异步图书
www.epubit.com

# 自制编程语言

## 基于C语言

郑钢 著

手把手地教读者从零去实现一门语言，从原理到实践事无巨细

每一步都有实际的代码和详尽的原理说明，读者可以很轻松地掌握各个实现细节

实现脚本语言重要的垃圾回收（GC）、虚拟机运行时和线程等技术都在本书一一呈现

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

# 自制编程语言

## 基于C语言

郑钢 著



人民邮电出版社
北京

**图书在版编目（CIP）数据**

自制编程语言 / 郑钢著. -- 北京 : 人民邮电出版社, 2018.9
ISBN 978-7-115-48737-7

Ⅰ. ①自… Ⅱ. ①郑… Ⅲ. ①C语言－程序设计
Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2018)第137473号

## 内 容 提 要

本书是一本专门介绍自制编程语言的图书，书中深入浅出地讲述了如何开发一门编程语言，以及运行这门编程语言的虚拟机。本书主要内容包括：脚本语言的功能、词法分析器、类、对象、原生方法、自上而下算符优先、语法分析、语义分析、虚拟机、内建类、垃圾回收、命令行及调试等技术。

本书适合程序员阅读，也适合对编程语言原理感兴趣的计算机从业人员学习。

◆ 著　　郑　钢
责任编辑　张　涛
责任印制　马振武

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
固安县铭成印刷有限公司印刷

◆ 开本：787×1092　1/16
印张：28
字数：743 千字　　2018 年 9 月第 1 版
印数：1－2 400 册　　2018 年 9 月河北第 1 次印刷

定价：89.00 元

**读者服务热线：(010)81055410　印装质量热线：(010)81055316**
**反盗版热线：(010)81055315**
**广告经营许可证：京东工商广登字 20170147 号**

# 目　录

# 第0章　一些可能令人迷惑的问题

本章涉及一些编译原理基础，我担心没学过编译原理的读者会觉得吃力，因此顺带介绍了编译原理的基础知识。第 1 章以后的内容就不需要这些基础了，不会编译原理也无法阻止你成功写出一门脚本语言。因为原理太抽象了，而且为了严谨，理论总是把简单的描述成复杂的。在实践中你会发现，编译器的实现比理解编译器原理容易，你会发现——原来晦涩难懂的概念其实就是这么简单，以至于你是通过实践才懂得了编译原理。毕竟纸上得来终觉浅，绝知此事要躬行。

总之，如果有一些内容不感兴趣或者我解释得不够清楚你也不必着急，这并不影响后面章节的阅读。

## 0.0 成功的基石不是坚持，而是“不放弃”

人们常说，坚持是成功的“前提”。我说，既然只是前提，这说明坚持也未必会成功。要想成功，人们需要的是成功的“基石”，而不是“前提”，这个基石就是 3 个字：不放弃。

大部分读者都觉得开发一门编程语言是很难的事，甚至想都不敢想，我担心你也有这个想法，所以特意用这种方式先和你说说心里话：这本书你买都买了，多少发挥点价值才对得起买书的钱，谁的钱也不是白来的。

首先，我并不会为了鼓励大家而大言不惭地说开发语言“其实不难”“很容易”之类的话，相反，这个方向确实很难，而且就应该很难，我想这也正是吸引你的地方，没有难度哪来的价值，“其实不难、很容易”之类的话是对大家上进心的不尊重。

其次，只有在“我也认为很难”的前提下才能保证大部分的朋友能看懂本书。你看，在普通人眼里从 A 到 D，需要有 B 和 C 的推理过程，一个步骤都不能少，在天才眼里，A 到 D 是理所应当的事，不需要解释得太清楚，天才认为 B 和 C 都是废话，明摆着的事不需要解释。而我不是天才，所以我会把 B 和 C 解释清楚。

回到开头的话，为什么说成功的基石不是“坚持”而是“不放弃”呢？这两个词有啥区别？也许有读者说，不放弃就是做着喜欢的事，让自己爱上学习技术。个人觉得这有点不对了，我觉得我更喜欢吃喝玩乐，因为那是生物的本能，选择技术的原因只是我没那么讨厌它，它是我从众多讨厌的事物中选择的最不讨厌的东西。

放弃是为了减少痛苦，坚持是带着痛苦继续前行。“坚持”是个痛苦的词，但凡靠坚持来做的事情必然建立在痛苦之上，而痛苦就会使人产生放弃的念头，这是生物的本能。用“坚持”来“鼓励”自己硬着头皮干，其实已经输了一半，自己认为痛苦的事很难干下去，干不下去的原因是遇到困难时头脑里有“放弃”的念头，如果把这个念头去掉，那么，只要活着，成功无非是时间长短的问题。这个念头其实就是心理预期，“提前”做好心理预期很重要。

总之，不要给自己“可以放弃”的念头，不要让“可以放弃”成为一种选项，把这个选项去掉，那么，只剩下成功。

## 0.1 你懂编程语言的“心”吗

先来猜猜这是什么？

它是一种人人必不可少，拥有多种颜色、多种外形的物品。

它是一种质地柔软，可使人免受风寒，给予人们温暖的日常物品。

它是一种使人更加美丽，更受年轻女性欢迎的物品。

它是一种用纽扣、拉链或绳带绑定到身体上的物品。

猜到了吗？其实这是对“衣服”的描述。由于我们都知道什么是衣服，因此我们认为以上 4 种描述都是正确的，通过“免受风寒”这 4 个字便有可能想到是衣服。但对于没见过衣服的人，比如刚出生的小孩儿，他肯定还是不懂，甚至不知道什么是纽扣。

什么是编程语言呢？以下摘自百度百科。

（1）“编程语言"（programming language），是用来定义计算机程序的形式语言。它是一种被标准化的交流技巧，用来向计算机发出指令……

（2）编程语言的描述一般可以分为语法及语义。语法是说明编程语言中，哪些符号或文字的组合方式是正确的，语义则是对于编程的解释……

（3）编程语言俗称“计算机语言”，种类非常多，总的来说可以分成机器语言、汇编语言、高级语言三大类。程序是计算机要执行的指令的集合，而程序全部都是用我们所掌握的语言来编写的……

就像刚才我对衣服的描述，以上的 3 个概念，懂的人早已经懂了，不懂的人还是不懂，回答显得很“鸡肋”。因为对于编程语言的理解并不在语言本身，而是在编译器，编译器是编程语言的“心”，而我们很少有人像了解衣服那样了解编译器，因此对于我们大多数人来说只是熟悉了语言的语法，仅仅是“会用”而已。

那什么是编程语言呢？无论我用多少文字都不足以表述精准与全面，因为语言的本质就是编译器，等你了解编译器后，答案自在心中。目前我只能给出同样“鸡肋”的答案——编程语言是编译器用来“将人类思想转换为计算机行为”的语法规则。

## 0.2 编程语言的来历

世界上本没有编程语言，有的只是编译器。语言本身只是一系列的语法规则， 这个规则对应的“行为”才是我们编程的“意图”，因此从“规则”到“行为”解析便是语言的本质，这就是编译器所做的工作。估计大伙儿都知道，如果想输出字符串，在 PHP 语言中可以用语句 echo，在 C 语言中使用 printf 函数，在 C++中使用 cout，这说明不同的规则对应相同的行为，因此语言规则的多样性只是迷惑人的外表，而本质的行为都是一样的，万变不离其宗。并不是“打印”功能就一定得是 print、out 等相关的字眼儿，那是编译器的设计者为了用户使用方便（当然也是为了他自己设计方便）而采用了大伙儿有共识的关键字，避免不必要的混乱。

## 0.3 语言一定要用更底层的语言来编写吗

有这个疑问并不奇怪，比如：

（1）Python 是用 C 写的，C 较 Python 来说更适合底层执行。

（2）C 代码在编译后会转换为更底层的汇编代码给汇编器，再由汇编器将汇编代码转换为机

器码。

因此给人的感觉是，一种语言必须要用更底层的语言来实现，其实这是个误解。C 只是起初是用汇编语言写的，因为在 C 语言之前只有汇编语言和机器语言。人总是懒惰的，肯定是挑最方便的用，汇编语言好歹是机器语言的符号化，因此相对来说更好用一些，所以只好用汇编来编写 C 语言，等第一版 C 语言诞生后，他们就用 C 语言来写了。什么？用 C 来编写 C？有些读者内心就崩溃了，似乎像是陷入了死循环。其实这根本不是一回事，因为起作用的并不是 C 语言，而是 C 编译器。语言只是规则，编译器产生的行为才是最关键的，编译器就是个程序，C 代码只是它的文本输入。用 C 来编写 C，这就是自举，假如编译器是用别的语言写的，也许你心里就好受一些了。其实只要所使用的语言具有一定的写文件功能就能够写编译器，为什么这么说呢？因为编译器本身是程序，程序本身是由操作系统加载执行的，操作系统识别程序的格式后按照格式读取程序中的段并加载到内存，最后使程序计数器（寄存器 pc 或 ip）跳到程序入口，该程序就执行了。因此用来编写编译器的语言只要具有一定程度的写文件的能力即可，比如至少要具有形同 seek 的文件定位功能，这可用于按照不同格式的协议在不同的偏移处写入数据，因此用 Python 是可以写出 C 编译器的。在这之前我写过《操作系统真象还原》一书，里面的第 0 章第 0.17 小节“先有的语言还是先有的编译器，第 1 个编译器是怎么产生的”，详细地说明 C 编译器是如何自举的，下面我把它贴过来。

首先肯定的是先有的编程语言，哪怕这个语言简单到只有一个符号。先是设计好语言的规则，然后编写能够识别这套规则的编译器，否则若没有语言规则作为指导方向，编译器的编写将无从下笔。第 1 个编译器是怎么产生的，这个问题我并没有求证，不过可以谈下自己的理解，请大伙儿辩证地看。

这个问题属于哲学中鸡生蛋，蛋生鸡的问题，这种思维回旋性质的本源问题经常让人产生迷惑。可是现实生活中这样的例子太多了，具体如下。

（1）英语老师教学生英语，学生成了英语老师后又可以教其他学生英语。

（2）写新的书需要参考其他旧书，新的书将来又会被更新的书参考，就像本书编写过程一样，要参考许多前辈的著作。

（3）用工具可以制造工具，被制造出来的工具将来又可以制造新的工具。

（4）编译器可以编译出新的编译器。

这种自己创造自己的现象，称为自举。

自举？是不是自己把自己举起来？是的，人是不能把自己举起来的，这个词很形象地描述了这类“后果必须有前因”的现象。

以上前 3 个举的都是生活例子，似乎比第 4 个更容易接受。即使这样，对于前 3 个例子大家依然会有疑问：

（1）第一个会英语的人是谁教的？

（2）第一本书是怎样产生的？

（3）第一个工具是如何制造出来的？

其实看到第（2）个例子大家就可能明白了。世界上的第一本书，它的知识来源肯定是人的记忆，通过向个人或群众打听，把大家都认同的知识记录到某个介质上，这样第一本书就出生了。此后再记录新的知识时，由于有了这本书的参考，不需要重新再向众人打听原有知识了，从此以后便形成了书生书的因果循环。

从书的例子可以证明，本源问题中的第一个，都是由其他事物创建出来的，不是自己创造的自己。

就像先有鸡还是先有蛋一样，一定是先有的其他生命体，这个生命体不是今天所说的鸡。伴

随这个生命体漫长的进化中，突然有一天具备了生蛋的能力（也许这个蛋在最初并不能孵化成鸡，这个生命体又经过漫长的进化，最终可以生出能够孵化成鸡的蛋），于是这个蛋可以生出鸡了。过了很久之后，才有的人类。人一开始便接触的是现在的鸡而不知道那个生命体的存在，所以人只知道鸡是由蛋生出来的。

很容易让人混淆的是编译 C 语言时，它先是被编译成汇编代码，再由汇编代码编译为机器码，这样很容易让人误以为一种语言是基于一种更底层的语言的。似乎没有汇编语言，C 语言就没有办法编译一样。拿 gcc 来说，其内部确实要调用汇编器来完成汇编语言到机器码的翻译工作。因为已经有了汇编语言编译器，那何必浪费这个资源不用，自己非要把 C 语言直接翻译成机器码呢，毕竟汇编器已经无比健壮了，将 C 直接变成机器码这个难度比将 C 语言翻译为汇编语言大多了，这属于重新造轮子的行为。

曾经我就这样问过自己，PHP 解释器是用 C 语言写的，C 编译器是用汇编语言写的（这句话不正确），汇编语言是谁写的呢？后来才知道，编译器 gcc 其实是用 C 语言写的。乍一听，什么？用 C 语言写 C 编译器？自己创造自己，就像电影《超验骇客》一样。当时的思维似乎陷入了死循环一样，现在看来这不奇怪。其实编译器用什么语言写是无所谓的，关键是能编译出指令就行了。编译出的可执行文件是要写到磁盘上的，理论上，某个进程，无论其是不是编译器，只要其关于读写文件的功能足够强大，可以往磁盘上写任意内容，都可以生成可执行文件，直接让操作系统加载运行。想象一下，用 Python 写一个脚本，功能是复制一个二进制可执行文件，新复制出来的文件肯定是可以执行的。那 Python 脚本直接输出这样的一个二进制可执行文件，它自然就是可以直接执行的，完全脱离 Python 解释器了。

编译器其实就是语言，因为编译器在设计之初就是先要规划好某种语言，根据这个语言规则来写合适的编译器。所以说，要发明一种语言，关键是得写出与之配套的编译器，这两者是同时出来的。最初的编译器肯定是简单、粗糙的，因为当时的编程语言肯定不完善，顶多是几个符号而已，所以难以称之为语言。只有功能完善且符合规范，有自己一套体系后才能称之为语言。不用说，这个最初的编译器肯定无法编译今天的 C 语言代码。编程语言只是文本，文本只是用来看的，没有执行能力。最初的编译器肯定是用机器码写出来的。这个编译器能识别文本，可以处理一些符号关键字。随着符号越来越多，不断地去改进这个编译器就是了。

以上的符号说的就是编程语言。后来这个编译器支持的关键字越来越多了，也就是这个编译器支持的编程语言越发强大了，可以写出一些复杂的功能的时候，干脆直接用这个语言写个新的编译器，这个新的编译器出生时，还是需要用旧的编译器编译出来的。只要有了新的编译器，之后就可以和旧的编译器说拜拜了。发明新的编译器实际上就是能够处理更多的符号关键字，也就是又有新的开发语言了，这门语言可以是全新的也可以是最初的语言，这取决于编译器的实现。这个过程不断持续，不断进化，逐渐才有了今天的各种语言解释器，这是个迭代的过程。

图 0-1 在网络上非常火，它常常与励志类的文字相关。起初看到这个雕像在雕刻自己时，我着实被感动了，感受到的是一种成长之痛。今天把它贴过来的目的是想告诉大家，起初的编译器也是功能简单，不成规范的，然而经过不断自我“雕刻”，它才有了今天功能的完善。

下面的内容我参考了别人的文章，由于找不到这位大师的署名，只好在此先献上我真挚的敬意，感谢他对求知者的奉献。

要说到 C 编译器的发展，必须要提到这两位大神——C 语言之父 Dennis Ritchie 和 Ken Thompson。Dennis 和 Ken 在编程语言和操作系统的深远贡献让他们获得了计算机科学的最高荣誉，Dennis 和 Ken 于 1983 年赢得了 ACM 图灵奖。

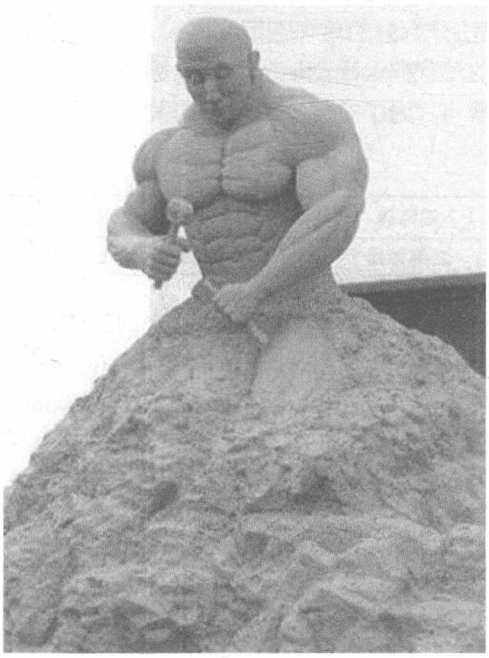

图 0-1

编译器是靠不断学习，不断积累才发展起来的，这是自我学习的过程。下面来看看他们是如何让编译器长大的。

我们都知道转义字符，转义字符是以\开头的多个字符，通常表示某些控制字符，它们通常是不可键入的，也就是这些字符无法在键盘上直接输入，比如\n 表示回车换行，\t 表示 tab。由于以\开头的字符表示转义，因此要想表示\字符本身，就约定用\来转义自己，即\\表示字符\。转义字符虽然表示的是单个字符的意义，在编译器眼里转义字符是多个字符组成的字符串，比如\n 是字符\和 n 组成的字符串。

起初的 C 编译器中并没有处理转义字符，为叙述方便，我们现在称之为旧编译器。如果待编译的代码文件中有字符串\\，这在旧编译器眼里就是\\字符串，并不是转义后的单个字符\。为了表明编译器与作为其输入的代码文件的关系，我们称“作为输入的代码文件”为应用程序文件。尽管被编译的代码文件是实现了一个编译器，而在编译器眼里，它只是一个应用程序级的角色。例如，gcc –c a.c 中，a.c 就是应用程序文件。

现在想在编译器中添加对转义字符的支持，那就需要修改旧编译器的源代码，假设旧编译器的源代码文件名为 compile_old.c。被修改后的编译器代码，已不属于旧编译器的源代码，故我们命名其文件名为 compile_new_a.c，图 0-2 是修改后的内容。

代码 compile_new_a.c

```
1    ...
2    c = next();
3    if(c != '\')
4    return c;
5    c = next();
6    if(c == '\')
7    return '\';
8    ...
```

图 0-2

其中，函数 next()的功能是返回待处理文本（即被编译的源码文件）中的下一字符，强调一下是“单个字符”，并不是记法分析中的单词（即 token）。

用旧编译器将新编译器的源代码 compile_new_a.c 编译，生成可执行文件，该文件就是新的编译器，我们取名为新编译器_a。为了方便理清它们的关系，将它们列入表 0-1 中。

表 0-1

| 编译器自身源代码 | 编译器 | 应用程序源代码 | 输出文件名 |
|---|---|---|---|
| compile_old.c | 老编译器 | compile_new_a.c | 新编译器_a，支持\\ |

这下编译出来的新编译器_a 可以编译含有转义字符\\的应用程序代码了，也就是说，待编译的文件（也就是应用程序代码）中，应该用\\来表示\。而单独的字符\在新编译器_a 中未做处理而无法通过编译。所以此时新编译器_a 是无法编译自己的源代码 compile_new_a.c 的，因为该源文件中只是单个\字符，新编译器_a 只认得\\。

先更新它们的关系，见表 0-2。

表 0-2

| 编译器自身源代码 | 编译器 | 应用程序源代码 | 输出文件名 |
|---|---|---|---|
| compile_old.c | 老编译器 | compile_new_a.c | 新编译器_a，支持\\ |
| compile_new_a.c | 新编译器_a | compile_new_a.c | 编译失败 |

也就是说，现在新编译器_a，无法编译自己的源文件 compile_new_a.c，只有旧编译器才能编译它。再说一下，新编译器_a 无法正确编译自己的源文件 compile_new_a.c 的原因是，compile_new_a.c 中\字符应该用转义字符的方式来引用，即所有用\的地方都应该替换为\\。再回头看一下新编译器_a 的源代码 compile_new_a.c，它只处理了字符串\\，单个\没有对应的处理逻辑。下面修改代码，将新修改后的代码命名为 compile_new_b.c，如图 0-3 所示。

代码 compile_new_b.c

```
1    ...
2    c = next();
3    if(c != '\\')
4    return c;
5    c = next();
6    if(c == '\\')
7    return '\\';
8    ...
```

图 0-3

其实 compile_new_b.c 只是更新了转义字符的语法，这是新编译器_a 所支持的新的语法，下面还是以新编译器_a 来编译新的编译器。

用新编译器_a 编译此文件，将生成新编译器_b，将新的关系录入到表 0-3 中。

表 0-3

| 编译器自身源代码 | 编译器 | 应用程序源代码 | 输出文件名 |
|---|---|---|---|
| compile_old.c | 旧编译器 | compile_new_a.c | 新编译器_a，支持\\ |
| compile_new_a.c | 新编译器_a | compile_new_a.c | 编译失败 |
| compile_new_a.c | 新编译器_a | compile_new_b.c | 新编译器_b，支持\\ |

继续之前再说一下：用编译器去编译另一编译器的源码，也许有的读者觉得很费解，其实你把“被编译的编译器源码”当成普通的应用程序源码就特别容易理解了。上面的编译器代码 compile_new_b.c，其第 3、6、7 行的字符串\\被新编译器_a 处理后，会以单字符\来代替（这是新编译器_a 源码中 return 语句的功能），因此最终处理完成后的代码等同于代码 compile_new_a.c。

现在想加上换行符\n 的支持，如图 0-4 所示。

```
1   if(c == 'n')
2   return '\n';
```

图 0-4

由于现在编译器还不认识\n，故这样做肯定不行，不过可以用其 ASCII 码来代替，将其命名为 compile_new_c.c，如图 0-5 所示。

代码 compile_new_c.c

```
1   ...
2   c = next();
3   if(c != '\\')
4   return c;
5   c = next();
6   if(c == '\\')
7   return '\\';
8   if(c == 'n')
9   return 10;
10  ...
```

图 0-5

用新编译器_a 来编译 compile_new_c.c，将生成新编译器_c，新编译器_c 的代码相当于代码 compile_new_c.c 中所有\\被替换为\后的样子，如表 0-4 所列，暂且称之为代码 compile_new_c1.c，如图 0-6 所示。

代码 compile_new_c1.c

```
1   ...
2   c = next();
3   if(c != '\')
4   return c;
5   c = next();
6   if(c == '\')
7   return '\';
8   if(c == 'n')
9   return 10;
10  ...
```

图 0-6

表 0-4

| 编译器自身源代码 | 编译器 | 应用程序源代码 | 输出文件名 |
|---|---|---|---|
| compile_old.c | 旧编译器 | compile_new_a.c | 新编译器_a，支持\\ |
| compile_new_a.c | 新编译器_a | compile_new_a.c | 编译失败 |
| compile_new_a.c | 新编译器_a | compile_new_b.c | 新编译器_b，支持\\ |
| compile_new_a.c | 新编译器_a | compile_new_c.c | 新编译器_c，间接支持\n |

最后再修改 compile_new_c.c 为 compile_new_d.c，将 10 用\n 替代，如图 0-7 所示。

代码 compile_new_d.c

```
...
c = next();
if(c != '\\')
return c;
c = next();
if(c == '\\')
return '\\';
if(c == 'n')
return '\n';
...
```

图 0-7

用新编译器_c 编译 compile_new_d.c，生成新编译器 d，将直接识别\n。同理，新编译器 d 的代码相当于代码 compile_new_d.c 中，所有字符串\\被替换为字符\、字符\n 被替换为数字 10 后的样子，即等同于代码 compile_new_c1.c，如表 0-5 所列。

表 0-5

| 编译器自身源代码 | 编译器 | 应用程序源代码 | 输出文件名 |
|---|---|---|---|
| compile_old.c | 老编译器 | compile_new_a.c | 新编译器_a，支持\\ |
| compile_new_a.c | 新编译器_a | compile_new_a.c | 编译失败 |
| compile_new_a.c | 新编译器_a | compile_new_b.c | 新编译器_b，支持\\ |
| compile_new_a.c | 新编译器_a | compile_new_c.c | 新编译器_c，间接支持\n |
| compile_new_c.c | 新编译器_c | compile_new_d.c | 新编译器 d，直接支持\n |

编译器经过这样不断地训练，功能越来越强大，不过占用的空间也越来越大了。

## 0.4 编译型程序和脚本程序的异同

两者最明显的区别就是看它们各是谁的“菜”。两者的共性是最终生成的指令都包含操作码和操作数两部分。

编译型程序所生成的指令是二进制形式的机器码和操作数，即二进制流。同样是数据，和文本文件相比，这里的数据是二进制形式，并不是文本字符串（如 ASCII 码或 unicode 等）形式。如果二进制流按照有无格式来划分，无格式的便是纯粹的二进制流，程序的入口便是文件的开始。另外一种是按照某种协议（即格式）组织的二进制流，比如 Lnux 下 elf 格式的可执行文件。它是硬件 CPU 的直接输入，因此硬件 CPU 是“看得到”编译型程序所对应的指令的，CPU 亲自执行它，即机器码是 CPU 的菜。编译型语言编译出来的程序，运行时本身就是一个进程，它是由操作系统直接调用的，也就是由操作系统加载到内存后，操作系统将 CS:IP 寄存器（IA32 体系架构的 CPU）指向这个程序的入口，使它直接上 CPU 运行，这就是所说的 CPU“看得到”它。总之调度器在就绪队列中能看到此进程。

脚本语言，也称为解释型语言，如 JavaScript、Python、Perl、Php、Shell 脚本等。它们本身是文本文件，是作为某个应用程序的输入，这个应用程序是脚本解释器。由于只是文本，这些脚本中的代码在脚本解释器看来和字符串无异。也就是说，脚本中的代码从来没真正上过 CPU 去执行，CPU 的 CS：IP 寄存器从来没指向过它们，在 CPU 眼里只看得到脚本解释器，而这些脚本中的代码，CPU 从来就不知道有它们的存在，脚本程序却因硬件 CPU 而间接“运行”着。这就像

家长给孩子生活费，孩子用生活费养了只狗狗，家长只关心孩子的成长，从不知道狗狗的存在，但狗狗却间接地成长。这些脚本代码看似在按照开发人员的逻辑在执行，本质上是脚本解释器在时时分析这个脚本，动态根据关键字和语法来做出相应的行为。解释器有两大类，一类是边解释边执行，另一类是分析完整个文件后再执行。如果是第一类，那么脚本中若有语法错误，先前正确的部分也会被正常执行，直到遇到错误才退出；如果是第二类，分析整个文件后才执行的目的是为了创建抽象语法树或者是用与之等价的遍历去生成指令，有了指令之后再运行这些指令以表示程序的执行，这一点和编译型程序是一致的。

脚本程序所生成的指令是文本形式的操作码和操作数，即数据以文本字符串的形式存在。其中的操作码称为 opcode，通常 opcode 是自定义的，所以相应的操作数也要符合 opcode 的规则。为了提高效率，一个 opcode 的功能往往相当于几百上千条机器指令的组合。如果虚拟机不是为了效率，多半是用于跨平台模拟程序运行。这种虚拟机所处理的 opcode 就是另一体系架构的机器码，比如在 x86 上模拟执行 MIPS 上的程序，运行在 x86 上的虚拟机所接收的 opcode 就是 MIPS 的机器码。除跨平台模拟外，通常虚拟机的用途是提高执行效率，因此 opcode 很少按照实际机器码来定义，否则还不如直接生成机器指令交给硬件 CPU 执行更快呢。故此种自定义的指令是虚拟机的输入，即所谓虚拟机的菜。虚拟机分为两大类，一类是模拟 CPU，也就是用软件来模拟硬件 CPU 的行为，这种往往是给语言解释器用的，比如 Python 虚拟机。另一类是要虚拟一套完整的计算机硬件，比如用数组虚拟寄存器，用文件虚拟硬盘等，这种虚拟机往往是用来运行操作系统的，比如 VMware，因为只有操作系统才会操作硬件。

脚本程序是文本字符流（即字符串），其以文本文件的形式存储在磁盘上。具体的文本格式由文本编译器决定，执行时由解释器将其读到内存后，逐行语句地分析并执行。执行过程可能是先生成操作码，然后交给虚拟机逐句执行，此时虚拟机起到的就是 CPU 的作用，操作码便是虚拟机器的输入。当然也可以不通过虚拟机而直接解析，因为解析源码的顺序就是按照程序的逻辑执行的顺序，也就是生成语法树的顺序，因此在解析过程中就可以同时执行了，比如解析到 2+3 时就可以直接输出 5 了。但方便是有限的，实现复杂的功能就不容易了，因为计算过程中需要额外的数据结构，比较对于函数调用来说总该有个运行时栈来存储参数和局部变量以及函数运行过程中对栈的需求开销。因此对于复杂功能，多数情况下还是专门写个虚拟机来完成。

顺便猜想一下解释型语言是如何执行的。我们在执行一个 PHP 脚本时，其实就是启动一个 C 语言编写出来的解释器而已。这个解释器就是一个进程，和一般的进程是没有区别的，只是这个进程的输入则是这个 PHP 脚本。在 PHP 解释器中，这个脚本就是个长一些的字符串，根本不是什么指令代码之类。只是这种解释器了解这种语法，按照语法规则来输出罢了。举个例子，假设下面是文件名为 a.php 的 PHP 代码。

```
<?php                    这是 php 语法中的固定开始标签
   echo "abcd";          输出字符串 abcd
?>                       固定结束标签
```

php 解释器分析文本文件 a.php 时，发现里面的 echo 关键字，将其后面的参数获取后就调用 C 语言中提供的输出函数，比如 printf((echo 的参数))。PHP 解释器对于 PHP 脚本，就相当于浏览器对于 JavaScript 一样。不过这个完全是我猜测的，我不知道 PHP 解释器里面的具体工作，以上只是为了说清楚我的想法，请大家辩证地看。

说到最后，也许你有疑问，如果 CPU 的操作数是字符串的话，那 CPU 就能直接执行脚本语言了，为什么 CPU 不直接支持字符串作为指令呢？后面小节有分享。

## 0.5 脚本语言的分类

脚本语言大致可分为以下 4 类。

（1）基于命令的语言系统

在这种语言系统中，每一行的代码实际上就是命令和相应的参数，早期的汇编语言就是这种形式。此类语言系统编写的程序就是解决某一问题的一系列步骤，程序的执行过程就是解决问题的过程，就像做菜一样，步骤是提前写好在脑子里（或菜谱中）的。如以下炒菜脚本。

```
WashVegetable  tomato  3   // 洗 3 个西红柿
chopVegetable  tomato  3   // 切 3 个西红柿
heatPot  iron          // 加热铁锅
pourOilIntoPot  50     // 放油 50 克
putVegetable  tomato   // 放入西红柿
stir       //搅拌菜
stir       //搅拌菜
stir       //搅拌菜
……略
```

以上步骤中第 1 列都是命令，后面是命令的参数。其中把菜放进锅后不断地搅拌（示意而已，不用太严谨），由于命令式语言系统中没有循环语句，需要连续填入多个 stir 以实现连续多个相同的操作。会有一个解释器逐行分析此文件，执行相应命令的处理函数。以下是一个解释器示例。

```
#define MAX_CMD_BUF 1024;
char line[MAX_CMD_BUF] = {'\0'};
#define MAX_CMD_ARG 10;
char* fields[1 + MAX_CMD_ARG];  //最多支持 9 个参数的命令
while (getline(line)) {
  //split 是自定义函数，将参数 1 按照参数 2 拆分成多个字段，
  //各字段起始地址放在 fields 数组中，参数 2 被替换为\0
   split(line, ' ', fields);
   if (memcmp(fields[0], "WashVegetable", strlen(WashVegetable)) == 0) {
       doWashVegetable(fields[1], fields[2]);
   } else if (memcmp(fields[0], "chopVegetable", strlen(chopVegetable)) == 0) {
       doChopVegetable(fields[1], fields[2]);
   } else if (memcmp(fields[0], "heatPot", strlen(heatPot)) == 0) {
       doHeatPot(fields[1]);
   } else if (memcmp(fields[0], "pourOilIntoPot", strlen(pourOilIntoPot)) == 0) {
       doPourOilIntoPot(fields[1]);
   } else if (memcmp(fields[0], "putVegetable", strlen(putVegetable)) == 0) {
       doPutVegetable(fields[1]);
   } else if(memcmp(fields[0], "stir", strlen(stir)) == 0) {
       doStir()
   }
   ……
}
```

（2）基于规则的语言系统

此类语言的执行是基于条件规则，当满足规则时便触发相应的动作。其语言结构是谓词逻