一般数据库存储哈希运算后的 16 进制密码以防止明文泄漏。而 16 进制数有 $\frac{1}{256}$ 的概率以 `0e` 开头，同时以 `00e`、`000e` 开头的同样也会被以同样的方式处理，因此给定 `N` 位长的字符串，PHP 会有 $\sum_{i=2}^{n} \frac{1}{16^n}$ 的概率认为其等于 0。

这两类由类型带来的问题是高级语言需要尽力避免的。

在运行过程中不可能出现类型错误问题的，是强类型系统；而如果这门语言在运行过程中会因为类型错误而产生不可知的任意行为的，那么就是弱类型系统。

在编译期（运行程序前）检查类型错误的是静态类型系统，在运行过程中拒绝类型错误的程序继续运行的，则是动态类型系统。

虽然 Ruby 考虑在未来的 3.0 版本中引入静态类型检查系统，目前的 Ruby 2.7 仍是运行时的动态类型检查。但是 Ruby 是强类型语言，会在运行通过抛出类型错误，来避免错误的类型转换带来潜在的风险。

如何理解 Ruby 变量与常量

本章节涉及较为进阶内容，建议第一次接触编程或缺乏编程经验的开发者暂时跳过这一章节。

让人出乎意料的是：Ruby 并没有真正意义的常量

这句话一出口，必然有人不服。「Ruby 里全大写的关键字，不就是常量吗？」此言差矣。虽然 Ruby 中的全大写的关键字会被作为常量来识别，但是 Ruby 中的常量本质上还是变量，因为你依然可以给常量任意赋值，只是会触发警告。

```
CONST_A = 'foo'
CONST_A = 'bar' # => warning: already initialized constant CONST_A
puts CONST_A # 'bar'
```

另一个常常被人混淆的概念则是 Ruby 中的 `Object#freeze` 方法。「把变量冻结后我就不能编辑了，这难道不是常量吗？」

```
a = 'foo'.freeze
a << 'bar' # FrozenError (can't modify frozen String: "a")
```

这同样不是常量，而是另一个名词「不变量（immutable）」。我们可以看下面的例子，只要不是编辑对象本身，我依然可以给冻结后的对象赋予一个完全的新值。

```
a = 'foo'.freeze
a = 'bar'
puts a # 'bar'
```

虽然可变量（mutable）、不变量（immutable）听起来和变量（variable）以及常量（const）听起来非常像，但它们是完全不一样的东西。我们要把数据的存储拆成两个部分来看，代号和内容。变量、常量说的是代号所指的对象的可变还是不可变；而可变量、不变量指的是指向的那个对象里面内容物的可变还是不可变。

我们举个例子，现在有一座房子，地址是 A，A 就是变量名。A 房子里住着小明。如果小明可以把这 A 房子卖给小王，那么这个 A 房子就是个「变量」；如果小明不被允许把 A 房子卖给其他人，那么 A 就是一个「常量」。A 房子里的小明体重 100 斤，今天小明吃成了 200 斤，那么他就是个「可变量」；如果小明住进房子里，所有性状不可能变化了，那么他就是个「不变量」。至于今天 A 房子里住着一个 100 斤的小明，我明天把 A 房子转给另一个 200 斤的小王住，那就不管「可变/不变」什么事，而是「变量/常量」的事情了。

简而言之，「可变/不变」是数据结构本身的特性，而「变量/常量」是指向数据的代号的特性。

讲到这里结论就已经很明显了：Ruby 没有常量，全大写的常量其实本质上还是变量只是会在改变时抛出 warning，而 freeze 是将可变量转换成不可变量。

函数基础

Ruby 中的函数 (function) 的本质其实是方法 (method)。关于这两者的差异，我们会在《第四周：面向对象再入门》中再详述。我们在这一章节中还是作为函数来处理。

什么是函数?

理解什么是程序中的函数，我们先理解什么是数学中的函数。在小学和初中中学习的函数通常是比较狭隘的，往

往把函数和表达式画上等号。但站在集合论的角度，我们可以更好理解函数的本质。我们把定义域和值域看成两个集合，函数就是定义域和值域的对应关系。

如果我们用比较粗糙的观点来看，函数 f 就是一个黑盒 (black box)。当你扔进去一个定义域里的 x，函数就会返回一个经过函数处理的结果 y，即 $y = f(x)$。程序中的函数也是这样一个东西，但是和数学中的函数也有一些细微差异。

目的上来说，程序中的函数和数学上的函数是一致的。在数学中，函数的引入极大简化了表达。重复的运算可以由函数或函数的组合来表达，同时我们也可以对函数本身进行进一步的研究（比如导数）。程序中也是同理。程序的开发是为了解决人类的重复性工作，其代码过程中也必然会有大量重复的，需要复用的地方。而函数就为这种场景提供了绝佳的解决方案。

但程序中的函数和数学上的函数还存在一定差异。对于一个数学函数，对于一个给定的 x，无论传入多少次，其结果都是不变的。但这一行为在大多数编程语言中通常是不被保证的，即这一函数函数是不是纯函数 (pure function)。我们会在之后的例子中进一步讨论这一问题。

~~当我们把需求、比萨和咖啡扔给程序员，程序员就会输出程序。~~

~~从这个角度上来看，程序员不也是一个函数吗？~~

函数定义

函数的三大要素：定义域、值域和映射方法。

对应到程序中就是参数、返回值和函数代码片段。我们也从这三块来理解 Ruby 中的函数定义。

```ruby
def succ(x) # <- 函数的名称与参数
  x + 1 # <- 函数的返回值
end # <- 函数代码结束

succ(1) # => 2
```

Ruby 的函数以 `def` 开始，`def` 是 define 的缩写，即要在此定义函数。

`def` 后接一个空格，然后是函数的名称，这里例子中是 `succ`，即之后要用 `succ` 这个名字来调用这一函数。

函数名后紧跟一对括号，括号内是参数列表。函数的定义域在程序中就是以「参数 (arguments)」来实现的。如果没有参数，可以省略这一对括号。例如：

```ruby
def hello
  puts 'Hello'
end
```

括号内的参数是一些变量名，外面传入的参数就会在这里以这些变量被传入函数内。如果需要传入多个参数，则以逗号 , 分割。

```ruby
def add(a, b)
  a + b
end

add(1, 2) # => 3
```

函数的映射方法是通过函数代码来实现的。参数列表后，end 关键字前即为函数代码，函数代码可以利用函数中的变量进行一些变换从而实现函数的功能。而最后一步，则是将函数的计算结果传回给函数的调用。函数的值域是通过「返回值 (return value)」来实现的。

在 Ruby 中函数有两种方法定义返回值。

- 使用 return 来返回值
- 默认将最后一次运算结果返回

在下面这个例子中：

```ruby
def add(a, b)
  a + b
end

add(1, 2) # => 3
```

函数的第一行也是最后一行代码是 a + b，Ruby 会自动将这最后的计算结果作为返回值。

而类似于其它编程语言，Ruby 也支持使用 return 关键字来返回值。同时 return 也意味者程序的提前运行，这在我们之后讲到流程控制时会非常有用。

```ruby
def add(a, b)
  return a + b
  puts 'wat?' # 这一行不会被运行
end
```

函数调用

调用自己定义的函数如同我们之前调用过的所有系统内建的函数一样，直接函数名加上参数即可。

```ruby
def add(a, b)
  return a + b
end

add(1, 1) # => 2
add 1, 1  # => 2
```

在许多变成语言中，参数必须被 () 包裹住。在 Ruby 中，如果嵌套关系明确，那么 () 也可以省略。这也是为什么我们之前打印内容到屏幕时可以写成 `puts 'Hello World'`，这其实和 `puts('Hello World')` 等价。

豆知识：`void` v.s. `nil`

Ruby 和其它语言还有一个在函数返回值上的差异。在 Ruby 中任何方法都有「返回值」，但返回值可能为「空」。这句话听起来非常拗口，我们来简单比较一下。如果我们在 C++ 语言中定义一个函数如下：

```
void void_function() {
  return;
}

int main() {
  auto x = void_function(); // illegal, 非法代码。
  return 0;
}
```

这段代码是非法的，因为 `void_function()` 没有返回值，不能让其结果赋值给 `x`。

但是如果在 Ruby 中定义一个空函数，

```
def nil_func
  return
end

x = nil_func
p x # => nil
```

但是我们保持语义改用 Ruby 来写。`nil_func` 函数确实是有返回值的，只不过这个返回值是 `nil` 而已。Ruby 中的 **任何函数都有返回值**。

小练习

1. 创建一个函数 `pow`，接受两个参数 `x` 和 `y`，计算 x^y 的结果并返回。
2. 创建一个函数 `succ`，接受一个参数 `x` 返回 `x + 1`。定义另一个函数 `succ2`，在不使用 `+` 运算符的前提下，返回 `x + 2`。

豆知识：定义好的函数可以取消吗？

虽然这样的需求很少会发生，但 Ruby 还真的支持取消定义好的函数的特性。可以试一试下面的代码：

```
def foo
  'bar'
end

foo # => 'bar'

undef foo

foo # => NameError
```

条件与循环

条件判断

`if` `case…when…`

循环

`while`

范围

`for i in (N..M)`

`(N..M).each`

迭代器

`.each`

`N.times do`

语法糖

Ruby 喜欢语法糖（syntactic sugar）。

在计算机科学中，语法糖是一类语法，这类语法对于语言的功能不产生任何影响。但是提高代码的可读性，从而更方便程序员使用。Ruby 提供了一些语法，使得其阅读起来更接近于自然语言。比如下面的代码：

```
stop = false

if not stop
  puts 'Go!'
end
```

在自然语言中，我们很少会使用 if not 这样的语法，而是会使用 `unless`。Ruby 中也提供了 `unless` 关键词，作为 if not 的替代。于是你可以把代码写成下面这样：

```
stop = false

unless stop
  puts 'Go!'
end
```

在 Ruby 中，如果条件判断内只有一行，你可以把条件判断放在你要执行的代码后面，从而写成：

```
stop = false

puts 'Go!' unless stop
```

这样读起来确实变得更接近自然语言了。类似地，对于 `while not` 循环，你可以用 `until` 关键词来替代：

```
a = 5

until a == 0
  a = a - 1
end
```

同样，你也可以简化成一行：

```
a = 5

a = a - 1 until a == 0
```

对于四则运算，一个常见的需求就是累加累减，也就是说，对于某一变量计算的结果会存回变量本身。对于这一点 Ruby 也提供了语法糖，即可以使用

```
a += b
a -= b
a *= b
a /= b
```

来替代

```
a = a + b
a = a - b
a = a * b
a = a / b
```

于是我们可以把上面的代码简化成：

```
a = 5

a -= 1 until a == 0
```

特别注意

如果你是 C 语言或者 C++ 语言的使用者，那么你必然还熟悉另一种累加累减运算符 `a++` `++a` `a--` `--a`。但是在 Ruby 中没有对应语法。

对于死循环（infinite loop），你还可以使用 `loop` 来替代 `while true`：

```ruby
loop do
  puts 'Hello World'
end
```

可计算性与语法设计

本章节涉及较为进阶内容，建议第一次接触编程或缺乏编程经验的开发者暂时跳过这一章节。

事实上，有了变量赋值和条件、循环（或者跳转），如果不考虑交互和通讯，就已经可以解决了所有计算机可以计算的问题。关于这一方面的研究被称为可计算性理论（Computability Theory）。20 世纪上半叶，对于可计算性的公式化表达主要有三：

- 阿隆佐·邱奇提出的 Lambda 演算。
- 阿兰·图灵提出的通用图灵机。
- 邱奇、克莱尼和罗斯塞尔提出的递归算法。

现在已经证明，这三种模型的可计算性上是等价的。在今天的高级语言中，语言通常会提供高于这三者的抽象。以 Ruby 为例，Ruby 同时提供偏向于图灵机设计理念的变量、分支系统，提供接近 Lambda 演算概念的函数式编程泛型，也支持递归函数定义。但即使如此，Ruby 编程语言的可计算性并不比只提供一种模型强。今天的计算机和 20 年前的计算机，也没有可计算性上的差异，只是存储更大、运行速度更快而已。这如同标准大气压下三杯 100℃ 的开水，把三杯水倒在一起，温度还是 100℃。但是不同语言确实会有不同的抽象，语言提供丰富的抽象的目的主要有两点：

- 更适合现代计算机系统的设计，从而提高性能。
- 更符合人类直觉，从而增加代码的可读性。

为性能服务的语法设计

比如我们给定一个数组和范围，让程序计算其平方，在 Ruby 中可以写成：

```ruby
def square(arr, a, b)
  (a...b).each do |i|
    arr[i] = arr[i] ** 2
  end
end
```

这一程序可以同时对整数和小数作用。不同于 Ruby，像 C++ 之类的语言就要求必须要在编译器定义数据的类型。比如下面的程序只能对浮点数有效：

```cpp
void square(float *A, int start, int end) {
  for (int i = start; i < end; ++i) {
    A[i] = A[i] * A[i];
  }
}
```

但如果我们检查 C++ 编译器（此处使用 LLVM 9.0 (x86-64)，编译参数为 -O2）我们可以看到如下的交给 CPU 执行的汇编代码：

```
square(float*, int, int):                             # @square(float*, int, int)
        cmp     esi, edx
        jge     .LBB0_11
        movsxd  rax, esi
        movsxd  r11, edx
        mov     r10, r11
        sub     r10, rax
        cmp     r10, 7
        jbe     .LBB0_10
        mov     r8, r10
        and     r8, -8
        lea     rcx, [r8 - 8]
        mov     rdx, rcx
        shr     rdx, 3
        add     rdx, 1
        mov     r9d, edx
        and     r9d, 1
        test    rcx, rcx
        je      .LBB0_3
        sub     rdx, r9
```

```
        lea     rcx, [rdi + 4*rax]
        add     rcx, 48
        xor     esi, esi
.LBB0_5:                                # =>This Inner Loop Header: Depth=1
        movups  xmm0, xmmword ptr [rcx + 4*rsi - 48]
        movups  xmm1, xmmword ptr [rcx + 4*rsi - 32]
        movups  xmm2, xmmword ptr [rcx + 4*rsi - 16]
        movups  xmm3, xmmword ptr [rcx + 4*rsi]
        mulps   xmm0, xmm0
        mulps   xmm1, xmm1
        movups  xmmword ptr [rcx + 4*rsi - 48], xmm0
        movups  xmmword ptr [rcx + 4*rsi - 32], xmm1
        mulps   xmm2, xmm2
        mulps   xmm3, xmm3
        movups  xmmword ptr [rcx + 4*rsi - 16], xmm2
        movups  xmmword ptr [rcx + 4*rsi], xmm3
        add     rsi, 16
        add     rdx, -2
        jne     .LBB0_5
        test    r9, r9
        je      .LBB0_8
.LBB0_7:
        add     rsi, rax
        movups  xmm0, xmmword ptr [rdi + 4*rsi]
        movups  xmm1, xmmword ptr [rdi + 4*rsi + 16]
        mulps   xmm0, xmm0
        mulps   xmm1, xmm1
        movups  xmmword ptr [rdi + 4*rsi], xmm0
        movups  xmmword ptr [rdi + 4*rsi + 16], xmm1
.LBB0_8:
        cmp     r10, r8
        je      .LBB0_11
        add     rax, r8
.LBB0_10:                               # =>This Inner Loop Header: Depth=1
        movss   xmm0, dword ptr [rdi + 4*rax] # xmm0 = mem[0],zero,zero,zero
        mulss   xmm0, xmm0
        movss   dword ptr [rdi + 4*rax], xmm0
        add     rax, 1
        cmp     r11, rax
        jne     .LBB0_10
.LBB0_11:
```

```
        ret
.LBB0_3:
        xor     esi, esi
        test    r9, r9
        jne     .LBB0_7
        jmp     .LBB0_8
```

我们会发现，编译结果和我们的语义有非常大的差异。这是因为编译器识别出了这是一个循环，同时发现了循环内的数据没有依赖性。最后由于数据结构的类型已经被提前定义，数据的宽度可以精确确认。所以编译器就能精确使用 CPU 的 SIMD 指令集来进行运算。于是它就会把多个浮点数数据同时计算，并且会引入一些代码来处理边界情况，从而极大提高运行的性能。

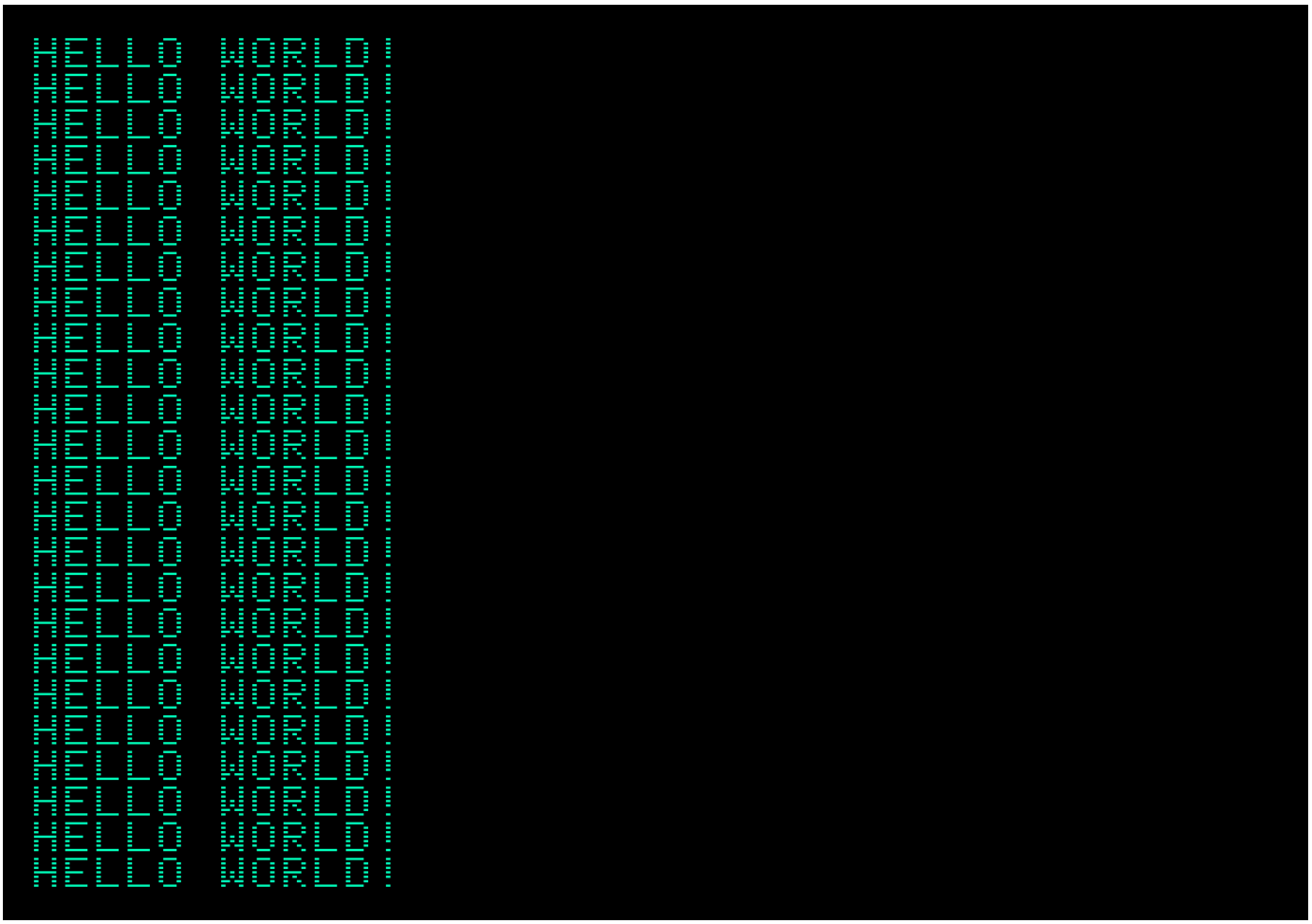
这对于 Ruby 灵活的动态类型系统是无法做到的。这就是所谓为性能服务的语法设计。

为可读性服务的语法设计

在早年流行的高级语言 BASIC 中，一个非常常用的流程控制指令是 `goto`，一个常见的无限循环写法如下（以 1978 年 Apple II Plus 上的 Applesoft BASIC 为例）：

```
]10 PRINT "HELLO WORLD!"
]20 GOTO 10
]LIST
10   PRINT "HELLO WORLD!"
20   GOTO 10
]RUN 10
```

运行结果如下：



仔细思考会发现，goto 的语义不但可以替代所有循环（while for），还能替代循环内部的流程控制（break continue）。goto 在现代的 CPU 中都有直接的指令对应实现。在 x86 和 x86-64 系统上就是最直接的 JMP 指令，虽然还需要处理一些栈上内存相关的事物，但也可以简单由数个指令执行完毕，性能上是绝对没有问题的。

但我们现在很少使用 goto 语法的一大原因是因为，goto 的功能过于强大，虽然符合机器的执行原理，但是却不符合人的思考直觉。代码不仅仅需要被计算机执行，还需要被人类编写、讨论和维护。可读性的重要性随着工程的复杂度的提高会显得越来越重要。而 while for break continue 等一系列限制更严格的条件控制方法的引入更符合了人类的逻辑直觉，从而提高的代码的可读性。

第二周：数学运算再入门

基础数学运算

整数以外的运算

在第一周的《变量与基本算数》中我们已经学习了整数的运算，在这一章节，我们会学习其它的 Ruby 数学运算。

浮点数运算

Ruby 中的小数默认是根据 IEEE 754 规范定义的双精度浮点数。

```
0.1 + 0.2 # => 0.30000000000000004
```

高精度小数运算

```
require 'bigdecimal'

BigDecimal('0.1') + BigDecimal('0.2') # => 0.3e0
```

布尔（逻辑）运算

比起整数和浮点数运算，布尔运算可能是最简单的，但却是大家日常最不熟悉的。

布尔运算就是关于「是」「非」的计算，在 Ruby 中这两者的关键字是 `true` 和 `false`。布尔运算的三个最基本运算是「与（and）」、「或（or）」、「非（not）」。

- 与运算（`and`）就是当算符两侧皆为 `true` 那么结果还是 `true`，否则为 `false`。
- 或运算（`or`）当算符两侧只要有一个操作数为 `true` 那么结果就为 `true`。
- 非运算（`not`）只接受一个操作数，将 `true` 变成 `false`，将 `false` 变成 `true`。

用 Ruby 代码写如下，大家可以打开 irb 试一下：

```ruby
true and true # => true
true and false # => false
false and true # => false
false and false # => false

true or true # => true
true or false # => true
false or true # => true
false or false # => false

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

Ruby 中的 `and` `or` `not` 也可以由符号 `&&` `||` `!` 来替代。

```ruby
true && true # => true
true && false # => false
false && true # => false
false && false # => false

true || true # => true
true || false # => true
false || true # => true
false || false # => false

!true # => false
!false # => true
```

特别注意

是 `&&` 和 `||` 而不是 `&` 和 `|`。`&` 和 `|` 这两个是二进制运算符，和布尔运算有一定差异，我们会在之后具体介绍这两个运算。

逻辑运算也可以使用括号进行优先级的组合，如果没有括号，默认的计算优先级是 `not` 优先于 `and` 优先于 `or`。一个好记的优先级口诀是「not at all. (not and or)」。

尝试猜测下面这句代码的结果，打开 irb 试一试，检验自己的理解是否正确。

```
true && (true and not false) or !false || false
```

Ruby 标准数学库