

如何掌握所有的程序语言

对的，我这里要讲的不是如何掌握一种程序语言，而是所有的……

很多编程初学者至今还在给我写信请教，问我该学习什么程序语言，怎么学习。由于我知道如何掌握“所有”的程序语言，总是感觉这种该学“一种”什么语言的问题比较低级，所以一直没来得及回复他们:P 可是逐渐的，我发现原来不只是小白们有这个问题，就连美国大公司的很多资深工程师，其实也没搞明白。

今天我有动力了，想来统一回答一下这个搁置已久的“初级问题”。类似的话题貌似曾经写过，然而现在我想把它重新写一遍。因为在跟很多人交流之后，我对自己头脑中的（未转化为语言的）想法，有了更精准的表达。

如果你存在以下的种种困惑，那么这篇文章也许会对你有所帮助：

1. 你是编程初学者，不知道该选择什么程序语言来入门。
2. 你是资深的程序员或者团队领导，对新出现的种种语言感到困惑，不知道该“投资”哪种语言。
3. 你的团队为使用哪种程序语言争论不休，发生各种宗教斗争。
4. 你追逐潮流采用了某种时髦的语言，结果两个月之后发现深陷泥潭，痛苦不堪……

虽然我已经不再过问这些世事，然而无可置疑的现实是，程序语言仍然是很重要的话题，这个情况短时间内不会改变。程序员的岗位往往会要求熟悉某些语言，甚至某些奇葩的公司要求你“深入理解 OOP 或者 FP 设计模式”。对于在职的程序员，程序语言至今仍然是可以争得面红耳赤的宗教话题。它的宗教性之强，以至于我在批评和调侃某些语言（比如 Go 语言）的时候，有些人会本能地以为我是另外一种语言（比如 Java）的粉丝。

显然我不可能是任何一种语言的粉丝，我甚至不是 Yin 语言的粉丝;) 对于任何从没见过的语言，我都是直接拿起来就用，而不需要经过学习的过程。看了这篇文章，也许你会明白我为什么可以达到这个效果。理解了这里面的东西，每个程序员都应该可以做到这一点。嗯，但愿吧。

重视语言特性，而不是语言

很多人在乎自己或者别人是否“会”某种语言，对“发明”了某种语言的人倍加崇拜，为各种语言的孰优孰劣争得面红耳赤。这些问题对于我来说都是不存在的。虽然我写文章批评过不少语言的缺陷，在实际工作中我却很少跟人争论这些。如果有其它人在我身边争论，我甚至会戴上耳机，都懒得听他们说什么;) 为什么呢? 我发现归根结底的原因，是因为我重视的是“语言特性”，而不是整个的“语言”。我能用任何语言写出不错的代码，就算再糟糕的语言也差不了多少。

任何一种“语言”，都是各种“语言特性”的组合。打个比方吧，一个程序语言就像一台电脑。它的牌子可能叫“联想”，或者“IBM”，或者“Dell”，或者“苹果”。那么，你可以说苹果一定比 IBM 好吗? 你不能。你得看看它里面装的是什么型号的处理器，有多少个核，主频多少，有多少 L1 cache, L2 cache……，有多少内存和硬盘，显示器分辨率有多大，显卡是什么 GPU，网卡速度，等等各种“配置”。有时候你还得看各个组件之间的兼容性。

这些配置对应到程序语言里面，就是所谓“语言特性”。举一些语言特性的例子：

- 变量定义
- 算术运算
- for 循环语句，while 循环语句
- 函数定义，函数调用
- 递归
- 静态类型系统
- 类型推导
- lambda 函数
- 面向对象
- 垃圾回收
- 指针算术
- goto 语句

这些语言特性，就像你在选择一台电脑的时候，看它里面是什么配置。选电脑的时候，没有人会说 Dell 一定是最好的，他们只会说这个型号里面装的是 Intel 的 i7 处理器，这个比 i5 的好，DDR3 的内存比 DDR2 的快这么多，SSD 比磁盘快很多，ATI 的显卡是垃圾…… 如此等等。

程序语言也是一样的道理。对于初学者来说，其实没必要纠结到底要先学哪一种语言，再学哪一种。曾经有人给我发信问这种问题，纠结了好几个星期，结果一个语言都还没开始学。有这纠结的时间，其实都可以把他纠结过的语言全部掌握了。

初学者往往不理解，每一种语言里面必然有一套“通用”的特性。比如变量，函数，整数和浮点数运算，等等。这些是每个通用程序语言里面都必须有的，一个都不能少。你只要通过“某种语言”学会了这些特性，掌握这些特性的根本概念，就能随时把这些知识应用到任何其它语言。你为此投入的时间基本不会浪费。所以初学者纠结要“先学哪种语言”，这种时间花的很不值得，还不如随便挑一个语言，跳进去。

如果你不能用一种语言里面的基本特性写出好的代码，那你换成另外一种语言也无济于事。你会写出一样差的代码。我经常看到有些人 Java 代码写得相当乱，相当糟糕，却骂 Java 不好，雄心勃勃要换用 Go 语言。这些人没有明白，是否能写出好的代码在于人，而不在于语言。如果你的心中没有清晰简单的思维模型，你用任何语言表述出来都是一堆乱麻。如果你 Java 代码写得很糟糕，那么你写 Go 语言代码也会一样糟糕，甚至更差。

很多初学者不了解，一个高明的程序员如果开始用一种新的程序语言，他往往不是去看这个语言的大部头手册或者书籍，而是先有一个需要解决的问题。手头有了问题，他可以用两分钟浏览一下这语言的手册，看看这语言大概长什么样。然后，他直接拿起一段例子代码来开始修改捣鼓，想法把这代码改成自己正想解决的问题。在这个简短的过程中，他很快的掌握了这个语言，并用它表达出心里的想法。

在这个过程中，随着需求的出现，他可能会问这样的问题：

- 这个语言的“变量定义”是什么语法，需要“声明类型”吗，还是可以用“类型推导”？
- 它的“类型”是什么语法？是否支持“泛型”？泛型的“variance”如何表达？
- 这个语言的“函数”是什么语法，“函数调用”是什么语法，可否使用“缺省参数”？
-

注意到了吗？上面每一个引号里面的内容，都是一种语言特性（或者叫概念）。这些概念可以存在于任何的语言里面，虽然语法可能不一样，它们的本质都是一样的。比如，有些语言的参数类型写在变量前面，有些写在后面，有些中间隔了一个冒号，有些没有。

这些实际问题都是随着写实际的代码，解决手头的问题，自然而然带出来的，而不是一开头就抱着语言手册看得仔仔细细。因为掌握了语言特性的人都知道，自己需要的特性，在任何语言里面一定有对应的表达方式。如果没有直接的方式表达，那么一定有某种“绕过方式”。如果有直接的表达方式，那么它只是语法稍微有所不同而已。所以，他是带着问题找特性，就像查字典一样，而不是被淹没于大部头的手册里面，昏昏欲睡一个月才开始写代码。

掌握了通用的语言特性，剩下的就只剩某些语言“特有”的特性了。研究语言的人都知道，要设计出新的、好的、无害的特性，是非常困难的。所以一般说来，一种好的语言，它所特有的新特性，终究不会超过一两种。如果有个语言号称自己有超过 5 种新特性，那你就得小心了，因为它们带来的和可能不是优势，而是灾难！

同样的道理，最好的语言研究者，往往不是某种语言的设计者，而是某种关键语言特性的设计者（或者支持者）。举个例子，著名的计算机科学家 Dijkstra 就是“递归”的强烈支持者。现在的语言里面都有递归，然而你可能不知道，早期的程序语言是不支持递归的。直到 Dijkstra 强烈要求 Algol 60 委员会加入对递归的支持，这个局面才改变了。Tony Hoare 也是语言特性设计者。他设计了几个重要的语言特性，却没有设计过任何语言。另外大家不要忘了，有个语言专家叫王垠，他是早期 union type 的支持者和实现者，也是 checked exception 特性的支持者，他在自己的[博文](#)里指出了 checked exception 和 union type 之间的关系：P

很多人盲目的崇拜语言设计者，只要听到有人设计（或者美其民曰“发明”）了一个语言，就热血沸腾，佩服的五体投地。他们却没有理解，其实所有的程序语言，不过是像 Dell，联想一样的“组装机”。语言特性的设计者，才是像 Intel，AMD，ARM，Qualcomm 那样核心技术的创造者。

合理的入门语言

所以初学者要想事半功倍，就应该从一种“合理”的，没有明显严重问题的语言出发，掌握最关键的 language 特性，然后由此把这些概念应用到其它语言。哪些是合理的入门语言呢？我个人觉得这些语言都可以用来入门：

- Scheme
- C
- Java
- Python
- JavaScript

那么相比之下，我不推荐用哪些语言入门呢？

- Shell
- PowerShell
- AWK
- Perl
- PHP

- Basic
- Go
- Rust

总的说来，你不应该使用所谓“[脚本语言](#)”作为入门语言，特别是那些源于早期 Unix 系统的脚本语言工具。PowerShell 虽然比 Unix 的 Shell 有所进步，然而它仍然没有摆脱脚本语言的根本问题——他们的设计者不知道他们自己在干什么:P

采用脚本语言学编程，一个很严重的问题就是使得学习者抓不住关键。脚本语言往往把一些系统工具性质的东西（比如正则表达式，Web 概念）加入到语法里面，导致初学者为它们浪费太多时间，却没有理解编程最关键的 concept：变量，函数，递归，类型……

不推荐 Go 语言的原因类似，虽然 Go 语言不算脚本语言，然而他的设计者显然不明白自己在干什么。所以使用 Go 语言来学编程，你不能专注于最关键，最好的语言特性。关于 Go 语言的各种毛病，你可以参考这篇[文章](#)。

同样的，我不觉得 Rust 适合作为入门语言。Rust 花了太多精力来夸耀它的“新特性”，而这些新特性不但不是最关键的部分，而且很多是有问题的。初学者过早的关注这些特性，不仅学不会最关键的编程思想，而且可能误入歧途。关于 Rust 的一些问题，你可以参考这篇[文章](#)。

掌握关键语言特性，忽略次要特性

为了达到我之前提到的融会贯通，一通百通的效果，初学者应该专注于语言里面最关键的特性，而不是被次要的特性分心。

举个夸张点的例子。我发现很多编程培训班和野鸡大学的编程入门课，往往一来就教学生如何使用 printf 打印“Hello World！”，进而要他们记忆 printf 的各种“格式字符”的意义，要他们实现各种复杂格式的打印输出，甚至要求打印到文本文件里，然后再读出来……

可是殊不知，这种输出输入操作其实根本不算是语言的一部分，而且对于掌握编程的核心概念来说，都是次要的。有些人的 Java 课程进行了好几个星期，居然还在布置各种 printf 的作业。学生写出几百行的 printf，却不理解变量和函数是什么，甚至连算术语句和循环语句都不知道怎么用！这就是为什么很多初学者感觉编程很难，我连 %d, %f, %.2f 的含义都记不住，还怎么学编程！

然而这些野鸡大学的“教授”头衔是如此的洗脑，以至于被他们教过的学生（比如我女朋友）到我这里请教，居然骂我净教一些没用的东西，学了连 printf 的作业都没法完成:P 你别跟我讲 for 循环，函数什么的了……可不可以等几个月，等我背熟了 printf 的用法再学那些啊？

所以你就发现一旦被差劲的老师教过，这个程序员基本就毁了。就算遇到好的老师，他们也很难纠正过来。

当然这是一个夸张的例子，因为 printf 根本不算是语言特性，但这个例子从同样的角度说明了次要肤浅的语言特性带来的问题。

这里举一些次要语言特性的例子：

- C 语言的语句块，如果里面只有一条语句，可以不打花括号。
- Go 语言的函数参数类型如果一样可以合并在一起写，比如 `func foo(s string, x, y, z int, c bool) { ... }`
- Perl 把正则表达式作为语言的一种特殊语法
- JavaScript 语句可以在某些时候省略句尾的分号
- Haskell 和 ML 等语言的 [currying](#)

自己动手实现语言特性

在基本学会了各种语言特性，能用它们来写代码之后，下一步的进阶就是去实现它们。只有实现了各种语言特性，你才能完全地拥有它们，成为它们的主人。否则你就只是它们的使用者，你会被语言的设计者牵着鼻子走。

有个大师说得好，完全理解一种语言最好的方法就是自己动手实现它，也就是自己写一个解释器来实现它的语义。但我觉得这句话应该稍微修改一下：完全理解一种“语言特性”最好的方法就是自己亲自实现它。

注意我在这里把“语言”改为了“语言特性”。你并不需要实现整个语言来达到这个目的，因为我们最终使用的是语言特性。只要你自己实现了一种语言特性，你就能理解这个特性在任何语言里的实现方式和用法。

举个例子，学习 SICP 的时候，大家都会亲自用 Scheme 实现一个面向对象系统。用 Scheme 实现的面向对象系统，跟 Java, C++, Python 之类的语言语法相去甚远，然而它却能帮助你理解任何这些 OOP 语言里面的“面向对象”这一概念，它甚至能帮助你理解各种面向对象实现的差异。

这种效果是你直接学习 OOP 语言得不到的，因为在学习 Java, C++, Python 之类语言的时候，你只是一个用户，而用 Scheme 自己动手实现了 OO 系统之后，你成为了一个创造者。

类似的特性还包括类型推导，类型检查，惰性求值，如此等等。我实现过几乎所有的语言特性，所以任何语言在我的面前，都是可以被任意拆卸组装的玩具，而不再是凌驾于我之上的神圣。

总结

写了这么多，重要的话重复三遍：语言特性，语言特性，语言特性，语言特性！不管是初学者还是资深程序员，应该专注于语言特性，而不是纠结于整个的“语言品牌”。只有这样才能达到融会贯通，拿起任何语言几乎立即就会用，并且写出高质量的代码。

（如果你觉得这篇文章有所帮助，可以[付款](#)购买，价格随意。）