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

异步社区 www.epubit.com

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

异步社区 www.epubit.com.cn

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。

异步社区 www.epubit.com.cn

◆ 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

推 荐 序

很高兴能成为本书的首批读者，也很高兴能为本书写推荐序。

刚拿到本书手稿时，从书名上我意识到这是对我胃口的书。果然，整书阅读以后，收获颇多。如今程序员的开发成本已经很低了，项目中有各种成熟的框架和库可供选择和使用，但还有人能静下心来研究编译器这么底层的技术，实属难得。本书犹如一把火炬，点燃了技术人内心对开发的热情。

依稀记得 2010 年年初在百度与郑钢初次见面的情景，那时他工作之余的时间基本都用在向各个技术专家请教、讨论各类技术问题上，他是我带过的人中最勤奋的人之一。时光荏苒，一分耕耘一分收获，看到他今天的成长，尤感欣慰。

本书讲述了一门脚本语言（sparrow）的开发过程，这是一本“步步为营”式的书籍，延续了他编写《操作系统真象还原》的风格，手把手地教读者从零实现一门语言，从原理到实践每一步都有实际的代码和详尽的原理说明，通过运行书中各小节中的代码，读者可以很轻松地掌握各个细节，因此本书的学习曲线并不陡峭，甚至很平坦。另外，值得欣喜的是，本书所编写的脚本语言并不是用 Java、C++等入门难度略大的语言实现的，而是用 C 语言，这是我们学习编程的基础语言。也就是说，本书不需要专业的开发经验即可上手学习。另外，在实现过程中并未用到复杂的库函数或系统调用，可以负责地说，本书已经将学习成本降到最低。

C 语言是一种面向过程的语言，如何用一种面向过程的语言去实现一种面向对象的语言很有意思。另外，PHP 和 Perl 语言虽然也实现了类，但它们其实是一种面向过程的语言，并不是纯粹的面向对象语言，而 sparrow 语言是一种纯粹的面向对象语言，它在设计之初就采用对象的方式来处理脚本语言中类的成员和方法，这仿佛让我们看到了面向对象编程语言的基因。众所周知，当今流行的脚本语言应属 Python，Python 也是用 C 语言实现的，也许你很好奇 Python 的内部原理，但是想到它有将近 4 万行的源代码时，也许甚至不想看它的源程序了。那么研读本书中的 sparrow 语言会是一种更好的选择，其源码不足 7100 行，阅读过程轻松愉快，但可以学到 Python 这种语言的实现原理。

对于脚本语言来说，两个重要方面就是垃圾回收和运行环境。垃圾回收就是我们平时所说的 GC（Garbage Collection）。有了 GC，程序员不需要手工释放所分配的对象，可以使精力专注于业务逻辑而不用担心内存泄漏问题。在 sparrow 语言中同样实现了 GC，通过此部分代码你可以看到 GC 的原理，以及哪些对象才能被回收。 运行时环境就是脚本语言中的虚拟机，即 VM（如 Java 语言的 JVM 也是一种 VM）。脚本语言是通过虚拟机才能运行的，如何把编译器生成的操作码转换为实际的代码，这里面的工作对大多数人来说很神秘。相信各位在源码中一探究竟之后会发现：GC 和 VM 这两个神秘的黑盒子不过如此。另外，也许程序员最感兴趣的就是线程，关于线程在用户态下是如何实现的、线程如何实现调度，本书将告诉你答案。总之，但凡涉猎，开卷有益。

每个程序员都有实现属于自己编程语言的梦想，说其是梦想，原因是实现的难度很大.....本书讲的是纯粹的技术“干货”，符合郑钢一贯的写作风格，这是他静心写出来的东西，内容满满，很值得阅读。

于晓声

滴滴系统部技术高级总监

业界热评

本书详细阐述了设计编程语言所需的基本理论，并且以作者自己开发设计的 sparrow 编程语言为例，引导读者一步一步地实现一门完善的编程语言，本书还讲解了大量的基础类以及垃圾内存回收功能，切实帮助读者从理论过渡到实践，再走向实用。

——肖金亮，阿里蚂蚁金服技术专家

日常有些运维、开发的工作之所以难以开展，很多时候就受限于对底层技术的不了解，本书从独立开发一门编程语言和虚拟机的实践入手，对相关知识进行了阐述，讲解很清楚，实现的技术很值得称赞。

——陈晓聪，百度资深运维工程师

回顾计算机技术发展的这几十年，编程语言层出不穷，语言特性愈发抽象，语言使用则愈发简洁，底层机制隐藏得也越来越深。这种情况下，程序员想深入理解编程语言原理愈发困难，需要花费大量精力去学习语言设计理论并深入阅读语言实现的源码，但往往事倍功半，收效甚微。本书另辟蹊径，带领读者从零开始自己动手实现一个编程语言及其运行环境，循序渐进，在实践过程中透彻理解编程语言的来龙去脉。

——冯顾，360 企业安全集团政企云事业部技术总监

在云与人工智能时代的大背景下，软件从业者都有必要去了解一下虚拟机与一门编程语言。通过对这部分知识的了解能够对操作系统和语言编程有更深刻的理解。市场上介绍编程语言和操作系统入门的书籍不多，而本书是一本较好的读物，很值得读者学习。

——陆景玉，ACFUN 高级运维总监

自制编程语言和虚拟机，这是一个看似很深奥的课题，也涉及当今互联网流行的主题，许多技术人员对其心驰神往，但要领悟其精髓步履维艰。本书循序渐进、由浅到深地讲解了丰富的基础知识，覆盖了常见的编译原理入门知识，更难能可贵的是，作者讲解的知识具有其独特的理解和视角，相信本书能让读者受益匪浅。

——黄梦溪，Mobvista 运维总监

异步社区 www.ituibi.com.cn

前　言

很多读者看了我写的《操作系统真象还原》（一本一步步编写操作系统的书）书后，纷纷来信，要求我再写一本自制编程语言的书。这也在情理之中，对于很多计算机从业者来说，操作系统和编译器几乎是两座无法逾越的大山，其难度之大，令很多人员望而生畏。最终，在读者的鼓励下，一冲动就答应了写作本书，其实我很“后悔”做出这样的决定。

为什么后悔呢？因为写书代价很大。

首先，写书相当累，占用很多精力。其次，占用自己学习的时间，在当今个人进步缓慢就算退步的时代，自己没有提升技术会很恐慌。再次，精力全放在写书上会影响家庭、影响工作。最后，还要负责解答许多问题，确实很累。而且，这一次可是在创造编程语言，难度系数太高了，不亚于开发一个操作系统，甚至我父母都劝我：小刚，你都多大了还写书，好好过日子、踏实上班就行了。但是，我最后还是决定写本书。

下面是我跨过重重拦阻创作本书的动机。

1. 有梦想，有远方

既然写书代价那么大，那我为什么还要“明知山有虎，偏向虎山行”呢？因为我就是奔着“老虎”去的，没有老虎的山就没有探险的乐趣。

2. 有难度才有价值

每次遇到一件很难的工作时，我先是“痛苦”，然后随之而来的就是“兴奋”，因为这意味着我要进步。也许读者会说，一定会进步吗？也许 99%会失败。

同样一件事，每个人对它的态度都不同，懦夫看到的是：99%会失败，别干了。勇士看到的是：还有 1%成功的机会，干吧！

只要不放弃（注意，不是坚持），一定会成功，成功只是时间长短的问题。

3. 人生的意义

人生最大的遗憾是“壮志未酬”。如果你是天才，请将自己的才华“挥霍”得一滴不剩，直到触碰到自己智力上的天花板，这样才甘心。如果你是大力士，请努力在奥运赛场上为国争光，直到累得站不起来，这样才甘心。

这正是我写本书的信仰。

学习很累并且无止境，但是多知道一些就会有多一些的欣喜。本着“把自己的知识多掏点给大家”的诚意，本书依然从第 0 章开始，相对《操作系统真象还原》来说，本书的语言不再那么活泼（啰唆）了，毕竟编译器的开发难度略小于开发操作系统，没必要穿插一些“过渡”的话题。本书一步步地实现了一种称为 sparrow 的编程语言，它是用虚拟机运行的，因此最后还要实现一个虚拟机。sparrow 语言是用 C 语言编写的，学习的难度较低，实现的代码不长，希望大家在学习的旅途中愉快。

4. 作者介绍

郑钢，大家都叫我刚子，毕业于北京大学，前百度运维开发工程师，《操作系统真象还原》作

异步社区 www.epubit.com.cn

者。爱父母、爱老婆、爱街舞、爱学习、爱养生。欢迎关注作者的微博“程序员刚子”。

谨以此书感谢我的妻子：有了你对我无微不至的照顾，本书才得以顺利完成。也许以后会有很多人通过本书写出自己的编程语言，成为某某语言之父，这其中也有你的功劳，在这里轻轻地说一声：你辛苦了，我永远爱你！

本书读者 QQ 交流群 148177180，欢迎大家随时沟通。源码下载地址为人民邮电出版社官网。

本书编辑和投稿邮箱为：zhangtao@ptpress.com.cn。

本书适合对编译原理及语言处理器设计有兴趣的读者，以及正在学习相关课程的大中专院校学生。同时，已经学习过相关知识，有一定经验的开发者，也一定能从本书新颖的实现方式中受益良多。

作者

异步社区 www.epubit.com.cn

目　录

www.epubit.com.cn 异步社区

异步社区 www.epubit.com.cn

异步社区 www.epubit.com

异步社区 www.epubit.com.cn

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

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

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\\

www.epubit.com.cn

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。

www.enubit.com.cn

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

辑→动作，如图 0-8 所示。

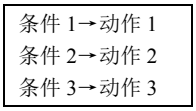


图 0-8

因此此类语言常称为逻辑语言，常用于自然语言处理及人工智能方面，典型的代表有 Prolog。

（3）面向过程的语言系统

面向过程的语言系统我们都比较熟悉，批处理脚本和 shell 脚本，perl、lua 等属于此类，和基于命令的语言系统相比，它可以把一系列命令封装成一个代码块供反复调用。此代码块便是借用了数学中函数的概念，一个 x 对应一个 y，即给一个输入便有一个输出，于是这个代码块便称为函数。

（4）面向对象的语言系统

现代脚本语言基本上都是面向对象，大伙儿用的都挺多的，比如 python。很多读者误以为只要语言中含有关键字 class，那么该语言就是面向对象的语言，这就不严谨了。因为在 perl 语言中也可以通过关键字 class 定义一个类，但其内部实现上并不是完全面向对象，其本质是面向过程的语言。世界上第一款血统纯正的面向对象语言是 smalltalk，它在实现上就是一切皆对象，具有完全面向对象的基因。

0.6 为什么 CPU 要用数字而不是字符串作为指令

在之前小节“编译型程序和脚本程序的异同”的结束处我们讨论过，为什么 CPU 不直接支持字符串作为指令。我估计有的读者会误以为 CPU 将直接执行汇编代码，这是不对的，因为汇编代码是机器码的符号化表示，几乎是与机器码一一对应，但汇编代码绝对不是机器语言。你想，如果汇编代码是机器指令的话，那么 CPU 看到的输入便是字符串，比如以下汇编代码用于计算 1+10-2。

```
mov  a,  1
add  a,  10
dec  a,  2
```

汇编语言其实是汇编器的输入，对于汇编器来说，汇编代码文件也是文本，因此其中 mov 指令也是字符串。如果让 CPU 直接读取汇编文件逐行分析各种字符串以判断指令，这效率必然非常低下。毕竟要比较的字符数太多，比较的次数多了效率当然就低了，因此把指令编号为数字，这样比较数字多省事。而且最主要的是，CPU 更擅长处理数字，它本身的基因就是数字电路，数字计算是建立在数值处理的基础上，这就是本质上二进制数据比文本 ASCII 码更快更紧凑的原因。

0.7 为什么脚本语言比编译型语言慢

而脚本语言的编译有两类，一类是边解释边执行，不产生指令，这个解释过程最占时间的部分就是字符串的比较过程，字符串比较的时间复杂度是 O(n)，也就是在比较 n 次之后解释器才确定了操作码是什么，然后再去获取操作码的操作数，你看能不慢吗？而编译型语言编译后是机器码，是二进制数字，因此可直接上 CPU 运行，而 CPU 擅长处理数字，比较一次数字便可确定操

www.epubit.com.cn

作码。另一类脚本语言是先编译，再生成操作码，最后交给虚拟机执行，这样多了一个生成操作码的过程，似乎“显得”更慢了。其实这都不是主要的。你看，程序“执行”速度的快慢是比较出来的，编译型语言在执行时已经是二进制语言了，而大多数脚本语言在执行时还是文本，必然要先有个编译过程。这里面全是字符串处理，整个脚本的源码对于编译器来说就是一个长长的字符串，都要完整地进行各种比较，因此多了一个冗长的步骤，必然要慢。有些脚本系统为减少编译的过程，第一次编译后将编译结果缓存为文件，如 Python 会将.py 文件编译后存储为.pyc 文件，下次无须编译直接运行便可。但是，这样无须二次编译的脚本语言就能和编译型程序媲美吗？不见得磁盘 IO 是整个系统最慢的部分，解释器读取缓存文件难道不需要时间吗？等等，有读者说了，编译型的程序被操作系统加载时也要从磁盘上读取啊，这不一样吗？当然不一样，别忘了，脚本程序在执行时先要加载解释器，解释器也是位于硬盘上的文件，只是二进制可执行文件而已，依然需要读取硬盘，然后解释器再去从硬盘上读取脚本语言文件并编译脚本文件。你看，编译型程序在执行时只有 1 个 IO，而脚本程序在执行时有两个，比前者多了 1 个低速的 IO 操作，因此，脚本语言更慢一些是注定的。

0.8 既然脚本语言比较慢，为什么大家还要用

这里的语言是指语言的编译器或解释器，以下简称为语言。

语言慢并不影响整个系统，影响整个系统速度的短板并不是语言本身，目前来说系统的瓶颈普遍是在 IO 部分。语言再慢也比 IO 快一个数量级，并不是语言执行速度快 10 倍后整个系统就快 10 倍，语言慢了，整个系统依然不受影响，这要看瓶颈是哪块儿。这就像动物园运送动物的船超载了，人们不会埋怨某些人太胖了，而是清楚地知道占分量的主要是船上的大象，人的体重和大象根本就不是一个量级。

再说，即使是语言提速后，由于 IO 这块跟不上，依然会被阻塞（由于是脚本语言，这里阻塞的是脚本解释器），而且由于语言太慢而显得阻塞时间更漫长。为什么会阻塞呢？这种阻塞往往是由于程序后续的指令需要从 IO 设备读取到的数据，也就是说程序后面的步骤依赖这些数据，没这些数据程序运行没意义。比如说 Web 服务器先要读取硬盘上的数据然后通过网卡发送给用户，必须获得硬盘数据后，web 服务器进程中那部分操作网卡发送数据的指令才能上 CPU 上执行。由于语言的解释器是由 CPU 处理的，CPU 速率肯定比 IO 设备快太多，因此在等待 IO 设备响应的过程中啥也干不了。操作系统为了让宝贵的 CPU 资源得到最大的利用，肯定会把进程（二进制可执行程序或脚本语言的解释器）加入阻塞队列，让其他可直接运行的、不需要阻塞的进程使用 CPU（阻塞指的是并不会上 CPU 运行，也就是将该进程从操作系统调度器的就绪队列中去掉）。而语言（脚本语言解释器）再慢也比 IO 设备快，因此依然会因为更慢的 IO 而难逃阻塞的命运。也就是说，拖慢整个系统后腿的一定是系统中最慢的部分，而无论脚本语言多慢，IO 设备总是会比语言更慢，因此“影响系统性能”这个黑锅，脚本语言不能背。

另一方面大伙儿喜欢用脚本语言的原因是开发效率高，这也是脚本语言被发明的初衷，很多在 C 中需要多个步骤才能实现的功能在脚本语言中一句话就搞定，当然更受开发人员欢迎了。

0.9 什么是中间代码

很多编译器会将源语言先编译为中间代码，最后再编译为目标代码，但中间语言并不是必需的。中间代码简称 IR（Intermediate Representation），是介于源程序和机器语言之间的语言，有 *N* 元式（如三元式、四元式）、逆波兰、树等形式。目标代码是指运行在目标机器上的代码，与目

标机器的体系架构直接相关，编译器干吗不直接生成目标代码，多这一道程序有什么好处呢？

（1）可以跨平台

由于中间代码并不是目标代码，因此可以作为所有平台的公共语言，从而可通过中间代码实现前后端分离。比如在多平台、多语言的环境下开发可提高开发效率，只要在某一平台上编译出中间代码后，中间代码到目标代码的剩余工作可以由目标平台的编译器继续完成。

（2）便于优化

中间代码更接近于源代码，对于优化来说更直接有效。而且可以在一种平台上优化好中间代码，再发送到其他平台编译为目标机器，提高优化效率。

0.10 什么是编译器的前端、后端

编译器的前后端是由中间代码来划分的，如图 0-9 所示。

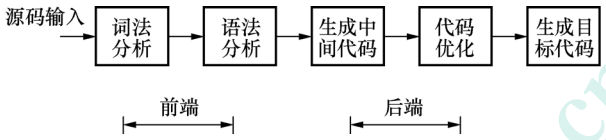


图 0-9

前端主要负责读取源码，对源码进行预处理，通过词法分析把单词变成 Token 流，然后进行语法分析，语义分析，将源码转换为中间代码。

后端负责把中间代码优化后转换为目标代码。

0.11 词法分析、语法分析、语义分析和生成代码并不是串行执行

很多教材上会把编译阶段分为几个独立的部分：

（1）词法分析；

（2）语法分析；

（3）语义分析；

（4）生成中间代码；

（5）优化中间代码；

（6）生成目标代码。

这容易给人造成“这几个步骤是串行执行”的错觉，即“从源码到目标代码必须要顺序地执行这 6 个步骤”，其实不是这样子的，至少一个高效的编译器绝不会这样做。这只是在功能逻辑上的步骤，就拿前 4 步来说，它们是以语法分析为主线，以并行的、穿插的方式在一起执行的，即这 4 个步骤是随语法分析同时开始，同时结束。

每个步骤的功能实现由其实际的模块完成，负责词法分析的模块称为词法分析器，负责生成代码的模块称为代码生成器，负责语法分析的模块称为语法分析器。我们所说的编译器就是由词法分析器、语法分析器和代码生成器组成的（如果有目标代码优化的话还包括优化模块）。编译工作的入口是语法分析，因此编译是以调用语法分析器为开始的，语法分析器会把词法分析器和代码生成器视为两个子例程去调用。换句话说，词法分析器和代码生成器只会被语法分析器调用，如果没有语法分析器，它们就没有“露脸儿”的机会。因此说编译是以语法分析器为主线，由语

异步社区 www.epubit.com.cn

法分析器穿插调用词法分析器和代码生成器并行完成的。

语法分析和语义分析尽管是两个功能，但这其实可以合并为一个。因为在语法分析过后便知道了其语义。这个很好理解，毕竟语法就是语义的规则，规则是由编译器（的设计者）制定的，那么编译器（的设计者）分析了自己设定的规则后当然就明白了语义（不可能不明白自己所制定规则的意义）。比如读英文句子，尤其是复杂的长句，先找到句子谓语动词，以谓语动词为分界线把句子拆分主谓两大部分，在前一部分中找主语，后一部分中找宾语等，在分析完语法后句子的意思就搞清楚了。也就是说，语法分析和语义分析是同时，又是前后脚的事儿，因此合并到一起并不奇怪。你看，语法分析和语义分析确实是并行。

为了语法分析的效率，词法分析器往往是作为一个子例程被语法分析器调用，即每次语法分析器需要一个单词的 token 时就调用词法分析器。你看，语法分析和词法分析确实也是并行。

最后说生成代码。目前生成代码的方式叫语法制导，什么是语法制导呢？就是在分析语法的“同时”生成目标代码或中间代码，实际上就是以语法分析为导向，语法分析器在了解源码语义后立即调用代码生成器生成目标代码或中间代码，因此这也是和语法分析器并行。提醒一下，并不是在语法分析器分析完整个源码后，再一次性地生成整个源码对应的目标代码或中间代码，而是分析一部分源码后就立即生成该部分源码对应的目标代码或中间代码，这样做比较高效且更容易实现。举个例子，比如源码文件中有 10 行代码，语法分析器不断调用词法分析器，每次获得一个单词的 token，把前 3 行源码都读完后确定了源码的语义，立即生成与这 3 行源码同等意义的目标代码或中间代码。然后语法分析器继续调用词法分析器读取第 4 行之后的源码，重复分析语法、生成代码的过程。总之是以语法分析为主线，语法分析把源码按照语法来拆分成多个小部分，每次生成这一小部分的目标代码或中间代码。

总结，为了使编译更加高效，词法分析、语法分析、语义分析和生成代码是以语法分析为中心并行执行的，词法分析和生成代码都是被语法分析器调用的子例程。

0.12 什么是符号表

把符号表列出来是因为这个词听上去“挺唬”人的，由于看不见摸不着，很多初学者都以为它是个非常神秘的东西。其实符号表就是存储符号的表，就是这么简单。你想，源码中的那些符号总该存储在某个地方，这样在引用的时候才能找得到，因此符号表的用途就是记录文件中的符号。符号包括字符串、方法名、变量名、变量值等。符号放在表中的另一个重要原因是便于生成指令，使指令格式统一。编译器会把符号在符号表中的索引作为指令的操作数，如果不用索引的话，指令就会很乱，比如若直接用函数名或字符串作为操作数，指令就冗长了。“表”在计算机中并不专指“表格”，“表”是个笼统的概念，用以表示一切可供增、删、改、查的数据结构，因此符号表可以用任何结构来实现，比如链表、散列表、数组等。

0.13 什么是关系中的闭包

任何事物都会根据其特征来命名，先看看闭包中的“闭”指的是什么。

假如某集合中元素的运算结果还在该集合中，我们称具有这种特征的运算是封闭的，简称闭运算。比如整数集合中的两个整数相加，其结果依然是整数，依然属于整数集合，这就是“闭”的意义。下面先上点甜点。

（1）序偶

序是指顺序，偶是指成对儿。因此序偶是与顺序相关的、一组成对儿出现的两个元素。序偶

类似于两个元素的集合，但由于与顺序相关，故又不与集合完全相同，比如（小兔，大狗）不等于（大狗，小兔）。

（2）笛卡儿积

有 A 和 B 两个集合，假设某类序偶的第 1 个元素来自集合 A，第 2 个元素来自集合 B，所有这类序偶组成的集合便称集合 A 和集合 B 的笛卡儿积。相当于两个集合中所有元素一一连接，类似数学中多项式相乘，因此笛卡儿积记作 $A\times B$。注意，其中的×是乘号，不是字母 x 的大写形式。

（3）方幂

一个符号串 x 与其自身的 n-1 次连接称此符号串 x 的 n 次方幂，记作 x^n。即 $x^1=x$，$x^2=xx$，$x^3=xxx$……特别地，定义 $x^0=\varepsilon$。

（4）集合的方幂

集合的方幂是集合与自身的连接，如 $A^1=$A，$A^2=AA$，$A^3=AAA$，……，特别地，定义 $A^0=\{\varepsilon\}$。集合的连接运算就是笛卡儿积。若 A 为$\{a,b,c\}$，那么

$A^1=\{a,b,c\}$

$A^2=\{aa,ab,ac,ba,bb,bc,ca,cb,cc\}$

$A^3=\{aaa,aab,aac,aba,abb,abc,aca,acb,acc,...ccc\}$

……

现在可以说闭包了。

闭包：

集合 A 所有方幂的集合称为闭包 A，记作 A^*。

$$A^*=A^0\cup A^1\cup A^2\cup A^3\cup\cdots\cdots$$

闭包 A 去除 A^0 就称为正闭包 A，记作 A^+，也就是说正闭包比闭包少了一个 ε。

0.14 什么是程序中的闭包

程序中的闭包是什么？最直观的解释是闭包是在函数中定义的函数，内部函数引用了外层函数中的局部变量。

程序中的闭包既然也叫“闭”，当然也具备封闭的概念。把什么封闭了呢？封闭的是外部函数作用域中的局部变量，该局部变量称为自由变量。先通过例子建立个感性认识。

```
[work@work tmp]$ cat -n closure.py
     1 def foo ():
     2     vec = [0]
     3     def bar ():
     4         vec[0] += 1
     5         print vec[0]
     6     return bar
     7
     8 foobar = foo()
     9 foobar()
    10 foobar()
    11 foobar()
[work@work tmp]$ python closure.py
1
```

异步社区 www.epubit.com.cn

```
2
3
[work@work tmp]$
```

在 foo 函数内部定义了 bar 函数，bar 函数引用了外部函数 foo 中的 list 元素 vec[0]，对于 bar 函数来说，vec[0]就是自由变量，bar 函数使其自加 1 后打印。但 bar 函数对外并不可见，外部没法直接调用它。为了调用 bar 函数，使 bar 作为 foo 函数的返回值返回给调用者。因此在第 8 行调用 foo 之后返回的是 bar 函数，随后是其输出，符合预期。

有了感性认识后再看下其定义：计算机科学中，闭包（Closure）是引用了自由变量的函数。即使自由变量原来所属的内存空间不存在了，该自由变量也依然与对该函数有效。闭包是函数和其相关的“环境”组成的实体。

我们知道，任何变量都是存在于内存中的，若变量所属的内存被回收了，变量肯定就不存在了，这是常识。可上面定义中说自由变量所属的内存空间不在了（也就是内存空间被回收了）变量也依然有效，看上去这很反常理。因此，要让这一切变得合理的话，只能说明是那个“环境”起了关键作用，并且一定是变量的引用机制对该环境做了很好的屏蔽。用户程序并不知道有那个“环境”的存在，也就是说变量的引用机制会判断：如果自由变量的空间被回收了，就从“环境”中获取变量，否则就去原空间获取。

再说下闭包中的“闭”。闭在此就是封闭，封闭的目的就是为了提供环境。就像院子一样，用院墙封闭起来以构建人的生活环境，在院子里厨房、柴房、厕所都有了，人才能舒适地在里面生存。那闭包为函数提供了什么环境？提供的是自由变量的存储空间。

只有在自由变量原来的空间消失后，编译器才会为该变量提供环境，因此最好先弄明白变量所属的空间为何会消失。这里面涉及局部变量的存储了，我们知道局部变量是存储在栈中的，函数在调用时才会被分配运行时栈，那时才会为函数中的局部变量分配空间。当函数运行结束后，该函数的运行时栈也要被回收，毕竟栈本身占用的空间也在内存中，必须要及时回收以备它用。可是，内部函数引用了外部函数中的局部变量，外部函数调用结束后，外部函数的运行时栈被回收了，位于该运行时栈中的局部变量也就消失了，内部函数引用了那个已经消失的局部变量就会出大问题。闭包的作用就体现在这里，它要为内部函数提供引用该变量的环境，即尽管该变量所在的存储空间消失了，但它额外提供个环境来存储该变量以供内部函数引用，相当于给内部函数“包”了一层环境。该环境是“闭”合的，只供该内部函数使用，因此内部函数便能安心地使用“已经消失”的自由变量。顺便说一句，如何保证自由变量只供内部函数使用？答案是只要保证只有该内部函数能访问到此自由变量就行了。是这样的，代码中定义的函数有很多，为了跟踪各个函数，编译器为每个函数定义了一个结构，这个结构不仅起到标识函数的作用，它也正是我们所说的“环境”，这就是自由变量原空间被回收后的新家。此结构是与函数绑定的，因此保证了只有函数自己独享，即自由变量只供它自己使用。

这下大伙儿也清楚自由变量中“自由”二字的意义了。什么是自由？就是无拘无束不受限制，在此表示变量不受其存储空间的限制，即使它所属的空间被回收了，它也依然还存活。当然，大伙儿都明白这只是“表相”，实际上是编译器给它另外找了个地方安身，在后面实践中你会有更深认识。没学过编译原理的读者最好是按顺序阅读以下内容。

0.15 什么是字母表

字母表是一个有穷符号组成的集合，也就是所能表示的符号的范围，用 Σ 表示。注意，符号并不专指 Φ、λ、Ω 等传统意义的符号，像文字、数字、英文字母等在此都算符号。任何表达式都

是由符号组成的，符号肯定属于某个集合，表达式也只在该符号集合上有效，字母表就是符号集。比如汉字表达式所属的字母表是中文，英文表达式所属的字母表是英文，汉字不能应用在英文字母表中。如果用程序中的作用域来理解，字母表是表达式的作用域。比如Σ={x,y,z}，那么在此字母表上的表达式只能用 x、y 和 z 这 3 个字母组成，形式和长度不限，如 xxyyyyz、xyz、yyzx……

0.16 什么是语言

语言是由字母表、文法和语义组成的。

字母表是符号的有穷集合，由字母表中的符号组成单词、句子。文法用来制定单词和句子的种种规则，包括符号在单词中出现顺序的规则、单词在句子中出现顺序的规则等。语义是句子在文法规则下所表达的意图，也就是句子的意义，语义是基于单词和规则，特定的单词在特定的规则下才能表达特定的语义。比如代码 while(exp){循环体}，单词 while 后面必须接单词(，这是由一个字母组成的单词，再后面是循环表达式 exp，这就是“文法”。单词 while 的意思是只要表达式 exp 成立就一直循环执行后面的循环体，这就是“语义”。

后面还会有相关介绍。

0.17 正规式就是正则表达式

网上有人说正规式和正则表达式不是一回事儿，我觉得这么说有些欠妥。

正规式就是正则表达式，两者都是用于模式匹配，很多编译原理教材都有说明，如清华大学出版社出版的《编译原理（第 2 版）》（张素琴、吕映芝、蒋维杜、戴桂兰编著）第 52 页 4.2.2 节倒数第 6 行：正规式也称正则表达式。因此两者是一回事儿。

正规式是个规范、协议，而我们平时所用的正则表达式是这种规范协议的具体实现（这就像 C 语言有很多种编译器一样，C 语法就是规范，每一种编译器就是一种实现），只不过实现的功能更丰富，以至于无法和编译原理教材上所说的正规式对上号了，可以理解为：正规式是最原始的正则表达式。

0.18 什么是正规（表达）式和正规集

正规式也称正规表达式，就是表达“字符排列顺序”的公式。既然称为公式，那么正规式中肯定有运算符，没错，它正是用 3 个运算符来表达字符的排列顺序，如表 0-6 所列。

表 0-6

运算符	文字描述	读作	操作	优先级	示例
|	竖线	或	选择	低	a=x|y 表示 a=x 或 a=y
.	小圆点，可省略	连接	连接	中	x.y 表示 xy
*	星号	闭包	0 次或多次的自我连接	高	a=x*表示 a=ε， 或 a=x，或 a=xx， 或 a=xxx，或 a=xxxx…

以上的 ε 表示空，在此表示 0 次连接。多次的自我连接也称为方幂，就像数学中的幂一样，后面有介绍。

反过来说，由有限次使用这 3 种运算符得到的表达式就是正规式，假设正规式中的符号都属

于字母表 Σ，那么能用此正规式匹配的所有表达式组成的集合称为字母表 Σ 上的正规集。

概念为了严谨就难免会抽象，简单来说，正规集是正规式所能匹配的一切表达式的集合。举个例子，假设字母表 Σ={a,b}，对于正规式 ba*来说，其正规集是以 1 个 b 开头，后接 0 个或多个 a 的字符串。解释：*表示 0 次以上（包括 0 次）的重复连续出现（即连接），能被 ba*匹配到的表达式有 bε、ba、baa、baaa……等，这些就是 a*的正规集。那么对于正规式 aaab*来说，其正规集是以 3 个 a 开头，后接 0 个或多个 b 的字符串，同理不解释。

0.19 什么是有穷自动机

有穷自动机（Finite Automata）也称有穷状态自动机，是一种数学模型，也称有限状态自动机，这种模型对应的系统具有有限个数的内部状态，系统只需要根据当前所处的状态和面临的输入就能够决定系统的后继行为。当系统处理了输入后，状态也会发生改变。

有穷自动机是由两位神经物理学家 MeCuloch 和 Pitts 提出的概念模型，他们认为人脑也是个有穷状态的系统，尽管状态数目很大，但依然是有穷的。他们所提出的有穷自动机的模型是由 3 部分组成：一条有穷长度的输入带、一个读头和一个控制器，如图 0-10 所示。

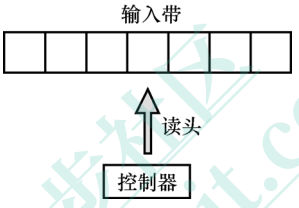


图 0-10

在这个模型中，单个输入称为输入符号，输入带用来存放输入串，每个输入符号占据一个方格，输入带的长度和输入串长度相同。控制器的状态数是有限的，读头只能只读输入带，控制器控制读头从左向右逐个读入每个符号，控制器根据当前状态和输入符号控制转移到的下个状态。

一个有穷自动机包括下列内容。

（1）一个有限的状态集合。

（2）一个有限的字母表。

（3）一个“状态+输入字符”的转移集合，以字母表中的符号作为输入字符。

（4）一个开始状态，也称初态。

（5）一个被称为“可接受状态的状态子集”，可接受状态也称为终态。

如图 0-11 所示。

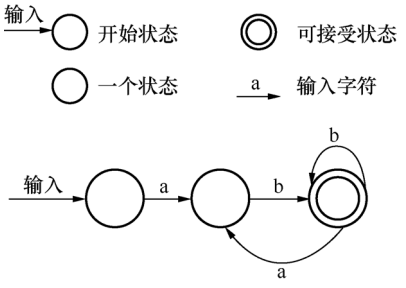


图 0-11

有穷自动机分为两大类：确定的有穷自动机（Deterministic Finite Automata）和不确定的有穷自动机（Nondeterministic Finite Automata），它们分别是常说的 DFA 和 NFA。

不再介绍，已经够用了。

0.20 有穷自动机与词法分析的关系

先回答，这两者间没直接关系。

编译原理教材中经常把词法分析器和有穷自动机一起介绍，以至于很多兄弟认为词法分析器就是有穷自动机，这样理解欠妥。

有穷自动机也称有限状态自动机，是一种识别装置，“可以”用来识别正规式，也就是说可以识别正规集。注意啦，说的是“可以”用来识别正规式，并不是只用来识别正规式，“可以”具有选项、可选的意思，这说明，词法分析器是把有穷自动机当成识别单词的工具，也就是说，有穷自动机只是词法分析器的一种实现。

计算机科学中可以找到很多有穷状态系统的例子，比如 TCP/IP 网络连接的状态：syn_sent、time_wait、close_wait 等，还有本文中的词法分析器，总之，有穷自动机是设计这些系统的理论工具，也可以说是一种算法。

词分析器为什么要采用有穷自动机来实现单词识别呢？

词法分析器是按照文法的规则来识别单词，文法是人设计的，同样，人在设计词法分析器时也要按照人自己制定的文法规则来识别单词，这是人与自己协调好的。比如文法规定整数是数字字符的正闭包，也就是由一个以上的数字字符连接而成的，浮点数是由整数内部加个小数点表示的，即“整数.整数”为浮点数。人也按照这个“思路”去实现识别，步骤如下。

（1）词法分析器启动后，若输入字符是数字，词法分析器就进入“处理整数的阶段”。

（2）当词法分析器准备处理整数时，如果下一个输入字符还是数字，继续处于“处理整数的阶段”。

（3）当词法分析器处理整数时，如果输入字符是小数点，词法分析器进入“处理浮点数的阶段”。

（4）当词法分析器处理整数时，如果输入字符不是数字或小数点，词法分析器处理整数完成。

（5）当词法分析器处理浮点数时，如果输入字符不是数字，词法分析器处理浮点数完成。

你看，我这里强调了“思路”，即表示先有的“想法”，然后再涉及如何“实现”。词法分析设计者在脑子中构思这种思路时，觉得“处理 xxx 的阶段”很符合有穷自动机的“状态”，输入字符的意义也和有穷自动机一致，“处理整数的阶段”进入“处理浮点数的阶段”完全符合有穷自动机的状态转移，很自然地，就用有穷自动机这个现成的理论工具来帮助实现词法分析器。有穷自动机是一种识别装置，把它应用在词法分析中，可以用来识别正规式所匹配的所有表达式，也就是说可以识别正规集。开始时自动机所处的状态是初态，当自动机从初态进入终态时，输入带上读过的字符串即是自动机所匹配的字符串，正好完成了词法分析的需要。因此，有穷自动机是被用来实现词法分析的算法，属于工具，无论是工具还是算法也都只是一种选择，方法并不是唯一的，因此也可以不采用它。你会看到在第 2 章实现的词法分析器，我们并没有使用有穷自动机。

0.21 词法分析用有穷自动机（有穷状态自动机）的弊端

有穷自动机也称有穷状态自动机，它是编译原理教材上实现词法分析器的经典方案，以下简

称状态机。

状态机的特点是，从当前状态接收一个字符后马上转移到下一个状态，即使下一个状态就是当前状态，转移的过程也要发生。比如图 0-12 用来识别数字的状态机。

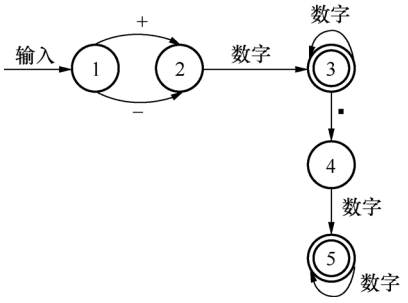


图 0-12

从输入开始，状态机进入状态 1，在读入正负号后状态机转移到状态 2，这是有符号数的开始。下一个接收的是数字，状态机就从状态 2 转移到了状态 3，状态 3 是双圆圈表示的终态，终态即终止态，也称可接受状态，表示状态机接受了所识别到的字符串。在状态 3 时接受任何一个数字都会使状态机重新转移到状态 3，即原地转移到当前状态，这是一个循环过程，故可接受无限个数字。根据此状态转移图，可接受的字符串形式为“+多个数字字符”或“−多个数字字符”，比如+1234 或−123456 等等。到此识别的是整数。如果下一个字符是小数点，那么状态机就从状态 3 进入到浮点数状态 4，继续读入数字，进入状态 5，状态 5 同样是终态，在此状态下可以接受无限个数字。

但实际上，现代词法分析器很少完全用状态机来识别单词，原因如下。

（1）效率低，需要重复进入状态分支。

状态转移的执行过程有冗余，因此效率低下。标准的状态机是每次读入 1 个字符后马上转移到某个状态，即使是状态原地不动也需要走一次这个转移状态的流程，多执行了一些不必要的操作。就拿上图右上角的状态 3 来说，如果严格地按照状态机的思路来编写程序，在读入 1 个数字后，下一步工作不是继续读入数字，而是把状态重复设置为状态 3（实现上也可不必，可理论上是这样的），然后进入状态的流程分支，再读入下一个字符，再继续判断接受此输入后的转移状态，尽管状态没变（还是状态 3），依然再重复上述过程。

（2）需要设置的状态太多，编码太累。

状态是由文法来决定的，文法复杂状态就会复杂，状态太多时编码会烦死的，就连单字符的单词（单词泛指文法中所能推导出的一切终结符，比如+也会被词法分析器视为单词）也要为其专门设置个状态，还不够麻烦的呢，而且这种单字符单词还特别多，比如（、）、{、} 和大部分单字符运算符等。也许仅凭我一面之词不足以让你信服，我当初也不相信，后来自己弄了一个有穷状态自动机版本的词法分析器，弄到一半我就信了。

因此我们的词法分析器也没用状态机，后面会讲实现部分。

0.22 什么是文法

如果一首曲子你不能哼唱给别人听，你该如何向别人描述这首曲子？答案就是用乐谱。同样，如果你设计了一门语言，如何向别人描述其用法？答案就是用文法。文法就是描述语言语法规则的方法。

文法即语言的语法，是由其产生式来描述的，产生式是语言的规则，因此在语法分析时，就是按照文法产生式去匹配所输入的代码。

“上下文”是指所处的环境。一个上下文无关文法是指文法所定义的语法范畴与上下文无关，完全独立于语法范畴所处的环境，简单来说就是“就事论事”，不必考虑人情世故。比如数学运算就是上下文无关。上下文有关就是指语义要取决于语（环）境。比如老妈说了句“几点了”，这句话在早上说的意思有可能是“还不起床，你想睡到什么时候”；而在晚上说的意思可能是“这么晚了你老爸怎么还不回来”。除非程序员要求代码的意义必须要结合上下文环境才能确定，否则编译器采用上下文无关文法就足以应付目前的编程需求了。上下文无关文法所对应的语言称为上下文无关语言，无特别指出的情况下，本书中所涉及的文法均指这种上下文无关文法。

文法是由一个四元组表示的，元组就是用小括号括起的元素对儿，四元组就是 4 个用括号括起的元素。文法本身是用字母 G 表示，文法内容用一个四元组表示，包括非终结符 Vn，终结符 Vt，产生式 P，开始符号 S，即 G=(Vn, Vt, P, S)。文法是一套有限的规则，但是它可以创建无限符合此规则的句子，这就是产生式的作用，有关产生式请看下回分解。

0.23 BNF 和 EBNF，非终结符和终结符，开始符号及产生式

这都是辅助解释文法的概念。

BNF（Backus-Naur Form），即巴科斯范式，是一种程序设计语言描述工具，是由 John Backus 和 Peter Naur 引入的一种形式化符号，用来描述给定语言的语法，即描述语言的语言，故可以称之为元语言。它的内容如表 0-7 所列。

表 0-7

元符号	含义	备注
->	定义为	也可用:=和::=表示
<>	用<>括起表示非终结符	通常用大写字符代替
“”	双引号中的内容表示终结符	通常用小写字符代替
|	或	用于候选式

由于它过于简单，所以在实际应用中暴露了不足。

（1）循环和可选项不能直接表示。

比如若想表示多个'a'循环，就要写成这样：P->aP。

若想表示可选项（即可以有也可以没有的项），就要写成 P->a|ε。其中 ε 表示空，即没有。

（2）不利于编写语法分析器。

下面为解释缘由会提及一些未介绍的概念，后面有详细介绍，不懂的读者不要急。

由于语法分析器的实现是要根据产生式来编写，所以用 BNF 写出的产生式也不利于程序实现，比如下面描述 if-else 语句的产生式：

IF_STAT -> if (EXP) {STAT} | if (EXP) {STAT} else {STAT}

语法分析器在执行非终结符 IF_STAT 对应的函数时，发现这两个候选式前面的“if (EXP) {STAT}”部分都一样，必需二选一，因此如果“死板地”按照此产生式来分析语法，必然要冒险读入大量的字符串，如果候选项判断错了，就要回溯，回溯就是回退到之前的分叉口进行重新选择，这代价太大了，因此根据这样的产生式来编写语法分析器很不容易。因此有了 EBNF，即扩展 BNF，内容如表 0-8 所列。

表 0-8

元符号	含义	备注
->	定义为	也可用:=和::=表示
<>	用<>括起表示非终结符	通常用大写字符代替
“”	双引号中的内容表示终结符	通常用小写字符代替
|	或	用于候选式
{}	0 次或多次重复出现	
[]	0 次或 1 次重复出现，即有或没有	
()	括号内的看成一项	

有了 EBNF，之前的循环、可选项及语法分析器的编写就容易了。

循环可写为：

P->a{a}

可选项可写为：

P->[a]

语法分析器所依赖的产生式可写为：

IF_STAT -> if (EXP) {STAT} [else {STAT}]

这样分析器在处理 if (EXP) {STAT}后，只要再从用户代码中判断是否有关键字 else 就可以了，只需要读取 1 个记法单元就可以判断了。

也许有读者会说：“对于以上的 IF_STAT 来说，无论是用 BNF 还是 EBNF 描述，其所描述的语法结构都是一样的，只不过看上去 EBNF 更方便，而 BNF 笨拙一点。我按照 if-else 的语法结构来识别也是一样，因此用 BNF 文法也行，只要我不死板地按照它的描述去编写语法分析器就行了”。道理是这样的，但你想想看，文法的目的是什么？就是用来向人们描述语法规则的，如果不看这个 IF_STAT 的文法，你是怎么知道关键字 if 后面要接左小括号“(”，甚至还可以接一个 else？还不是看了产生式 IF_STAT 后才知道的吗。而且，文法就是编写语法分析器的导航，按照导航去实现语法分析器肯定是最省事儿的。这么说吧，尽管演奏家已经把曲子背下来了，但有个乐谱摆在那，是不是演奏起来更得心应手。所以，文法是为了语法分析器服务的，让语法分析器的设计者更舒服，那才是最好的，否则非要自作聪明跨过文法，按照自己的理解去实现语法分析器，那还要文法干吗呢。

文法是利用“有限”的规则，通过“推导”的方式产生“无限”多的句子来描述某种语言所有合法的使用情况，这里的推导加了引号，非终结符和终结符便是针对这个“推导”来定义的。推导是推出、导出的意思，即用另一种与非终结符等价的表达式替换非终结符，将此非终结符“展开”。终结符是指“推导过程”终止、终结后的产物，也就是推导到头儿了，符号没法再展开了。非终结符是指推导过程还要继续，待推导的符号还没展开彻底，下面再细说说。

终结符通常用 Vt 表示，即 Vocabulary terminated，也就是推导“终结”的意思，这也正是终结符的意义，因此终结符相当于“常量”，通常为字面量，是某个具体、固定的值，比如“(”，终结符用来作为非终结符的值。

非终结符通常用 Vn 表示，即 Vocabulary not terminated。顾名思义，不终结的符号，是指推导可继续进行、还没有推导彻底，它就像变量，变量的值还可以用变量（其他变量或者自身）来表示，那么非终结符也可以用非终结符（其他非终结符或者自身）来表示，还可以用终结符来表示。

无特别指出，本书中终结符用小写字母表示，非终结符用大写字母表示。

语法分析时总该有个语法规则的起始入口，这个入口就是开始符号。开始符号通常用 S 表示，即 Start。它是文法中最大的非终结符，它可以推导出任何其他非终结符及终结符，因此是文法开始推导的入口，就像 DNS 的根域，从最上层开始总能到达任意下层。

文法中的规则称为产生式，即用来“产生”句子的表达式。通常用 P 表示，即 Production，比如规则 E->E+T 就是产生式，->表示定义为。->左边的部分称为产生式的左部，->右边的部分称为产生式的右部。非终结符就是用来“推导”的，就是用其右部等价的表达式替换，将非终结符“展开”为右部。推导是用=>表示，即可推出、可导出的意思，1 步推导也称为直接推导，用符号=>表示，1 步以上推导用 $\overset{+}{\Rightarrow}$ 表示，0 步以上推导用 $\overset{*}{\Rightarrow}$ 表示。如果一个非终结符可直接同时推导出多个表达式，这几个表达式之间属于并列的关系，这几个并列的表达式便称为该终结符的候选式，只有“并列”才能“同一级别”嘛，这样才有资格成为“候选”。也就是说，在产生式右部，若有多个用|分隔的表示“或”的并列的表达式，这几个表达式称为左部的候选式。候选嘛，随时待命等候被选择，如同候选人一样，若是只有 1 个肯定就没得选了，因此多于 1 个可供推导的表达式才称得上“候选”。比如 E->E+T|F。E+T 和 F 都是 E 的候选式。

为什么称为“产生式”呢？这是因为它是“产生”语法的式子。根据某一语法规则（即文法）我们可以写出无限多的代码，比如以下文法 1，如图 0-13 所示。

```
句子  ->  主语 谓语
主语  ->  名词 | 代词
谓语  ->  动词 [宾语]
宾语  ->  名词 | 代词
```

文法 1

图 0-13

|表示或的意思，表示选择其一，|两边的部分称为候选式。[]表示中括号内的部分可出现 0 次或 1 次，即有或没有。按照此方法我们可以写出很多的句子，比如“我走了”“我吃早餐”“我笑了”“我去游泳”，总之可以“产生”无限多的句子，这就是称为“产生”式的原因。

这里补充一下，在实际生产环境中为方便编码会采用“正规文法”来定义语法，正规文法简单来说就是利用正则表达式定义的文法，产生式全是用正则表达式来描述的，咱们在后面也会采用正规文法，因此大伙儿最好有正则表达式的基础。

0.24 什么是句型、句子、短语

从一个文法的开始符号 S 可推导出的任意符号串 a 都称为句型，即 S $\overset{*}{\Rightarrow}$ a，当 a 中全是终结符时，a 就称为句子。

既然称之为句型，就应该按照句型的意义去理解。在英文里，句型是个句子的模板，是可以套用的句式，所以在编译原理中的句型也应具有模板的意义，这里的模板变量就是指非终结符。当句型中没有可替换的部分时，即非终结符全部展开为终结符了，也就成了最终的句子。句子是特殊的句型，相当于句型的实例化。

比如在文法 1 中，“句子 -> 主语 谓语”，其中的“句子”便是开始符号，非终结符“主语”的产生式是“主语 -> 名词 | 代词”，主语有两个候选式，这里选择用非终结符“名词”表示主语，故“句子 -> 主语 谓语”则变为“句子 -> 名词 谓语”，这就是句型。而“我去游泳”便是句子，因为这是把所有非终结符都展开后的结果，里面全是终结符了。

什么是短语呢？先对比一下，句子和句型都是“整”句，这里强调的是完整一句，那么自然

而然，短语就是指句子或句型的一部分，如同英语中的短语是一样的概念，都是句子的组成部分且不能单独存在。因此短语是这样定义的：在文法 G 中，S 是开始符号，假设 $S \overset{*}{\Rightarrow} \alpha A \delta$ 且 $A \overset{+}{\Rightarrow} \beta$，则称 β 是句型 αβδ 相对于非终结符 A 的短语。如果 A=>β，即 A 直接推导出 β，则称 β 是句型 αβδ 相对于产生式 A->β 的直接短语。举个例子，图 0-14 文法中的 3 个产生式。

```
“句子　->　主语 谓语”
“主语　->　名词 | 代词”
“名词　->　小明”
```

图 0-14

主语可直接推导出“名词”，那么“名词”就是句型“句子　->　名词 谓语”相对于非终结符“主语”（即产生式“主语　->　名词 | 代词”）的直接短语。强调下，短语是句型或句子的一部分，并不是完整的句子或句型，比如这里的短语是“名词”，要不干吗称为“短”语呢。“名词”其实也是非终结符，可以指代人名、物名等等，这里仅用人名“小明”作为产生式“名词”的右部，而“小明”也就是终结符了，没法再推导了。短语也可以是终结符，比如终结符“小明”是句型“句子 -> 小明 谓语”相对于非终结符“名词”的短语。

注意一，直接短语一定是非终结符的完整右部，如果右部中有多个候选式时就是其中的一个完整的候选式，这一点很重要，尤其是在句柄中，短语就是用来引出并解释句柄的概念，后面会介绍句柄。

注意二，文法就是用来创建句子的，对于语言（包括人类语言和计算机语言）来说句子才是唯一有用的部分，因为文法中的产生式和句型只是构建句子的规则、规律，这样就可以用有限的表达式来构造出无限的句子。总之方法并不是为了创建句型，句型是无用的，句型只是句子的模板，模板的实例化才有用，就像类和对象的关系，类是对象的模板，我们定义了类，目的是为了生成实例对象。人类语言也是一样，人们口中说出来的都是句子，不是句型，句子才能交流，句子才是最具体实在有用的，句型只是个句子“框架”。我们写的源码全是句子，语法分析也都是在分析句子，而句型只是语法分析的中间态，句型是向人们描述：符号应该怎样排列才是正确的句子。

0.25 什么是语法分析

语法分析就是按文法的产生式去识别输入串是否为文法的一个句子，这个输入串就是用户的源码，即语法正确的用户源码就是该文法的句子。

开始符号是文法中最大的非终结符，是文法定义的入口，因此是文法的最“上”层。文法是利用有限的语法规则来创建无限多的句子（源码串），句子是全部由终结符组成的字符串，是由上层的非终结符推导出来的、符合语法的最具体的部分，也就是没法再推导了，是最“下”层，我们平时所写的代码就是最下层的句子。你体会到了吗，语言的语法就那么几条，但我们可以写出千变万化无限多的代码，这就是文法规则的“以有限为无限”。故，语法分析从大体上有两种识别方式。

（1）自上而下的推导方式

自“上”而下语法分析是从文法的开始符号开始推导，直到最“下”层的句子，因此得名。从方法的开始符号往下推导，检查是否可以推导到实际的代码，如果可以，说明所输入的代码语法正确。比如一个树干延伸了多个树枝，每个树枝又长出多片树叶，推导便是从树干到树叶的过程。

（2）自下而上的归约方式

以最“下”层的源码串（即文法中的句子）往“上”层的非终结符“归约”，因此得名。归约

是推导的逆过程，推导是把非终结符（产生式的左部）按照产生式的右部“展开”为“终结符或其他非终结符”，归约是把“终结符或其他非终结符”按照同样产生式的右部“收敛”为非终结符（即产生式的左部）。归约是从最“下”层源码串为语法分析入口，检查是否可归约为某一产生式，产生式就是文法的规则，也就是判断代码是否符合语法，继续向上归约，直至开始符号。归约就是从树叶到树干的过程。

由于自上而下的推导方式比较容易用程序实现，大部分编译器都是采用此方式，这里就以它为主介绍语法分析。

自上而下推导就是检查从文法的开始符号（最大的非终结符）是否可推导出这个用户的代码，语法正确的话用户代码就是文法的句子，注意这里说的是句子，并不是句型，必须是把产生式中所有的非终结符推导到终结符为止。“语法正确的用户代码就是文法中的句子”这么说想想也很合理，你看，我们在给同事解释自己所负责模块的代码时，一般都会指着屏幕上的某行代码说：这“句”话的作用是啥啥啥。

匹配的过程就是非终结符推导的过程，如果源码语法正确，即源码是文法的一个句子的话，那么肯定能从文法的开始符号推导出用户的这句代码。举个例子，比如语法规则是前面所述的“文法 1”，如果句子是“小明踢球”，主语=>名词=>小明，谓语=>踢，宾语=>球，符合文法 1 中的规则。如果是源码是“小明和球”，这个“和”是连词，不符合文法 1 中的规则，因此就不符合语法，这就是语法分析的工作。当然，“山爬小明”虽然有语病，但还是符合语法规则的：主语是山，符合名词或代词，谓语是爬，符合动词，宾语是小明，符合名词或代码。估计有很多读者误以为判断句子是否有意义是语义分析的工作，这一定要纠正，这可不关语义分析的事儿。因为此类错误通常属于程序逻辑的问题，逻辑上的错误编译器可不好检查，比如某程序员误以为成年后就可以结婚了，而 18 岁就是成年，因此他在程序中判断男人必须大于等于 18 岁才可以结婚，但实际上我国婚姻法规定男人的法定结婚年龄是 22 岁，那么此程序员的这句大于等于 18 岁的代码的“意义”就错了，这种编程人员的逻辑错误难道也要让编译器检查出来吗，如果那样的话，编译器还得懂法律，那必须得是百科全书才行。显然这是不可能的。编译器制定了语法规则，只要用户遵守规则，怎么运作都行，如果不遵守规则，编译器就报错罢工，这已经是尽责了。

0.26 语法分析中的推导和归约为什么都要最“左”

其实这个很简单，只是编译原理比较难懂，而且很多读者都没实践过语法分析器，再加上编译原理书上很少有提到过原因（也许那些作者觉得很简单，没必要提），因此部分读者以为“最左”是规定。规定也是有理由的，肯定是为了规避某些问题而约定的，不能胡乱规定，通过下面的例子你也许自己能看出来为什么要最“左”。

自“上”而下语法分析是从文法的开始符号开始推导，通常它采用的是最左推导。最左推导是指每次都先推导最左边的非终结符，然后再推导右边的非终结符。比如如图 0-15 所示产生式。

```
S -> AB
A -> C
C -> c
B -> b
```

图 0-15

最左推导的过程如图 0-16 所示。

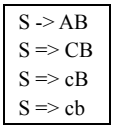


图 0-16

自“下”而上的语法分析是从句子往上归约为某一产生式，通常是最左归约，即每次都先归约最左边的，被归约的可以是终结符、非终结符或两者的组合。比如用户源码是 cb，那么先将 c 按照产生式 C -> c 归约为非终结符 C，输入串就为 Cb，再将 C 按照产生式 A -> C 归约为非终结符 A，即 Ab。最左边的非终结符 c 处理完了，再处理右边的 b，根据规则 B -> b 归约为 B，即成为 AB，再根据规则 S -> AB，最终得到开始符号 S，说明语法正确。

两种语法分析都是最先处理“左”边的，介绍上面的两个例子下面说明下最“左”的理由，其实原因好理解，估计读者也想到了，就是因为用户的源码同写字的顺序一样都是从左往右写的，用户的代码逻辑肯定是从左往右读才能被正确理解，因此肯定要先处理最左边的字符才能正确理解用户的意思，因此最左边的字符要最先进来才对。

0.27 什么是语义分析

越是重要的东西就越是不容易察觉到其存在，当它成为我们必不可少的依赖时我们就容易忽略它，因为它已与我们融为一体，能感觉到的皆是身外之物。语义分析是整个编译过程中最重要的部分，以至于我们都不知道它在哪里。

语义分析混迹于语法分析和代码生成之中，功能并不像词法分析、语法分析那么独立、明显，因此不容易搞清楚它的作用。语义分析的工作可“不”是分析句子是否有意义、是否符合逻辑、通顺成句等等，语义分析的工作是识别出源码的功能逻辑，即搞清楚源码的意义、功能、意图是什么。搞清楚之后，就要按照这个意义生成目标代码了，因此语义分析是为“生成代码”服务的。这和翻译人员的工作是一样的，比如把中文翻译为英文，翻译人员听到中文后，自己先在脑子中体会、消化这段中文，然后按照他自己的“理解”翻译成对等意义的英文，这个“理解”中文的过程就是语义分析，你看，翻译人员这个“语义分析”没人看得到，人们只看到了他在两种语言之间切换输出。同样，编译的目的就是把源码编译为目标代码（或中间代码，本章有关此内容统称为目标代码），必须要搞清楚源码的意义后才知道怎么把它翻（编）译为对等意义的目标代码，因此它通常与语法制导一同工作（语法制导的任务就是翻译成目标代码），在理解源码之后马上输出对等的目标代码，因此人们只注意到了编译后的“实实在在”的目标代码，忽略掉了语义分析的存在，就像人们只注意到了翻译人员翻译后的英文一样，其实那个“理解”原文的过程才是整个翻译工作的重中之重。

既然语义分析是为生成代码服务的，因此很多书上都会把语义分析和生成代码放在一起介绍，为了更形象一点，看个例子。比如有一句这样的代码 *a*=*b*+2，进行语义分析后，知道了此行源码的意图是把 *b*+2 的结果写入变量 *a*。然后开始为其生成目标代码，假如目标代码是汇编语言，那么就要生成：

```
mov tmp, 2
add tmp, b
mov a, tmp
```

即理解完了源码的语义后马上就生成对等的目标代码，这是一气呵成的。

好啦，本小节结束。

0.28 什么是语法制导

生成中间代码或目标代码不像记法分析和语法分析那样有法可依，可以很容易地量化到具体的实现，它没有固定的方法，因此这也是编译领域中较难的部分。而语法制导是生成中间代码或者目标代码的一种方式，仅是个概念，并不是具体的方法，它是指在语法分析的过程中同时完成绑定在产生式上语义规则的“动作”，这个动作便是指“生成代码”这个动作。也就是说，语法制导是指在语法分析的同时生成代码（中间代码或目标代码），从而完成编译。

编译的目的就是为了生成源程序对应的中间代码或目标代码，生成的代码肯定要能表达出源程序的功能逻辑才行，也就是说必须要如实表达出源程序的需求，因此生成代码的顺序要与源程序的功能逻辑的顺序相一致。如何保证逻辑一致？很简单，只要按照源程序的功能逻辑去分析代码就行了，这如何做到呢？还是很简单，因为语法肯定是不变的，按照语法去分析代码就行了，这个语法其实就是分析源码的“导航”，源码的功能逻辑肯定是遵循语法来实现的，因此按照“语法”去分析源码便识别出了源码的功能逻辑。一些传统的编译器会把此功能逻辑用树来记录，这个就是抽象语法树，即 AST，比如表达式 3+2 生成的语法树就是以+为根，以 3 和 2 分别为左、右子树的二叉树。也就是说，语法分析便是生成抽象语法树的过程，抽象语法树便是源码功能逻辑的本质表达，这就是“抽象”的意义所在——去掉不必要的部分，取其本质。当然，现在很少有编译器会生成抽象语法树了，尤其是对于一“遍”的编译器，原因是抽象语法树是在严格按照以语法为导向的分析过程中产生的，这个分析过程本身就是识别源码功能逻辑的过程，既然已经知道了源码的意图，那直接生成指令就行了，何必再额外生成个语法树呢，因此生成抽象语法树是完全没有必要的，即使对于某些多遍的编译器，抽象语法树也不是必要的，这一切都取决于实现。

啰唆之后你明白了，这个“导”是指“导向”，即生成的代码要以语法分析为导向，因此称为制导。

0.29 词法分析器吃的是 lex，挤出来的是 token

词法分析器的功能是将源码中的单词封装为 token，单词称为 lex（Lexeme），这里的“单词”不仅是英文单词，泛指文法中所能推导出的一切终结符，比如左小括号“(”也是单词，只不过是单个字符组成的单词。小节标题中的“吃”指的是输入，“挤”指的是输出。严谨地说，词法分析器“吃”的是长长的源码文本字符串，并不是 lex，因为 lex 是它从长长的源码串中解析出来的一小部分，这已是词法分析器处理文本之后的效果，因此并不是真正的输入。写成这样的标题是想让大伙儿注意到 lex 和 token 不是一回事儿，并且词法分析器在两者间起到承上启下的作用。

在最初的编译器中是没有词法分析器的，起初是由语法分析器代劳，其过程是：语法分析器从文件中获取单词，然后同已知的关键字比较，和哪个关键字匹配上了就表示该单词就是哪个关键字，否则就是用户自定义的标识符，这就要校验标识符的合法性，比如标识符不能以数字开头等等。同时还要记录该单词所在的行号，以备源码中若有语法错误就提示错误的地方是在哪行，甚至有时候需要知道上一个单词的相关信息等。你看，语法分析时需要“校验”单词和“收集”很多与单词相关的周边信息，而这个工作逻辑上其实是独立于语法分析的，那么语法分析器就干脆把这个工作拆分出来，这个拆出来的部分便是词法分析器。称为“器”之后显得有些“深不可测”了，其实在软件中这都是函数，只不过是把之前的工作封装为一个函数，语法分析器每次需要单词的相关信息时就调用这个函数（也就是词法分析器）。由于单词的相关信息还是挺多的，不

是一个简单的变量能存得下的，因此要用一个复合结构来存储，这个复合结构就是 token。

token 存在的另外一个原因是，它给了语法分析器一个“结果”，因此让语法分析更方便。比如若没有 token 的话，词法分析器把识别到的单词字符串（即 lex）返回给语法分析器，那么语法分析器如何知道这个单词是什么？肯定不知道，还是没有“结果”，还得和词法分析器一样再比较一次字符串，那么这个工作就重复了。

总结，词法分析器的工作就是，收集源码中各个单词的一系列周边信息，并把这些信息封装为 token 给语法分析器使用。

0.30 什么是“遍”

“遍”可以理解为编译器从头到尾扫描源码的次数，往往是次数越低编译器越高效，因此也能用来衡量编译器的性能。尽管有很多一遍的编译器，但多遍编译器也不少见，比如第一遍先收集所定义的全部变量，第二遍再判断所引用的变量是否已定义。或者为函数分配运行时栈的空间时，需要得知局部变量和参数的个数才能确定运行时栈的大小，那么第一遍先统计这些，第二遍时再确定分配栈空间的大小。

0.31 文法为什么可以变换

文法是语言的语法规则，因此可能给我们的“幻觉”是：“一种语言对应一种文法，文法变了语法规则就变了，就变成了另一门语言了，这不科学”。其实这是理解错误，应该是：一种文法对应一种语言，一种语言可以对应多种文法，如果多种文法描述的是同一种语言，那么就称这几种文法等价，文法变换就是把文法转换为另一种等价的文法，文法变换的基础就是文法等价。

举个例子，比如某个句子是奇数个字符 c，如 c、ccc、ccccc……它可以由文法

G1 = ({S}, {c}, {S->cSc, S->c}, S)

生成，也可以由文法

G2 = (S, A), {c}, {S->cA, S->c, A->cS}, S)

生成，也就是说文法 G1 和 G2 是等价的。现实生活中也有规则等价的例子，比如拿蒸馒头来说，和面时先放水还是先放面，蒸出的馒头都一样。

为什么要变换文法？这是为方便语法分析。因为语法分析就是按照语法规则去匹配用户的源码串，由于用户代码是自左向右写的，因此从左到右地处理用户代码串才是合理的，所以文法产生式右部中非终结符和终结符的位置顺序也应该符合源码从左到右的顺序。语法分析以产生式为导航，但是有时候会因语法规则本身设计的问题导致语法分析工作走入死胡同或者前进的方向不明确，这时候就需要变换一下规则以使语法分析可继续进行下去。有哪些情况会扰乱语法分析呢？请见“为什么消除左递归和提取左因子”小节。

0.32 为什么消除左递归和提取左因子

前面说过了，语法分析有两种方式，一种是自上而下的推导，另外一种是自下而上的归约。无论哪种方式，由于用户的代码是以从左到右的顺序书写的，只有从左到右地处理用户代码才能理解用户代码的逻辑，因此这两种语法分析方式都要优先处理用户源码中最左边的字符。那么语法分析是如何保证最左边的源码串先被处理呢，这就是产生式的作用了。我们说过，产生式就是语法分析的导航，确切地说是产生式右部中符号（非终结符和终结符）的位置顺序才是语法分析

的导航。这个顺序就是语法规则，语法的设计者知道用户习惯是以从左往右的顺序写字，他自己也不例外，写代码也是写字，因此也是按照从左到右的顺序来编制语法。语法是什么？就是一堆规则，规则是由产生式来体现的，确切地说是产生式右部，因此就按照从左到右的顺序来写产生式右部中的符号，这样就符合了用户源码的顺序。

用户得遵守此规则才能使用此语言，语法分析会读取这个规则，根据规则便知道用户源码串应该是什么，总之规则使源码串可预见，实现了最左边的源码串优先被处理。那么问题来了，产生式右部中最左边的符号可能是终结符，也可能是非终结符，如果是后者的话，该非终结符可能与产生式左部的非终结符同名，比如这种产生式：

$$P \overset{+}{\Rightarrow} Pa$$

右部中最“左”边的符号也是非终结符 P，在自上而下分析中，这样的推导会导致死循环，P 永远用 Pa 代替，无限“递归”下去，这就是左递归，即产生式右部中最左边的符号和产生式左部符号一样。最终推导的展开式为：

$$P \overset{+}{\Rightarrow} Paaaaaaaaaa\cdots\cdots$$

语法分析走不出来了。因此对于自上而下分析来说，必须要消除左递归。

消除左递归的方式通常是引用一新的非终结符，把左递归变成右递归，右递归就是产生式右部中最右边的符号和产生式左部符号一样。转换的公式我就不列出了，大伙儿有兴趣去查编译原理的书籍吧。

下面介绍提取左因子的必要性。假设非终结符用大写字母表示，我们知道，产生式的右部中可以包括多个以'|'分隔的候选式，比如这个表示 if 的非终结符的产生式：

```
IF_STAT -> if (EXP) {STAT} | if (EXP) {STAT} else {STAT}
```

这里说一下，自顶向下语法分析中，每个非终结符都对应一个函数，也称子例程，当源码串为 if(a > 0) {c++}时，语法分析器会执行非终结符 IF_STAT 的规则，即调用 IF_STAT 对应的子例程，该子例程应该选择哪个候选式呢？因为候选式前面的部分是一样的，都是 if (EXP) {STAT}，也就是说对于有公共前缀的候选式会让语法分析器失去方向很迷茫。因此为了更加明确地选择候选式，需要把左边的公共因子提取出来，就像数学中的提取公因式一样，此产生式改成：

```
IF_STAT -> if (EXP) {STAT} (ε| else {STAT})
```

其中的 ε 表示空，就是没有。或者用 EBNF 表示成：

```
IF_STAT -> if (EXP) {STAT} [else {STAT}]
```

其中的[]表示出现 0 次或 1 次，即有或没有。

话说回来了，如果各候选式没有公共前缀不就没这些问题了吗，如何判断是否有公共前缀呢？请见下节。

0.33 FIRST 集、FOLLOW 集、LL(1)文法

文法是人为设计出来的，为避免给语法分析制造麻烦，应该在设计之初就考虑规避一些问题。因此做文法设计的人最好做过语法分析，否则不容易清楚语法分析的需求。

之前说过提取左因子的原因是各候选式有公共的前缀，如果没有公共前缀就不用再提取啦，直接进行语法分析即可。如何确保没有公共前缀呢？你看，无论公共前缀多长，都是从第 1 个字符开始的，只要第 1 个字符不一样，就无法构造公共前缀啦。估计大伙儿都有答案了：只要保证产生式中各个候选式的首终结符的首字符不一样就一定没有公共前缀。假设产生式 A->a1|b2|c3，用公式表示则为：

$$FIRST(a1) \cap FIRST(b2) \cap FIRST(c3) = \varnothing$$

FIRST 表示终结符首字符集，上面的公式表示各候选式的 FIRST 集交集为空（∅表示空集）。注意，尽管这里的参数是终结符，但不要以为只能处理终结符，终结符首字符集是个集合，给 FIRST 的参数是什么，逻辑上它就返回相应终结符的首字符集。即如果参数是终结符，它就返回该终结符的首字符，如果参数是非终结符，它就自动进行推导，直到遇到终结符，然后返回终结符的首字符。切记，FIRST 求的是终结符首字符“集”，如 FIRST(a1)得到的只是该终结符的首字符 a，因此肯定不成“集”。FIRST 的参数还可以是非终结符，通过推导，该非终结符总会到达某一终结符，因此 FIRST 集最终还是求的终结符的首字符。比如 FIRST(A)={a,b,c}。

FIRST 集的提出是为避免用户源码可匹配产生式中“多个”候选式的一种辅助手段，只要各个候选式的 FIRST 集交集为空，语法分析器就不会面临如何选择的问题。倘若用户源码不属于任意一个候选式呢？难道就报语法错误吗？为时过早，在某种情况下，也许语法也是正确的。比如非终结符 A 的某个候选式为 ε（空），也就是说 A 可以用 ε 代替，要知道，非终结符也可以成为产生式的右部，在该右部中，碰巧非终结符 A 后面的终结符 b 可以匹配用户源码，而非终结符 A 恰好可以推导出 ε，也就是说相当于可以把非终结符 A 从产生式右部中去掉，那么终结符 b 就被提到原来非终结符 A 的位置，那么就正好满足用户代码的匹配，在这种情况语法就是正确的，这就引出了 FOLLOW 集。

FOLLOW 集称为后继终结符集，注意，并不是后继终结符首字符集。FOLLOW(A)是指出现在非终结符 A 之后的终结符或#，#表示输入结束符，类似文件结束符 EOF，表示源码都处理完了，#是为语法分析方便而人为加上的符号，产生式中并不存在。尽管 FOLLOW 集是为解决非终结符匹配空候选项后能用其后的终结符匹配用户源码而提出的，但这并不表示参数必须为非终结符，只是应用场景全是求非终结符后面的终结符，所以我们看到的参数才是非终结符。参数后面的终结符要么是推导出的，要么就是紧跟在参数后面的，如图 0-17 所示两个文法 FOLLOW(A)的结果都是{a}。

文法 1	S -> AB A -> ε B -> a
文法 2	S -> Aa A -> ε

图 0-17

另外，尽管前面说了非终结符 A 有个 ε 候选式，然后非终结符 A 后面的终结符与用户源码匹配，但这与 FOLLOW 集的定义无关，提出那个条件是为了引出下面 LL(1)文法的定义：

（1）文法中不含左递归；

（2）文法中所有非终结符的各候选式 a 的 FIRST(a)互不相交；

（3）文法中的每个非终结符 A，若它存在一个 ε 候选式，那么 FIRST(A)和 FOLLOW(A)交集为空。

满足以上 3 条的文法即是 LL(1)文法，其中的第一个 L 表示从左到右扫描源码串，第二个 L 表示最左推导（即每次推导产生式右部中最左边的非终结符），1 表示分析时的每一步只需要向前查看 1 个符号。提醒，“推导”就意味着自上而下，归约才意味着自下而上，后面会介绍。因此 LL(1)属于自上而下递归下降分析法，为什么是“下降”呢？因为在非终结符对应的子例程中，每次递归调用的都是该非终结符所属产生式的右部中的非终结符对应的子例程，相当于调用了自己的组成部分对应的子例程。

LL(1)是最简单易处理的文法，它是这样工作的。

（1）若输入串匹配到了非终结符的一个候选式，就执行该候选式对应的例程。

（2）若输入串不属于非终结符的任意一个候选式：若该非终结符存在一个 ε 候选式，且该非终结符的 FOLLOW 集匹配输入串，就让非终结符自动匹配 ε。否则就报语法错误。

0.34 最右推导、最左归约、句柄

既然是推导，就说明语法分析是自上而下，我们在前面介绍自上而下语法分析时都说了是最左推导，那么这个最右推导是干吗的？最右推导是为了解释最左归约。

前面说了，归约是推导的逆过程，而最左归约是最右推导的逆过程。走迷宫大伙儿都玩过吧，请教你个问题，你是怎么找到走出迷宫的路线的呢？有经验的你肯定是从出口往入口找，因为出口的路线较明确，就一条，不像在入口那样面对多条路，走哪一条路需要靠“撞大运”，发现走不通了再回来，这一现象在算法中称为回溯，即重新回到出发点再重新选路。回溯是最麻烦最讨厌的事情了，之前浪费的时间不说，回溯到之前的状态就意味着恢复到之前的环境，这个相对麻烦一些，因此减少回溯是提高效率的关键。那进一步讲，无回溯岂不是更好，如何保证无回溯？文法中的产生式有很多，在归约过程中也面临从众多路线中选路的问题，当有多个符合的产生式时该如何做出选择？假设源码的语法是正确的，如何确保源码能归约到开始符号以证明源码语法正确？思考一下迷宫的例子，从出口走到入口比较容易，容易确保迷宫走得通。我们知道一个句子可以从开始符号经过多步推导得到，因此该句子只要按照推导的逆序就能完美归约到开始符号啦，即怎么来的就怎么回去。之前说过了最左归约的原因，受迷宫启发，归约成功的前提是先保证从开始符号能推导成功，与最左归约相逆的便是最右推导。

顾名思义，最右推导就是在推导过程中每次都先推导最右边的非终结符，用户源码是从左到右写的，源码最后的部分肯定是最右边，因此最右推导的结果是用户源码中最先写入的部分（即源码最左边）是最后得到的，那么对于其逆过程来说就是最先处理的部分，也确实就应该这么做，前面说过了用户源码最左边的应该最先被处理，这才符合逻辑。最右推导的逆过程正是最左归约，完美契合。举个最右推导的例子，文法如图 0-18 所示。

```
S -> eAcBa
A -> Ab | b
B -> d
```

图 0-18

最右推导过程如图 0-19 所示。

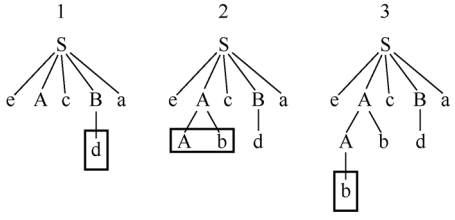


图 0-19

你看，以上 3 个推导过程中，都是根据某一产生式推导非终结符，产生式的右部我都用方框框起来了，它们是以非终结符为左部的产生式的“完整”右部，一定要记得“完整”这个特性。之前介绍过了短语的概念，方框中的就是短语，而且是直接短语。如在图 0-19 的 1 中，d 是句型

eAcda 相对于 B->d 的“直接”短语，它是由最“右”推导 B 时得到的。图 0-19 的 2 中，Ab 是句型 eAbcda 相对于 A->Ab 的“直接”短语，它是接上步的 eAcda 经最“右”推导 A 时得到的。当然也可以直接把 A 推导为 b，那就没后面的 3 什么事儿了。图 0-19 的 3 中，b 是句型 ebbcda（此时已经全是终结符，变成句子了，句子是特殊的句型）相对于 A->b 的“直接”短语，它也是上一步的 eAbcda 经最“右”推导 A 时得到的。把以上 3 个过程反过来看，各自方框里的短语都可以归约到上一步，即 3 变成 2，2 变成 1。因为方框中的短语是刚刚由上一步直接推导出来的，因此方框中的短语肯定能做到毫无差错地归约到上一步，这虽然像是废话，但却是最左归约的精髓，只有这样才能保证归约过程万无一失，零回溯。

我们应该都有过文件操作的经验，比如 open 一个文件后，系统返回的是这个文件的句柄。为什么叫句柄呢？因为它牵动着另一端更大的文件数据块，操作句柄就是操作一个文件。句柄在现实生活中就是“把手”（也叫柄），比如铁锅的把手，连着的是另一端的更大的锅，拿到了手柄就相当于获得了铁锅。总之，“柄”的特征就是自己虽然小，但它却“把持”着通往更多数据内容的入口。上图中方框里的是直接短语，尽管比较短小（因为短语就是句型的一部分），但它却是归约到上一句型的钥匙，好像是一个句型的“柄”，因此称方框中的部分为句柄。按照官方定义来说，句柄就是“最左直接短语”。你看，上图中的 3 个框框都是各自“最左”边的直接短语，符合定义。句柄就是最右推导中每一步的非终结符的直接展开项（即直接短语），最左边的肯定是最后展开的，但对于归约来说，正是应该最先归约的，因此最左边的就是最左归约的开始入口，最左归约中每次归约的就是句柄，句柄就是可归约串，只有归约的是句柄才能做到万无一失逆向回归到开始符号。

这里补充一下句柄容易弄混的部分，不是最左端的就一定是句柄。就拿上图中 1 最左边的“e”来说，它可不是句柄。句柄的概念包括“直接短语”“最左”，首先得是直接短语才行。直接短语是非终结符所在产生式的“完整”右部，如果有候选式的话得是“完整”的候选式，即必须是非终结符的完整展开项，那个“e”只是非终结符 S 的一部分，因此不是短语，更不是句柄了。

0.35 算符优先分析法

因为我们实践中采用的语法分析方法有一点点像算符优先分析法，所以本文对算符优先分析法的介绍只是蜻蜓点水，就当是一点点铺垫。

算符优先分析法也是一种自下而上语法分析，既然是自下而上，那语法分析的思路就是归约，但它并不是严格的最左归约，它是通过“算符”的优先级来寻找可归约串进行归约，因此称为“算符”优先分析法。注意！算符不仅包括运算符，它是广义的，泛指文法中的终结符。

算符的优先级是提前写入到一张表中的，在语法分析过程中读取这张表来比较不同算符的优先级。注：以下优先级的符号只是示意，在各自符号中间再加个’.’才是文法中优先级比较符号。优先级比较结果有：

（1）$a > b$ 算符 a 的优先级大于算符 b；

（2）$a = b$ 算符 a 的优先级等于算符 b；

（3）$a < b$ 算符 a 的优先级小于算符 b。

注意，这里优先级和数学中的比较运算是不一样的，反过来不一定也成立。比如 $a > b$ 并不表示 $b < a$，原因是算符的大小不只和优先级本身有关，对于相同优先级的算符，先计算哪个是由结合性决定的，如右结合、左结合。因此算符的优先级是有序的，这个有序便指的是结合性，即算符的优先级还与出现的顺序有关。

0.36 算符优先文法

算符优先文法是用上节的算符优先分析法做语法分析的，在定义算符优先文法之前要先定义算符优先分析法。

在文法 G 中不存在形如 P->…AB…的产生式，其中 A 和 B 都是非终结符，就称 G 为算符文法。

为什么规定两个非终结符不能相继挨着呢？原因是算符肯定是终结符，而操作数可能是终结符或非终结符，如果两个非终结符相继挨着，展开后很可能是两个操作数相继挨着，这在数学运算中是不可能存在的，因为两个运算符之间只有 1 个操作数。相反，两个运算符却是可以相继，因为有的运算符是前缀，有的是后缀，相继挨在一起并不表示操作数是冲突的，比如 1− ++*a*。

假如 G 是一个不包含 ε 产生式的算符文法，对于做任意两个终结符 *a* 和 *b*。

（1）*a* = *b* 当且仅当文法 G 中含有形如 P->…*ab*…或 P->…*aXb*…的产生式。

（2）*a* < *b* 当且仅当文法 G 中含有形如 P->…*aY*…的产生式，且 $Y \overset{+}{\Rightarrow} b$…或 $Y \overset{+}{\Rightarrow} Xb$…。

（3）*a* > *b* 当且仅当文法 G 中含有形如 P->…*Yb*…的产生式，且 $Y \overset{+}{\Rightarrow} \ldots a$ 或 $Y \overset{+}{\Rightarrow} \ldots aX$。

如果文法 G 中任何终结符对儿（a,b）的优先级满足以上 3 种关系之一，就称文法 G 为算符优先文法。

对于 *a*=*b* 还是没什么好说的，下面说说后两个。

高优先级体现在更加优先被处理，优先级越高，被处理的优先权就越高。在文法中先被推导的部分说明其优先级更高（因为更先被处理）。在第二项中，*Y* 是非终结符，其产生式包含终结符 *b*，在此 *Y* 先被推导处理，说明 *b* 的优先级更高，因此 *a* < *b*。第三项 *a* > *b* 同理。不知你发现没有，将后两项中的 *Y* 展开后便是…*ab*…或…*aXb*…，同第一项一样。

0.37 非终结符中常常定义的因子和项是什么

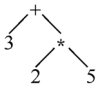
在 BNF 中经常会定义非终结符 TERM 和 FACTOR，翻译过来就是项和因子，他们常用作表示表达式计算中的操作数。加减法的操作数通常称为项，优先级更高的乘除法的操作数称为因子。因此乘除法的操作数称为因子。

0.38 什么是抽象语法树

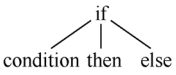
抽象语法树简称 AST（Abstract Syntax Tree），是对源代码的语法结构的抽象。抽象的意思是从事物中抽取本质性的特征，舍弃非本质的东西。因此抽象语法树就是源码的精髓。

既然称为树，那么源码就是以树形结构来表示。树上的每个节点都表示源码中的一种结构。举个例子，表达式 3+2 若用抽象语法树来表示，是以+为根，3 和 2 分别为其左右子树的二叉树，如下所示。

稍微复杂一点的，3+2*5 的语法树如下所示。



语法树并不是都是二叉树，像 if-condition-then-else 这样的语句就是三叉树，如下所示。



前面也说过，很少有生成抽象语法树的编译器，因为语法分析的过程就是遍历抽象语法树，因此没必要单独再生成它了，我们也是。

0.39 编译器如何使用或实现文法中的产生式

文法是语言的总体语法规则，用产生式来细化到具体规则，规则越多产生式就越多。产生式毕竟只是规则，编译器如何去实现这个规则，很多读者都不是很清楚，因此在学习产生式时往往不知书上所云，其实很简单。

产生式是分为两大部分，左部是非终结符，右部是该非终结符对应的定义，也称展开项。产生式是语法规则，因此语法分析器中会去实现这个规则。为方便编码，一般都采用自上而下递归下降的语法分析，自上而下大伙儿都懂，就是从文法中最大的非终结符即开始符号不断推导，直到推导出全是终结符的句子。递归下降是什么？这就是我所说的编码方便的地方，要想利用有限的规则创造无限句子，文法的设计必须要灵活、允许包含“递归”定义，即定义中包含自己，这样才能以“不变”的语法规则“应”“万变”的代码句子。比如以下产生式（定义不完整，仅举例）：

```
STAT -> WHILE_STAT | IF_STAT | BLOCK
…略
WHILE_STAT -> 'while' '('EXPRESSION')' '{'STAT'}'
```

一般在语言中与定义无关的代码称为 statement，即语句，与声明、定义有关的代码称为 definition，即定义。非终结符 STAT 表示语句。非终结符 WHILE_STAT 定义 while 的语法规则，其产生式右部中包含了非终结符 STAT，也就是 while 的循环体代码是用 STAT 表示的。STAT 对应的规则中又包含了 WHILE_STAT，即表示支持这样的 while 嵌套。

while(expression) {while(expression){}}

既然文法是递归的，那么在实现中递归调用是免不了的（当然也可以用循环代替，但有一定难度），一种可行方案是把非终结符对应到一个函数（也称子例程），即一种规则用一个函数来解析，递归调用各种非终结符对应的函数就行了。递归调用也得有个调用入口，开始符号也是非终结符，它所对应的函数就是整个语法分析递归调用的入口，由于开始符号是最上层，往下逐层调用便是下降的过程，因此称为递归下降。比如可以把开始符号对应的函数定义为 program，在 program 中调用其他非终结符对应的函数，期间发生各种递归，直到读完整个代码，最终回到 program 函数，语法分析结束。

这些非终结符对应的函数做了什么呢？也很简单，就是按照该非终结符的产生式右部分析语法，假设关键字 while 是用来做循环，其产生式（语法规则）是 WHILE_STAT =>while'('循环条件表达式')''{循环体}'，WHILE_STAT 对应的处理例程是 parseWhile，其伪码如下：

```
parseWhile ( ) {
   match('(');    // 匹配 while 关键字后面的“(”，如果没有“(”就报语法错误:while 后面应该接“(”
   parseExp();    // 解析循环条件
   match(')');    // 匹配条件后面的“)”，如果没有“)”就报语法错误:循环条件后面应该接“)”
```

```
    match('{');    // 条件之后是代码体，匹配代码体前面的“{”
    stat();        // 解析循环体代码
    match('}');    // 匹配循环体后面的“}”
}
```

好啦，就介绍到此，后面的实践会让你更有体会。

0.40 程序计数器 pc 与 ip 的区别

这两个都是 CPU 中的概念。CPU 是执行指令的机器，它必须要知道待执行的指令在哪里，这正是程序计数器的作用。程序计数器也称 PC，即 Program Counter，用于记录下一条待执行指令的地址。由于 CPU 每次只执行一条指令，执行了多个指令后，就相当于给程序中的指令统“计”了“数”量，因此称为程序计数器。程序计数器只是个概念，在各个 CPU 中都有自己的实现，比如 ARM 体系的 CPU 存储下一条指令地址的寄存器就叫 PC，而 X86 体系 CPU 存储下一条指令的寄存器称为 IP，即 Instruction Pointer。

本章内容如果不清楚也不要紧，不影响第 1 章～第 10 章的阅读。

异步社区 www.epubit.com.cn

第 1 章　设计一种面向对象脚本语言

有没有感觉设计一门语言实在是太有意思了，可以自定义语法规则，我的“地盘听我的”。

1.1 脚本语言的功能

本书设计一门纯粹的面向对象脚本语言，任何语言都有个名词，这里给这个语言起个名字——sparrow（麻雀）。它支持的功能如下。

（1）变量

支持局部变量和局部变量的定义。

变量可引用、赋值。

内部复合数据类型以大写字符开头，如 System.print()

（2）基本数据类型

数值：包括整数和浮点数。

字符串：包括普通字符和 unicode。

list：列表，如 Python 中的 list。

支持字面量创建，如['a', 'b']和 new 方法创建。

元素通过下标 list'[索引]'获得。

map：哈希数组，如 Python 中的字典。支持字面量创建和 new 创建。字面量创建如

```
{
    'k1':v1
    "k2":v2
}
```

key 可以是任何数据类型。同样支持 new 方法创建。

value 通过下标 map'[key]'获得。

range：用以确定一段整数范围，用符号..表示。range 包括 from 和 to 两个成员，分别表示这段范围的起和始，用区间表示[from, to]，即包括 from 和 to。如 range“2..6”，2 就是 from，6 就是 to，“2..6”表示 2、3、4、5、6。“..”类似于 Python 中的分片操作符“:”，只不过我们包括了结尾的 to，而 Python 不包括，若用区间表示则 to 后面的是右小括号“)”。

（3）运算

数值：+、−、*、/、%。

逻辑：>、<、==、!=、||、&&、？：、|、&等。

位运算：>>、<<。

方法调用：.。

异步社区 www.epubit.com.cn

索引：[]。

字符串：+、%（字符串内的表达式）

（4）控制结构

支持 if-else 选择。

支持 while 循环。

支持 for 循环。

支持 break 退出循环。

支持 continue，跳过本次循环体后面的部分，继续下一轮循环。

支持 return 返回。

（5）函数

尽管这是一门面向对象语言，但也支持传统意义上的函数，用关键字 fun 实现函数定义。

函数也是用类实现。

支持函数重载。

（6）类

就是传统意义上的 class，包括类定义和类实例，静态类。

实现继承，所有类都是 object 类的子类。

类成员（也称域，或字段）必须先声明再引用。

方法包括 method、getter、setter、subscript、subscriptSetter 和构造函数。支持块参数，块参数的参数是用“||”括起来的参数列表，以逗号分隔。

支持静态方法。

（7）线程

支持线程创建及调度。

（8）模块

支持执行模块和模块内模块变量的单独导入。

（9）注释

行注释：//

块注释：/*　块注释　*/

以上列举若有遗漏则以实际代码为主。

1.2 关键字

有以下关键字被提前征用了。

var：用于变量定义。

fun：用于函数定义。

if：用于条件判断。

else：用于条件判断的 else 分支。

true：bool 值真。

false：bool 值假。

while：用于 while 循环。

for：用于 for 循环。

break：用于退出循环。

continue：用于结束本次循环并进入下一轮循环。

异步社区 www.epubit.com.cn

return：用于从函数返回。
null：空值。
class：用于类定义。
is：用于判断类是否为某类的子类，即“is a”。
static：用于设置静态文法。
this：用于指向本实例。
super：用于指向父类。
import：用于导入模块。

1.3 脚本的执行方式

我们采用传统的虚拟机作为执行方式，即要实现一个虚拟机。编译器先把源码编译为 opcode，再让虚拟机执行 opcode。

opcode 即操作码，是自定义的一套专供虚拟机执行的指令，后面我们在实现虚拟机时会详细介绍。

1.4 “纯手工”的开发环境

既然本着教学的目的我觉得应该拿出教学的诚意，因此这里所说的“纯手工”是指编码中不想借助 STL 或其他类似的泛型语言，没有第三方库，一切以最基础最原始的形式展现语言的奥秘，因此选择了 C 语言，确定地说是 C89，并不是较新的 C99 标准，诚意满满，让我们纯手工去编码吧。

基础开发环境是：

（1）宿主系统是 Linux，采用 CentOS release 6.8 (Final)；

（2）编译器是 gcc，版本是 gcc version 4.4.7 20120313 (Red Hat 4.4.7-17) (GCC)。

1.5 定义 sparrow 语言的文法

在第 0 章介绍的文法中，我们采用的是大写字母表示非终结符，小写字母表示终结符，然而我们也说过了，在现实中为了便于编程，一般都用正规文法来定义语言，正规文法说白了就是用正则表达式定义的文法，因此本小节的基础是正则表达式，为保证容易看懂，我会用最简单的方式书写正则表达式。

值得注意的是，正规文法与第 0 章中介绍的文法有很大的差别，主要是涉及终结符是用"表示，原因是正规文法中会涉及()、[]和{}，这些在正规文法中都是元字符，有其特殊含义：()表示成为一组，[]表示范围，{}表示重复。但在实际语言它们只是字符串字面量（即终结符），比如在语言中()表示函数名后面的括号，也可表示表达式中的小括号，[]表示下标索引，因此为避免冲突，正规文法中用单引号括起的是终结符。

其实语法和传统语言差不多，只是用文法来描述就显得生涩了。注意，正规文法中的[]与 EBNF 中的意义不同，在此表示范围，其中可以用-表示一段连续的范围，比如 0-9 就表示 0 至 9 之间（包括 0 和 9）的任何数；a-z 同理，表示字母 a 到字母 z 之间的任何字母。[]后面一般会接量词，当然量词不一定只用在[]之后，但它一定是用在某个字符之后以表示该字符的数量，其前不能没有字符。按照数量级别划分有 3 种量词，+表示重复出现 1 次以上，*表示重复出现 0 次以上，?表示

出现 0 次或 1 次，比如可用[\t]*表示 0 个或多个空白字符，其中\t 是 tab。注意此处[]中的空白是空格，为了突显这里有个空格就写了两个。.表示任意字符，包括控制字符比如回车等，|表示或者、任意其一，比如 a|b，表示 a 和 b 两者取其一，注意，|是对两边的整体有效，并不是只对紧邻|的有效。比如对于 ab|cd 的意思是 ab 或者 cd，如果想表达 abd 或 acd，可以用分组符号()，就是小括号对儿。()表示作为一组考虑，使相应的正则符号应用于整个组成员。

如果不了解正则，可以暂时先看说明中的解释。文法产生式请见表 1-1 所列。

表 1-1

非终结符	产生式	说明	
数字	num -> 0x[0-9a-f]+| 0[0-9]+| [0-9]+'.'? [0-9]+	支持 0x 开头的十六进制整数 支持 0 开头的八进制整数 支持十进制整数及浮点数	
字符串	string -> '"'[.]* '"'	用双引号“”括起来的 0 至多个字符是字符串	
变量名	id -> [a-zA-Z_]+ [a-zA-Z_0-9]*	变量名只允许以字符 a-z、A-Z 和_开头，后面接 a-z、A-Z、_和数字	
定义变量	varDef -> 'var' [\t]+ id('=' exp)?	一个 var 后面只支持一个变量声明 声明变量：var id 或初始化：var id = 表达式	

if-else 分支选择	ifStat -> 'if' '(' exp ')'
	(block|statements)
	('else' block)?
		if (表达式) {
		代码块
		} else {
		代码块
		}
			if (表达式)
			语句
			else
			语句

while 循环	whileStat -> 'while'
	'(' exp ')' (block|
	statements)
		while (表达式) {
		循环体代码块
		}
			while (表达式)
			语句

for 循环	forStat -> 'for'[\t]+
	id[\t]+'('exp')' (block|
	statements)
		for i (可迭代序列)
		{
		循环体
		}
			for i (可迭代序列)
			语句

		注意，循环变量 i 后面没有 in	
break	breakStat -> 'break'	关键字 break	
continue	continueStat -> 'continue'	关键字 continue	
return	returnStat -> 'return'[\t]+ exp	关键字 return 无结果空返回：return 返回表达式结果：return 表达式	
参数列表	paralist -> id?(,id)*	参数列表: 参数可有可无，或者有多个，以逗号分隔	

函数定义	funDef -> 'fun' id
	'('paralist ')' block
		fun 函数名(形参列表) {
		函数体代码
		}	

下标调用	subscriptCall -> id[\t]* '['exp']'	下标是指[]，通常用来索引数组元素（脚本语言一般不支持数组），列表元素和哈希数组元素 [\t]*表示 0 个以上的空格或制表符，即有没有空白符都行 如 list[0]、map[key]	

异步社区 www.epubit.com.cn

续表

非终结符	产生式	说明
getter 方法调用	`getterCall -> id[ \t]*` `block?`	id 表示方法名，调用时只可以传递块参数，块参数是指函数体，即传递一个函数作为参数。下同 这里省略了方法前的类“.”和实例“.”，在后面的非终结符 methodsCall 会补全。假设 count 是类中的 getter 文法，那么完整调用如实例对象.count
setter 方法调用	`setterCall -> id[ \t]*` `'='[ \t]*exp`	与 getter 比，setter 后面多了个=和表达式，由于 setter 只支持一个赋值参数，因此不支持块参数，后面会介绍到：块参数也算作一个参数。假设 age=为 setter 文法，完整调用是： 对象.age=22

method 调用	`methodCall -> id[ \t]*`
	`'('paraList')'[ \t]*block?`
		method 就是类中的函数，只不过在“调用”时支持块参数，如：
		对象.method(实参) {
		块参数
		}
		强调!!!! 以上并不是定义，而是调用

类文法调用	`MethodsCall -> (id[ \t]*'.)?`
	`( methodCall |`
	`setterCall |`
	`getterCall )`
		(id[\t]*'.)?表示“实例对象名.”或“类名.”是可有可无的，这么做的原因是在类中调用本类中的文法不需要指定类名或实例对象。在类外调用方法的话必须要有“实例对象名.”或“类名.”

调用语句	`callStat -> subscriptCall |` `methodsCall`	调用包括下标调用和文法调用两类，如 list[1]是下标调用， instatant.method()是方法调用

语句列表	`statements -> (`
	`ifStat |`
	`whileStat |`
	`forStat|`
	`breakStat|`
	`continueStat|`
	`returnStat|`
	`callStat`
	`)*`
		参数可有可无，或者有多个，以逗号分隔

中缀运算符	`infixOp = '+' | '-' | '*' |`
	`'/' | '%' | '>' | '<' | '==' |`
	`'!=' | '>=' | '<=' | '&&' |`
	`'||' |'&' | '|' | '~' | '>>' |`
	`'<<'`
		中缀运算符即双目运算符

前缀运算符	`prefixOp = '-' | '!'`	
中缀运算符表达式	`infixExp -> exp` `([ \t*]infixOp [ \t]* exp)+`	中缀表达式，可以后接多个双目运算。非终结符 exp 在后面定义
前缀运算符表达式	`prefixExp -> prefixOp` `[ \t]*exp`	

表达式	`exp -> num |`
	`string |`
	`id |`
	`callStat |`
	`infixExp |`
	`prefixExp`
		递归调用表达式 exp

续表

非终结符	产生式	说明
语句块	block := '{' ('|' paralist '|')? statments '}'	语句块用大括号括起，支持块参数
实例域	instantField->varDef	实例域就是类的属性，属性各个对象自己的数据
静态域	staticField->'static'[\t]+ instantField	静态域就是类属性，属于所有类对象的共享数据
域	fieldDef -> instantField| staticField	
method 定义	methodDef -> 'static'?[\t]+ id'(' paralist')' block	

getter 定义	getterDef -> 'static'?[\t]+ id [\t]+ block	getter 是没有括号的文法，常用来返回类成员的值，逻辑上相当于只读，但功能不限制。如：
		count {
		return cnt
		}

setter 定义	setterDef -> 'static'?[\t]+id [\t]*'='[\t]*'('id')' [\t]+ block	setter 是 getter 后面接一个=，常用来修改类成员的值，逻辑上相当于写，只支持一个赋值，但方法内功能不限制
类中方法定义	methodsDef -> (methodDef | getterDef | setterDef)*	

类定义	classDef -> 'class' id (< id)?
	'{'
	fieldDef
	methodsDef
	'}'
		<表示继承，比如类定义：
		class 类名 [< 基类] {
		var instantFieldA
		var instantFieldA
		static var staticField
		instantMethodA(){
		xxx
		…
		}
		static staticMethod(){
		xxx
		…
		}
		}

初次接触文法的读者可能对递归定义感到“消化不良”，比如非终结符 exp 是用于定义表达式，exp 是由 infixExp 等非终结符组成，而 infixExp 又是由 exp 组成，看上去有点死循环出不来了，但你不要忘了，infixExp 只是 exp 的其中一个组成部分，exp 还可以由 num、id 等指代，num 和 id 下面再无递归定义，这就是递归终止的条件。因此 infixExp 的组成部分 exp 也会是 num 或 id 等。

以上只是大体上定义了语法，并不全面，大家知道大概意思就行了。这种方法定义看上去比较抽象，下面是个具体的样本。提示：可能由于排版软件的问题导致代码显示很乱，大家可以看本书附带的源码，以后的其他代码也是如此，不再赘述。

异步社区 www.epubit.com.cn

```
[work@work sparrow]$ cat -n employee.sp
 1 class Employee {
 2     var name
 3     var gender
 4     var age
 5     var salary
 6     static var employeeNum = 0
 7     new(n, g, a, s) {
 8        name = n
 9        gender = g
10        age = a
11        salary = s
12        employeeNum = employeeNum + 1
13     }
14
15     sayHi() {
16        System.print("My name is " +
17              name + ", I am a " + gender +
18              ", " + age.toString + "years old")
19     }
20
21     salary {
22         return salary
23     }
24
25     static employeeNum {
26         return employeeNum
27     }
28 }
[work@work sparrow]$ cat -n manager.sp
 1 /*
 2     本文件中的代码只为演示语法，无任何意义，不用深究
 3     这里只演示部分功能，如果读者感兴趣，
 4     可以参考 core.c 中后面注册的原生方法和 core.script.inc 中的脚本方法
 5     测试有限，难免还会有 bug，读过本书后应该有 bugfix 的能力，看好你，兄弟！
 6                                                  刚子
 7                                                  2018.4.12
 8 */
 9
10 import employee for Employee
11 var xh =  Employee.new("xiaohong", "female", 20, 6000)
12 System.print(xh.salary)
13
14 var xm =  Employee.new("xiaoming", "male", 23, 8000)
15 System.print(xm.salary)
16
17 System.print(Employee.employeeNum)
18
19 class Manager < Employee {
20     var bonus
```

异步社区 www.epubit.com.cn

```
    bonus=(v) {
        bonus = v
    }

    new(n, g, a, s, b) {
        super(n, g, a, s)
        bonus = b
    }

    salary {
        return super.salary + bonus
    }

}

fun employeeInfo() {
    System.print("number of employee:" + Employee.employeeNum.toString)
    var employeeTitle = Map.new()
    employeeTitle["xh"] = "rd"
    employeeTitle["xm"] = "op"
    employeeTitle["lw"] = "manager"
    employeeTitle["lz"] = "pm"

    for k (employeeTitle.keys) {
        System.print(k + " -> " + employeeTitle[k])
    }

    var employeeHeight = {
        "xh": 170,
        "xm": 172,
        "lw": 168,
        "lz": 173
    }
    var totalHeight = 0
    for v (employeeHeight.values) {
        totalHeight = totalHeight + v
    }
    System.print("averageHeight: %(totalHeight / employeeHeight.count)")

    var allEmployee = ["xh", "xm", "lw", "lz"]
    for e (allEmployee) {
        System.print(e)
    }

    allEmployee.add("xl")
    System.print("all employee are:%(allEmployee.toString)")
    var idx = 0
    var count = allEmployee.count
    while (idx < count) {
        System.print(allEmployee[idx])
```

异步社区 www.epubit.com.cn

```
71          idx = idx + 1
72      }
73
74      System.gc()  //可以手动回收内存
75
76      var a = 3 + 5 > 9 - 3  ? "yes" : "no"
77      if (a.endsWith("s")) {
78          System.print(System.clock)
79      } else {
80          System.print("error!!!!!")
81      }
82
83      var str = "hello, world."
84      System.print(str[-1..0])
85  }
86
87  var lw = Manager.new("laowang", "male", 35, 13000, 2000)
88  System.print(lw.salary)
89  lw.bonus=3100
90  System.print(lw.salary)
91  var lz = Manager.new("laozheng", "male", 36, 15000, 2300)
92  System.print(lz.salary)
93
94  var thread = Thread.new(employeeInfo)
95  thread.call()
```

以下是上述文件的执行结果，其中的 spr 是最终的脚本解释器（包括编译器及虚拟机），spr 是 sparrow 的缩写。

```
[work@work sparrow]$ ./spr manager.sp
6000
8000
2
15000
16100
17300
number of employee:4
xm -> op
lz -> pm
lw -> manager
xh -> rd
averageHeight: 170.75
xh
xm
lw
lz
all employee are:[xh,xm,lw,lz,xl]
xh
xm
lw
```

异步社区 www.epubit.com.cn

```
lz
xl
1468814885
.dlrow ,olleh
[work@work sparrow]$
```

以上./spr manager.sp 就是执行脚本文件 manager.sp，这与任何脚本语言的运行方法都是一致的，执行过就是脚本的输出，大家有兴趣可以核对一下结果，除了 System.clock 返回的时间戳是动态变化的外，其他不变。

异步社区 www.epubit.com.cn

第 10 章　命令行及调试

10.1 释放虚拟机（本节源码 stepByStep/c10/a）

创建虚拟机是 newVM 函数完成的，在虚拟机完成使命之后，我们还要将其释放，释放虚拟机的函数是 freeVM，freeVM 也是在 vm.c 中实现，代码如下。

stepByStep/c10/a/vm/vm.c

```
…略
58 //释放虚拟机 vm
59 void freeVM(VM* vm) {
60     ASSERT(vm->allMethodNames.count > 0, "VM have already been freed!");
61
62     //释放所有的对象
63     ObjHeader* objHeader = vm->allObjects;
64     while (objHeader != NULL) {
65         //释放之前先备份下一个节点地址
66         ObjHeader* next = objHeader->next;
67         freeObject(vm, objHeader);
68         objHeader = next;
69     }
70
71     vm->grays.grayObjects = DEALLOCATE(vm, vm->grays.grayObjects);
72     StringBufferClear(vm, &vm->allMethodNames);
73     DEALLOCATE(vm, vm);
74 }
…略
```

freeVM 的原理很简单，通过 while 循环遍历 vm->allObjects，调用 freeObject 逐一释放，然后调用 DEALLOCATE 释放 vm->grays.grayObjects，随后清空 vm->allMethodNames，最后调用 DEALLOCATE 释放虚拟机自身。

10.2 简单的命令行界面（本节源码 stepByStep/c10/a）

一直以来我们都是编译运行整个脚本文件，从未像 Python 那样在命令行中逐行运行代码，本节我们实现一个简单的命令行，这次修改的文件是如下的 cli.h 和 cli.c。

stepByStep/c10/a/cli/cli.h

```
1 #ifndef _CLI_CLI_H
```

www.epubit.com.cn 异步社区

```
 2 #define _CLI_CLI_H
 3
 4 #define VERSION 0.1.0
 5 #define MAX_LINE_LEN 1024
 6
 7 #endif
```

宏 MAX_LINE_LEN 表示支持的单行最大长度为 1024，我们用它来存储用户键入的单行代码。宏 VERSION 表示命令行的版本为 0.1.0。

stepByStep/c10/a/cli/cli.c

```
…略
24 //运行命令行
25 static void runCli(void) {
26    VM* vm = newVM();
27    char sourceLine[MAX_LINE_LEN];
28    printf("maque Version: 0.1\n");
29    while (true) {
30       printf(">>> ");
31
32       //若读取失败或者键入 quit 就退出循环
33       if (!fgets(sourceLine, MAX_LINE_LEN, stdin) ||
34             memcmp(sourceLine, "quit", 4) == 0) {
35          break;
36       }
37       executeModule(vm, OBJ_TO_VALUE(newObjString(vm, "cli", 3)), sourceLine);
38    }
39    freeVM(vm);
40 }
41
42 int main(int argc, const char** argv) {
43    if (argc == 1) {
44       runCli();
45    } else {
46       runFile(argv[1]);
47    }
48    return 0;
49 }
```

函数 runCli 就是我们的命令行的实现。实现比较简单，不支持多行编译，因此一次键入的代码必须符合完整语法，比如代码“for i ("abcd") {”会报错缺少右边的“}”。函数内部先调用 newVM 创建虚拟机实例，接下来的缓冲区 sourceLine 用来存储用户输入的代码，只存储一行。接下来的 while 是个死循环，先输出命令行提示符“>>> ”，然后读入用户的代码到 sourceLine，若键入的是“quit”就跳出循环，然后调用 freeVM 释放虚拟机，否则就调用 executeModule 编译执行 sourceLine 中的代码。

接下来在 main 函数中添加 runCli 的调用。下面是命令行运行的效果：

```
[work@work a]$ ./spr
maque Version: 0.1
>>> var a = ["a","b","c"]
>>> System.print(a)
```

异步社区 www.epubit.com.cn

```
[a,b,c]
>>> a.add("d")
>>> System.print(a[a.count-1])
d
>>> quit
[work@work a]$
```

10.3 调试（本节源码 stepByStep/c10/b）

本节是调试相关的代码，定义在 compiler 下 debug.h 和 debug.c 中，这部分我就不介绍了，大家需要的话自己看吧。说明一下，在使用 debug.c 中的代码时，记得在 makefile 中打开-DDEBUG 开关，也就是使用 makefile 中第 2 行的 CFLAGS 即可，把第 3 行的 CFLAGS 注释掉。

stepByStep/c10/b/compiler/debug.h

```
#ifdef DEBUG
    #ifndef _COMPILER_DEBUG_H
    #define _COMPILER_DEBUG_H
    #include "utils.h"
    #include "obj_fn.h"
    #include "obj_thread.h"

    void bindDebugFnName(VM* vm, FnDebug* fnDebug,
          const char* name, uint32_t length);
    void dumpValue(Value value);
    void dumpCode(VM* vm, ObjFn* fn);
    int dumpInstruction(VM* vm, ObjFn* fn, int i);
    void dumpStack(ObjThread* thread);
    void debugPrintStackTrace(VM* vm);
    #endif
#endif
```

stepByStep/c10/b/compiler/debug.c

```
#ifdef DEBUG
    #include <stdio.h>
    #include "debug.h"
    #include "vm.h"
    #include <string.h>

    //在 fnDebug 中绑定函数名
    void bindDebugFnName(VM* vm, FnDebug* fnDebug,
          const char* name, uint32_t length) {
       ASSERT(fnDebug->fnName == NULL, "debug.name has bound!");
       fnDebug->fnName = ALLOCATE_ARRAY(vm, char, length + 1);
       memcpy(fnDebug->fnName, name, length);
       fnDebug->fnName[length] = '\0';
    }

    //打印栈
    void dumpStack(ObjThread* thread) {
```

www.epubit.com.cn 异步社区

```
18        printf("(thread %p) stack:%p, esp:%p, slots:%ld ",
19              thread, thread->stack, thread->esp, thread->esp - thread->stack);
20        Value* slot = thread->stack;
21        while (slot < thread->esp) {
22           dumpValue(*slot);
23           printf(" | ");
24           slot++;
25        }
26        printf("\n");
27     }
28
29     //打印对象
30     static void dumpObject(ObjHeader* obj) {
31        switch (obj->type) {
32           case OT_CLASS:
33              printf("[class %s %p]", ((Class*)obj)->name->value.start, obj);
34              break;
35           case OT_CLOSURE:
36              printf("[closure %p]", obj);
37              break;
38           case OT_THREAD:
39              printf("[thread %p]", obj);
40              break;
41           case OT_FUNCTION:
42              printf("[fn %p]", obj);
43              break;
44           case OT_INSTANCE:
45              printf("[instance %p]", obj);
46              break;
47           case OT_LIST:
48              printf("[list %p]", obj);
49              break;
50           case OT_MAP:
51              printf("[map %p]", obj);
52              break;
53           case OT_MODULE:
54              printf("[module %p]", obj);
55              break;
56           case OT_RANGE:
57              printf("[range %p]", obj);
58              break;
59           case OT_STRING:
60              printf("%s", ((ObjString*)obj)->value.start);
61              break;
62           case OT_UPVALUE :
63              printf("[upvalue %p]", obj);
64              break;
65           default:
66              printf("[unknown object %d]", obj->type);
67              break;
```

异步社区 www.epubit.com.cn

```
68          }
69      }
70
71      //打印 value
72      void dumpValue(Value value) {
73          switch (value.type) {
74              case VT_FALSE:     printf("false"); break;
75              case VT_NULL:      printf("null"); break;
76              case VT_NUM:       printf("%.14g", VALUE_TO_NUM(value)); break;
77              case VT_TRUE:      printf("true"); break;
78              case VT_OBJ:       dumpObject(VALUE_TO_OBJ(value)); break;
79              case VT_UNDEFINED: NOT_REACHED();
80          }
81      }
82
83      //打印一条指令
84      static int dumpOneInstruction(VM* vm, ObjFn* fn, int i, int* lastLine) {
85          int start = i;
86          uint8_t* bytecode = fn->instrStream.datas;
87          OpCode opCode = (OpCode)bytecode[i];
88
89          int lineNo = fn->debug->lineNo.datas[i];
90
91          if (lastLine == NULL || *lastLine != lineNo) {
92              printf("%4d:", lineNo);    //输出源码行号
93              if (lastLine != NULL) {
94                  *lastLine = lineNo;
95              }
96          } else {
97              //不用输出源码行了,还是输出的 lastLine 行的指令流,空出行号的位置即可
98              printf("     ");
99          }
100
101         printf(" %04d  ", i++);   //输出指令流中的位置
102
103         #define READ_BYTE() (bytecode[i++])
104         #define READ_SHORT() (i += 2, (bytecode[i - 2] << 8) | bytecode[i - 1])
105
106         #define BYTE_INSTRUCTION(name) \
107             printf("%-16s %5d\n", name, READ_BYTE()); \
108             break; \
109
110         switch (opCode) {
111             case OPCODE_LOAD_CONSTANT: {
112                 int constant = READ_SHORT();
113                 printf("%-16s %5d '", "LOAD_CONSTANT", constant);
114                 dumpValue(fn->constants.datas[constant]);
115                 printf("'\n");
116                 break;
117             }
```

异步社区 www.epubit.com.cn

```
118
119         case OPCODE_PUSH_NULL:  printf("PUSH_NULL\n"); break;
120         case OPCODE_PUSH_FALSE: printf("PUSH_FALSE\n"); break;
121         case OPCODE_PUSH_TRUE:  printf("PUSH_TRUE\n"); break;
122
123         case OPCODE_LOAD_LOCAL_VAR: BYTE_INSTRUCTION("LOAD_LOCAL_VAR");
124         case OPCODE_STORE_LOCAL_VAR: BYTE_INSTRUCTION("STORE_LOCAL_VAR");
125         case OPCODE_LOAD_UPVALUE: BYTE_INSTRUCTION("LOAD_UPVALUE");
126         case OPCODE_STORE_UPVALUE: BYTE_INSTRUCTION("STORE_UPVALUE");
127
128         case OPCODE_LOAD_MODULE_VAR: {
129            int slot = READ_SHORT();
130            printf("%-16s %5d '%s'\n", "LOAD_MODULE_VAR", slot,
131                 fn->module->moduleVarName.datas[slot].str);
132            break;
133         }
134
135         case OPCODE_STORE_MODULE_VAR: {
136            int slot = READ_SHORT();
137            printf("%-16s %5d '%s'\n", "STORE_MODULE_VAR", slot,
138                 fn->module->moduleVarName.datas[slot].str);
139            break;
140         }
141
142         case OPCODE_LOAD_THIS_FIELD: BYTE_INSTRUCTION("LOAD_THIS_FIELD");
143         case OPCODE_STORE_THIS_FIELD: BYTE_INSTRUCTION("STORE_THIS_FIELD");
144         case OPCODE_LOAD_FIELD: BYTE_INSTRUCTION("LOAD_FIELD");
145         case OPCODE_STORE_FIELD: BYTE_INSTRUCTION("STORE_FIELD");
146
147         case OPCODE_POP: printf("POP\n"); break;
148
149         case OPCODE_CALL0:
150         case OPCODE_CALL1:
151         case OPCODE_CALL2:
152         case OPCODE_CALL3:
153         case OPCODE_CALL4:
154         case OPCODE_CALL5:
155         case OPCODE_CALL6:
156         case OPCODE_CALL7:
157         case OPCODE_CALL8:
158         case OPCODE_CALL9:
159         case OPCODE_CALL10:
160         case OPCODE_CALL11:
161         case OPCODE_CALL12:
162         case OPCODE_CALL13:
163         case OPCODE_CALL14:
164         case OPCODE_CALL15:
165         case OPCODE_CALL16: {
166            int numArgs = bytecode[i - 1] - OPCODE_CALL0;
167            int symbol = READ_SHORT();
```

www.epubit.com.cn

```
168             printf("CALL%-11d %5d '%s'\n", numArgs, symbol,
169                  vm->allMethodNames.datas[symbol].str);
170             break;
171          }
172
173          case OPCODE_SUPER0:
174          case OPCODE_SUPER1:
175          case OPCODE_SUPER2:
176          case OPCODE_SUPER3:
177          case OPCODE_SUPER4:
178          case OPCODE_SUPER5:
179          case OPCODE_SUPER6:
180          case OPCODE_SUPER7:
181          case OPCODE_SUPER8:
182          case OPCODE_SUPER9:
183          case OPCODE_SUPER10:
184          case OPCODE_SUPER11:
185          case OPCODE_SUPER12:
186          case OPCODE_SUPER13:
187          case OPCODE_SUPER14:
188          case OPCODE_SUPER15:
189          case OPCODE_SUPER16: {
190             int numArgs = bytecode[i - 1] - OPCODE_SUPER0;
191             int symbol = READ_SHORT();
192             int superclass = READ_SHORT();
193             printf("SUPER%-10d %5d '%s' %5d\n", numArgs, symbol,
194                  vm->allMethodNames.datas[symbol].str, superclass);
195             break;
196          }
197
198          case OPCODE_JUMP: {
199             int offset = READ_SHORT();
200             printf("%-16s offset:%-5d abs:%d\n", "JUMP", offset, i + offset);
201             break;
202          }
203
204          case OPCODE_LOOP: {
205             int offset = READ_SHORT();
206             printf("%-16s offset:%-5d abs:%d\n", "LOOP", offset, i - offset);
207             break;
208          }
209
210          case OPCODE_JUMP_IF_FALSE: {
211             int offset = READ_SHORT();
212             printf("%-16s offset:%-5d abs:%d\n", "JUMP_IF_FALSE", offset, i + offset);
213             break;
214          }
215
216          case OPCODE_AND: {
217             int offset = READ_SHORT();
```

异步社区 www.epubit.com.cn

```
218             printf("%-16s offset:%-5d abs:%d\n", "AND", offset, i + offset);
219             break;
220          }
221
222          case OPCODE_OR: {
223             int offset = READ_SHORT();
224             printf("%-16s offset:%-5d abs:%d\n", "OR", offset, i + offset);
225             break;
226          }
227
228          case OPCODE_CLOSE_UPVALUE:
229             printf("CLOSE_UPVALUE\n");
230             break;
231
232          case OPCODE_RETURN:
233             printf("RETURN\n");
234             break;
235
236          case OPCODE_CREATE_CLOSURE: {
237             int constant = READ_SHORT();
238             printf("%-16s %5d ", "CREATE_CLOSURE", constant);
239             dumpValue(fn->constants.datas[constant]);
240             printf(" ");
241             ObjFn* loadedFn = VALUE_TO_OBJFN(fn->constants.datas[constant]);
242             uint32_t j;
243             for (j = 0; j < loadedFn->upvalueNum; j++) {
244                int isLocal = READ_BYTE();
245                int index = READ_BYTE();
246                if (j > 0) printf(", ");
247                printf("%s %d", isLocal ? "local" : "upvalue", index);
248             }
249             printf("\n");
250             break;
251          }
252
253          case OPCODE_CONSTRUCT:
254             printf("CONSTRUCT\n");
255             break;
256
257          case OPCODE_CREATE_CLASS: {
258             int numFields = READ_BYTE();
259             printf("%-16s %5d fields\n", "CREATE_CLASS", numFields);
260             break;
261          }
262
263          case OPCODE_INSTANCE_METHOD: {
264             int symbol = READ_SHORT();
265             printf("%-16s %5d '%s'\n", "INSTANCE_METHOD", symbol,
266                    vm->allMethodNames.datas[symbol].str);
267             break;
```

www.cbbit.com.cn

```
268          }
269
270          case OPCODE_STATIC_METHOD: {
271             int symbol = READ_SHORT();
272             printf("%-16s %5d '%s'\n", "STATIC_METHOD", symbol,
273                   vm->allMethodNames.datas[symbol].str);
274             break;
275          }
276
277          case OPCODE_END:
278             printf("END\n");
279             break;
280
281          default:
282             printf("UKNOWN! [%d]\n", bytecode[i - 1]);
283             break;
284       }
285
286       //返回指令占用的字节数
287       if (opCode == OPCODE_END) {
288          return -1;
289       }
290       return i - start;
291
292       #undef READ_BYTE
293       #undef READ_SHORT
294    }
295
296    //打印指令
297    void dumpInstructions(VM* vm, ObjFn* fn) {
298       printf("module:[%s]\tfunction:[%s]\n",
299             fn->module->name == NULL ? "<core>" : fn->module->name->value.start,
300             fn->debug->fnName);
301
302       int i = 0;
303       int lastLine = -1;
304       while (true) {
305          int offset = dumpOneInstruction(vm, fn, i, &lastLine);
306          if (offset == -1) break;
307          i += offset;
308       }
309       printf("\n");
310    }
311
312 #endif
```

下面在 makefile 中打开调试选项，看看调试环境下的输出吧，GC 的工作只有在调试下才能看到，不能辜负了这位默默无闻的兄弟。

如果用默认的 makefile 文件编译，那么上面所说的在 makefile 中打开调试选项，是指把第二行 CFLAGS 前的注释字符“#”去掉，然后把第三行 CFLAGS 前加上注释字符“#”，很简单对吧，

异步社区 www.epubit.com.cn

其实这两步修改就是在 CFLAGS 中增加了选项“–DDEBUG”，此选项表示定义一个名为 DEBUG 的宏，因为我们程序中有判断，如果定义了宏 DEBUG，就额外做一些工作，比如调用不同的函数或输出额外的信息等。注意，尽管这里定义的宏名是 DEBUG，但不要误以为宏名必须为 DEBUG，gcc 的选项-D 可以定义任意名字的宏，我们这里将宏名定义为 DEBUG，是因为我们在程序中把名为 DEBUG 的宏做为调试的开关，这两者是配合在一起的，如果-D 中定义的是 XXX，那么程序中就要判断是否定义了名为 XXX 的宏。如果你不想修改默认的 makefile，那需要单独生成一个 makefile。为此我单独准备了一份，目录中的文件 makefile.debug 就是专门用来调试的 makefile，说明一下，文件名“makefile.debug”是随意起的，不需要包含“makefile”或“debug”。由于此文件并不是 make 程序默认使用的 makefile，那么用 make 编译时就要通过-f 参数指定该文件，如下所示。

```
[work@work b]$ make -f makefile.debug
```

回车之后就是编译过程了，编译过程就不贴出了，下面是打开调试选项后，spr 执行脚本文件 manager.sp 的过程，如下所示。

```
[work@work b]$ ./spr manager.sp
-- gc  before:40   nextGC:0  vm:0x734010  --
GC 40 before, 0 after (40 collected), next at 0. take 0.000s.
6000
8000
2
15000
16100
17300
number of employee:4
xm -> op
lz -> pm
lw -> manager
xh -> rd
averageHeight: 170.75
xh
xm
lw
lz
all employee are:[xh,xm,lw,lz,xl]
xh
xm
lw
lz
xl
-- gc  before:121976   nextGC:10485760  vm:0x734010  --
mark [thread 0x750af0] @ 0x750af0
mark [thread 0x744c40] @ 0x744c40
mark [closure 0x744b80] @ 0x744b80
mark [fn 0x740620] @ 0x740620
mark male @ 0x742560
mark laozheng @ 0x742520
mark male @ 0x7424f0
```

异步社区 www.epubit.com.cn

```
mark laowang @ 0x74c690
mark [fn 0x74c330] @ 0x74c330
mark hello, world. @ 0x74caa0
mark error!!!!! @ 0x74ca60
mark s @ 0x74ca30
mark no @ 0x74ca00
mark yes @ 0x74c9d0
mark  @ 0x74c960
mark all employee are: @ 0x74c990
mark xl @ 0x74c930
mark lz @ 0x74c900
mark lw @ 0x74c8d0
mark xm @ 0x74c8a0
mark xh @ 0x74c4b0
mark  @ 0x74c480
mark averageHeight:  @ 0x74c440
mark lz @ 0x74c600
mark lw @ 0x74c5d0
mark xm @ 0x743cc0
mark xh @ 0x7426a0
mark  ->  @ 0x74c5a0
mark pm @ 0x74c520
mark lz @ 0x74c4f0
mark manager @ 0x74c660
mark lw @ 0x74c630
mark op @ 0x74c570
mark xm @ 0x745150
mark rd @ 0x74c3b0
mark xh @ 0x7441d0
mark number of employee: @ 0x74c400
mark [fn 0x745040] @ 0x745040
mark [class Employee 0x74f620] @ 0x74f620
mark Employee @ 0x7503b0
mark [closure 0x74f6f0] @ 0x74f6f0
mark [fn 0x74fa50] @ 0x74fa50
mark years old @ 0x74fc30
mark ,  @ 0x74fc70
mark , I am a  @ 0x74fba0
mark My name is  @ 0x74fb10
mark [closure 0x74d1e0] @ 0x74d1e0
mark [fn 0x74fe40] @ 0x74fe40
mark [closure 0x750480] @ 0x750480
mark [upvalue 0x7504b0] @ 0x7504b0
mark [fn 0x74d2d0] @ 0x74d2d0
mark [class object 0x734ab0] @ 0x734ab0
mark object @ 0x734b00
mark [class objectMeta 0x734c70] @ 0x734c70
mark objectMeta @ 0x734f50
mark [class class 0x734cc0] @ 0x734cc0
mark class @ 0x734c40
```

异步社区 www.epubit.com.cn

```
mark [class Employee metaclass 0x744df0] @ 0x744df0
mark Employee metaclass @ 0x74fa00
mark [closure 0x74d210] @ 0x74d210
mark [fn 0x74fec0] @ 0x74fec0
mark [closure 0x74d180] @ 0x74d180
mark [fn 0x74f8c0] @ 0x74f8c0
mark [fn 0x744f50] @ 0x744f50
mark [fn 0x7440f0] @ 0x7440f0
mark [fn 0x740590] @ 0x740590
mark Manager @ 0x7441a0
mark male @ 0x7444d0
mark xiaoming @ 0x743d10
mark female @ 0x7405f0
mark xiaohong @ 0x743d50
mark Employee @ 0x743c80
mark employee @ 0x744450
mark [class System 0x743fb0] @ 0x743fb0
mark System @ 0x744000
mark [class System metaclass 0x743e90] @ 0x743e90
mark System metaclass @ 0x743ee0
mark [closure 0x744500] @ 0x744500
mark [fn 0x73db40] @ 0x73db40
mark [closure 0x743f80] @ 0x743f80
mark [fn 0x73d8c0] @ 0x73d8c0
mark [closure 0x743f50] @ 0x743f50
mark [fn 0x73d7e0] @ 0x73d7e0
mark
 @ 0x73d950
mark [closure 0x744530] @ 0x744530
mark [fn 0x73dc60] @ 0x73dc60
mark [invalid toString] @ 0x73def0
mark [closure 0x743f20] @ 0x743f20
mark [fn 0x739550] @ 0x739550
mark
 @ 0x73d6f0
mark [closure 0x7440c0] @ 0x7440c0
mark [fn 0x7393b0] @ 0x7393b0
mark
 @ 0x7394b0
mark [list 0x756060] @ 0x756060
mark [map 0x755600] @ 0x755600
mark [map 0x750bd0] @ 0x750bd0
mark [closure 0x7505e0] @ 0x7505e0
mark [map 0x734110] @ 0x734110
mark [module 0x744cd0] @ 0x744cd0
mark employee @ 0x744d20
mark [class Range 0x743c30] @ 0x743c30
mark Range @ 0x743200
mark [closure 0x73f260] @ 0x73f260
mark [fn 0x737bf0] @ 0x737bf0
```

异步社区 www.epubit.com.cn

```
mark [closure 0x73f230] @ 0x73f230
mark [fn 0x737a90] @ 0x737a90
mark  @ 0x737b30
mark [closure 0x73f200] @ 0x73f200
mark [fn 0x736040] @ 0x736040
mark  @ 0x737300
mark [closure 0x73f1d0] @ 0x73f1d0
mark [fn 0x7371b0] @ 0x7371b0
mark Can't reduce an empty sequence. @ 0x737330
mark [closure 0x73ef40] @ 0x73ef40
mark [fn 0x736f70] @ 0x736f70
mark [closure 0x73ef10] @ 0x73ef10
mark [fn 0x736df0] @ 0x736df0
mark [closure 0x73eee0] @ 0x73eee0
mark [fn 0x7353b0] @ 0x7353b0
mark [closure 0x73eeb0] @ 0x73eeb0
mark [fn 0x7368d0] @ 0x7368d0
mark [closure 0x73ee80] @ 0x73ee80
mark [fn 0x7366a0] @ 0x7366a0
mark [closure 0x73ef90] @ 0x73ef90
mark [fn 0x736450] @ 0x736450
mark [closure 0x73ee20] @ 0x73ee20
mark [fn 0x735690] @ 0x735690
mark [closure 0x73edf0] @ 0x73edf0
mark [fn 0x735e70] @ 0x735e70
mark [closure 0x73edc0] @ 0x73edc0
mark [fn 0x735c80] @ 0x735c80
mark [closure 0x73ee50] @ 0x73ee50
mark [fn 0x735580] @ 0x735580
mark [class Range metaclass 0x743b10] @ 0x743b10
mark Range metaclass @ 0x743b60
mark [class MapValueSequence 0x743230] @ 0x743230
mark MapValueSequence @ 0x743280
mark [closure 0x7427f0] @ 0x7427f0
mark [fn 0x73cc70] @ 0x73cc70
mark [closure 0x7431d0] @ 0x7431d0
mark [fn 0x739220] @ 0x739220
mark [closure 0x7431a0] @ 0x7431a0
mark [fn 0x738b10] @ 0x738b10
mark [class MapValueSequence metaclass 0x743100] @ 0x743100
mark MapValueSequence metaclass @ 0x743150
mark [closure 0x7436d0] @ 0x7436d0
mark [fn 0x73cd80] @ 0x73cd80
mark [class MapKeySequence 0x742820] @ 0x742820
mark MapKeySequence @ 0x742870
mark [closure 0x742280] @ 0x742280
mark [fn 0x73c7f0] @ 0x73c7f0
mark [closure 0x7427c0] @ 0x7427c0
mark [fn 0x73cb00] @ 0x73cb00
mark [closure 0x742790] @ 0x742790
```

异步社区 www.epubit.com.cn

```
mark [fn 0x73c9d0] @ 0x73c9d0
mark [class MapKeySequence metaclass 0x7426f0] @ 0x7426f0
mark MapKeySequence metaclass @ 0x742740
mark [closure 0x742cc0] @ 0x742cc0
mark [fn 0x73c8e0] @ 0x73c8e0
mark [class Map 0x741d50] @ 0x741d50
mark Map @ 0x741da0
mark [closure 0x742220] @ 0x742220
mark [fn 0x738820] @ 0x738820
mark [closure 0x7422b0] @ 0x7422b0
mark [fn 0x73afd0] @ 0x73afd0
mark [closure 0x742250] @ 0x742250
mark [fn 0x738970] @ 0x738970
mark } @ 0x73c7c0
mark  @ 0x73c450
mark :  @ 0x73c420
mark  @ 0x73c4a0
mark ,  @ 0x73c360
mark { @ 0x738a50
mark [class Map metaclass 0x741c30] @ 0x741c30
mark Map metaclass @ 0x741c80
mark [class StringCodePointSequence 0x740840] @ 0x740840
mark StringCodePointSequence @ 0x740890
mark [closure 0x7407b0] @ 0x7407b0
mark [fn 0x73a290] @ 0x73a290
mark [closure 0x741380] @ 0x741380
mark [fn 0x73a0f0] @ 0x73a0f0
mark [closure 0x741170] @ 0x741170
mark [fn 0x73a5e0] @ 0x73a5e0
mark [closure 0x740810] @ 0x740810
mark [fn 0x73a4d0] @ 0x73a4d0
mark [closure 0x7407e0] @ 0x7407e0
mark [fn 0x73a3c0] @ 0x73a3c0
mark [class StringCodePointSequence metaclass 0x740710] @ 0x740710
mark StringCodePointSequence metaclass @ 0x740760
mark [closure 0x7408d0] @ 0x7408d0
mark [fn 0x73a1d0] @ 0x73a1d0
mark [class StringByteSequence 0x740680] @ 0x740680
mark StringByteSequence @ 0x7406d0
mark [closure 0x7401f0] @ 0x7401f0
mark [fn 0x739be0] @ 0x739be0
mark [closure 0x740920] @ 0x740920
mark [fn 0x7376c0] @ 0x7376c0
mark [closure 0x740190] @ 0x740190
mark [fn 0x739f70] @ 0x739f70
mark [closure 0x740160] @ 0x740160
mark [fn 0x739e60] @ 0x739e60
mark [closure 0x740130] @ 0x740130
mark [fn 0x739d30] @ 0x739d30
mark [class StringByteSequence metaclass 0x740090] @ 0x740090
```

异步社区 www.epubit.com.cn

```
mark StringByteSequence metaclass @ 0x7400e0
mark [closure 0x7401c0] @ 0x7401c0
mark [fn 0x739b20] @ 0x739b20
mark [class String 0x73ffe0] @ 0x73ffe0
mark String @ 0x73f9b0
mark [closure 0x740060] @ 0x740060
mark [fn 0x7375b0] @ 0x7375b0
mark  @ 0x737720
mark Count must be a non-negative integer. @ 0x739790
mark [closure 0x740030] @ 0x740030
mark [fn 0x7362f0] @ 0x7362f0
mark [closure 0x740240] @ 0x740240
mark [fn 0x738db0] @ 0x738db0
mark [class String metaclass 0x73fec0] @ 0x73fec0
mark String metaclass @ 0x73ff10
mark [class List 0x7412c0] @ 0x7412c0
mark List @ 0x741310
mark [closure 0x741de0] @ 0x741de0
mark [fn 0x73a700] @ 0x73a700
mark [closure 0x741c00] @ 0x741c00
mark [fn 0x73aea0] @ 0x73aea0
mark Count must be a non-negative integer. @ 0x73b0f0
mark [closure 0x741bd0] @ 0x741bd0
mark [fn 0x73ace0] @ 0x73ace0
mark [closure 0x741340] @ 0x741340
mark [fn 0x73a7f0] @ 0x73a7f0
mark ] @ 0x73aba0
mark , @ 0x73aa80
mark [ @ 0x73a850
mark [class List metaclass 0x7411a0] @ 0x7411a0
mark List metaclass @ 0x7411f0
mark [class WhereSequence 0x73f9e0] @ 0x73f9e0
mark WhereSequence @ 0x73fa30
mark [closure 0x73f380] @ 0x73f380
mark [fn 0x735940] @ 0x735940
mark [closure 0x73f980] @ 0x73f980
mark [fn 0x738ca0] @ 0x738ca0
mark [closure 0x73f950] @ 0x73f950
mark [fn 0x736cc0] @ 0x736cc0
mark [class WhereSequence metaclass 0x73f8c0] @ 0x73f8c0
mark WhereSequence metaclass @ 0x73f910
mark [closure 0x73fc80] @ 0x73fc80
mark [fn 0x736bd0] @ 0x736bd0
mark [class MapSequence 0x73f3b0] @ 0x73f3b0
mark MapSequence @ 0x73f400
mark [closure 0x73f650] @ 0x73f650
mark [fn 0x737f20] @ 0x737f20
mark [closure 0x73f350] @ 0x73f350
mark [fn 0x738230] @ 0x738230
mark [closure 0x73f320] @ 0x73f320
```

异步社区 www.epubit.com.cn

```
mark [fn 0x738120] @ 0x738120
mark [class MapSequence metaclass 0x73f290] @ 0x73f290
mark MapSequence metaclass @ 0x73f2e0
mark [closure 0x73f680] @ 0x73f680
mark [fn 0x738030] @ 0x738030
mark [class Sequence 0x73ed30] @ 0x73ed30
mark Sequence @ 0x73ed80
mark [class Sequence metaclass 0x73ec10] @ 0x73ec10
mark Sequence metaclass @ 0x73ec60
mark [class Thread 0x73eb00] @ 0x73eb00
mark Thread @ 0x73eb50
mark [class Thread metaclass 0x73e9e0] @ 0x73e9e0
mark Thread metaclass @ 0x73ea30
mark [class Fn 0x73e8d0] @ 0x73e8d0
mark Fn @ 0x73e920
mark [class Fn metaclass 0x73e7b0] @ 0x73e7b0
mark Fn metaclass @ 0x73e800
mark [class Num 0x73e6a0] @ 0x73e6a0
mark Num @ 0x73e6f0
mark [class Num metaclass 0x73e580] @ 0x73e580
mark Num metaclass @ 0x73e5d0
mark [class Bool 0x73e470] @ 0x73e470
mark Bool @ 0x73e4c0
mark [class Bool metaclass 0x73e350] @ 0x73e350
mark Bool metaclass @ 0x73e3a0
mark [class Null 0x73e270] @ 0x73e270
mark Null @ 0x73dcc0
mark [class Null metaclass 0x73e190] @ 0x73e190
mark Null metaclass @ 0x73de20
mark [module 0x740480] @ 0x740480
mark manager.sp @ 0x7404d0
mark [instance 0x7508d0] @ 0x7508d0
mark [instance 0x750750] @ 0x750750
mark [class Manager 0x750640] @ 0x750640
mark Manager @ 0x750690
mark [closure 0x7506f0] @ 0x7506f0
mark [closure 0x7506c0] @ 0x7506c0
mark [closure 0x7505b0] @ 0x7505b0
mark [class Manager metaclass 0x750520] @ 0x750520
mark Manager metaclass @ 0x750570
mark [closure 0x750720] @ 0x750720
mark [instance 0x750320] @ 0x750320
mark [instance 0x7503f0] @ 0x7503f0
mark manager.sp @ 0x7391c0
mark [module 0x734250] @ 0x734250
free all employee are:[xh,xm,lw,lz,xl] @ 0x756770
free all employee are:[xh,xm,lw,lz,xl] @ 0x756720
free all employee are:[xh,xm,lw,lz,xl] @ 0x7566d0
free all employee are: @ 0x756690
free all employee are: @ 0x756650
```

异步社区 www.epubit.com.cn

```
free [xh,xm,lw,lz,xl] @ 0x7565c0
free [xh,xm,lw,lz,xl @ 0x756580
free [xh,xm,lw,lz,xl @ 0x756540
free [ @ 0x756510
free [ @ 0x7564e0
free xh,xm,lw,lz,xl @ 0x756450
free xh,xm,lw,lz, @ 0x756410
free xh,xm,lw,lz @ 0x7563d0
free xh,xm,lw, @ 0x756390
free xh,xm,lw @ 0x750aa0
free xh,xm, @ 0x750a70
free xh,xm @ 0x750a40
free xh, @ 0x750a10
free xh @ 0x7509e0
free [list 0x756150] @ 0x756150
free [list 0x756120] @ 0x756120
free averageHeight: 170.75 @ 0x756020
free averageHeight: 170.75 @ 0x755fe0
free averageHeight: 170.75 @ 0x755fa0
free 170.75 @ 0x755f70
free averageHeight:  @ 0x755f30
free averageHeight:  @ 0x755ef0
free [list 0x755e70] @ 0x755e70
free [instance 0x755e40] @ 0x755e40
free xh -> rd @ 0x7555c0
free xh ->  @ 0x755590
free lw -> manager @ 0x755550
free lw ->  @ 0x7514c0
free lz -> pm @ 0x751480
free lz ->  @ 0x751450
free xm -> op @ 0x751410
free xm ->  @ 0x744c10
free [instance 0x744be0] @ 0x744be0
free number of employee:4 @ 0x750b90
free 4 @ 0x750b60
free 17300 @ 0x750940
free 16100 @ 0x744bb0
free 15000 @ 0x750610
free 2 @ 0x7504f0
free 8000 @ 0x750380
free 6000 @ 0x750450
free [thread 0x7502b0] @ 0x7502b0
free [closure 0x74f9d0] @ 0x74f9d0
free Employee @ 0x74d140
free [fn 0x74d760] @ 0x74d760
free [thread 0x73e120] @ 0x73e120
free [closure 0x73dbe0] @ 0x73dbe0
free System @ 0x739380
free Range @ 0x739350
free MapValueSequence @ 0x73cc30
```

异步社区 www.epubit.com.cn

```
free MapKeySequence @ 0x738ad0
free Map @ 0x73ab70
free List @ 0x73a660
free StringCodePointSequence @ 0x73a0b0
free StringByteSequence @ 0x7396e0
free String @ 0x737d80
free WhereSequence @ 0x738340
free MapSequence @ 0x737d40
free Sequence @ 0x735370
free Thread @ 0x735490
free Fn @ 0x7350b0
free Num @ 0x735550
free Bool @ 0x734c10
free Null @ 0x734e90
free [fn 0x734d30] @ 0x734d30
GC 121976 before, 93580 after (28396 collected), next at 1048576. take 0.000s.
1502006315
.dlrow ,olleh
free .dlrow ,olleh @ 0x737d40
free [range 0x734d90] @ 0x734d90
free 1502006315 @ 0x735370
free [list 0x756060] @ 0x756060
free [map 0x755600] @ 0x755600
free [map 0x750bd0] @ 0x750bd0
free [thread 0x750af0] @ 0x750af0
free [instance 0x7508d0] @ 0x7508d0
free [instance 0x750750] @ 0x750750
free [closure 0x7505e0] @ 0x7505e0
free [closure 0x7505b0] @ 0x7505b0
free [closure 0x750720] @ 0x750720
free [closure 0x7506f0] @ 0x7506f0
free [closure 0x7506c0] @ 0x7506c0
free Manager @ 0x750690
free [class Manager 0x750640] @ 0x750640
free Manager metaclass @ 0x750570
free [class Manager metaclass 0x750520] @ 0x750520
free [instance 0x750320] @ 0x750320
free [instance 0x7503f0] @ 0x7503f0
free [closure 0x74d210] @ 0x74d210
free [closure 0x74d1e0] @ 0x74d1e0
free [closure 0x74f6f0] @ 0x74f6f0
free [closure 0x74d180] @ 0x74d180
free [upvalue 0x7504b0] @ 0x7504b0
free [closure 0x750480] @ 0x750480
free Employee @ 0x7503b0
free [class Employee 0x74f620] @ 0x74f620
free Employee metaclass @ 0x74fa00
free [class Employee metaclass 0x744df0] @ 0x744df0
free [fn 0x74fec0] @ 0x74fec0
free [fn 0x74fe40] @ 0x74fe40
```

异步社区 www.epubit.com.cn

```
free years old @ 0x74fc30
free ,  @ 0x74fc70
free , I am a  @ 0x74fba0
free My name is  @ 0x74fb10
free [fn 0x74fa50] @ 0x74fa50
free [fn 0x74f8c0] @ 0x74f8c0
free [fn 0x74d2d0] @ 0x74d2d0
free employee @ 0x744d20
free [module 0x744cd0] @ 0x744cd0
free [thread 0x744c40] @ 0x744c40
free [closure 0x744b80] @ 0x744b80
free male @ 0x742560
free laozheng @ 0x742520
free male @ 0x7424f0
free laowang @ 0x74c690
free hello, world. @ 0x74caa0
free error!!!!! @ 0x74ca60
free s @ 0x74ca30
free no @ 0x74ca00
free yes @ 0x74c9d0
free  @ 0x74c960
free all employee are: @ 0x74c990
free xl @ 0x74c930
free lz @ 0x74c900
free lw @ 0x74c8d0
free xm @ 0x74c8a0
free xh @ 0x74c4b0
free  @ 0x74c480
free averageHeight:  @ 0x74c440
free lz @ 0x74c600
free lw @ 0x74c5d0
free xm @ 0x743cc0
free xh @ 0x7426a0
free  ->  @ 0x74c5a0
free pm @ 0x74c520
free lz @ 0x74c4f0
free manager @ 0x74c660
free lw @ 0x74c630
free op @ 0x74c570
free xm @ 0x745150
free rd @ 0x74c3b0
free xh @ 0x7441d0
free number of employee: @ 0x74c400
free [fn 0x74c330] @ 0x74c330
free [fn 0x745040] @ 0x745040
free [fn 0x744f50] @ 0x744f50
free [fn 0x7440f0] @ 0x7440f0
free [fn 0x740590] @ 0x740590
free Manager @ 0x7441a0
free male @ 0x7444d0
```

异步社区 www.epubit.com.cn

```
free xiaoming @ 0x743d10
free female @ 0x7405f0
free xiaohong @ 0x743d50
free Employee @ 0x743c80
free employee @ 0x744450
free [fn 0x740620] @ 0x740620
free manager.sp @ 0x7404d0
free [module 0x740480] @ 0x740480
free manager.sp @ 0x7391c0
free [closure 0x744530] @ 0x744530
free [closure 0x744500] @ 0x744500
free [closure 0x743f80] @ 0x743f80
free [closure 0x743f50] @ 0x743f50
free [closure 0x743f20] @ 0x743f20
free [closure 0x7440c0] @ 0x7440c0
free System @ 0x744000
free [class System 0x743fb0] @ 0x743fb0
free System metaclass @ 0x743ee0
free [class System metaclass 0x743e90] @ 0x743e90
free Range @ 0x743200
free [class Range 0x743c30] @ 0x743c30
free Range metaclass @ 0x743b60
free [class Range metaclass 0x743b10] @ 0x743b10
free [closure 0x7431d0] @ 0x7431d0
free [closure 0x7431a0] @ 0x7431a0
free [closure 0x7436d0] @ 0x7436d0
free [closure 0x7427f0] @ 0x7427f0
free MapValueSequence @ 0x743280
free [class MapValueSequence 0x743230] @ 0x743230
free MapValueSequence metaclass @ 0x743150
free [class MapValueSequence metaclass 0x743100] @ 0x743100
free [closure 0x7427c0] @ 0x7427c0
free [closure 0x742790] @ 0x742790
free [closure 0x742cc0] @ 0x742cc0
free [closure 0x742280] @ 0x742280
free MapKeySequence @ 0x742870
free [class MapKeySequence 0x742820] @ 0x742820
free MapKeySequence metaclass @ 0x742740
free [class MapKeySequence metaclass 0x7426f0] @ 0x7426f0
free [closure 0x742250] @ 0x742250
free [closure 0x742220] @ 0x742220
free [closure 0x7422b0] @ 0x7422b0
free Map @ 0x741da0
free [class Map 0x741d50] @ 0x741d50
free Map metaclass @ 0x741c80
free [class Map metaclass 0x741c30] @ 0x741c30
free [closure 0x741c00] @ 0x741c00
free [closure 0x741bd0] @ 0x741bd0
free [closure 0x741340] @ 0x741340
free [closure 0x741de0] @ 0x741de0
```

异步社区 www.epubit.com.cn

```
free List @ 0x741310
free [class List 0x7412c0] @ 0x7412c0
free List metaclass @ 0x7411f0
free [class List metaclass 0x7411a0] @ 0x7411a0
free [closure 0x741170] @ 0x741170
free [closure 0x740810] @ 0x740810
free [closure 0x7407e0] @ 0x7407e0
free [closure 0x7407b0] @ 0x7407b0
free [closure 0x7408d0] @ 0x7408d0
free [closure 0x741380] @ 0x741380
free StringCodePointSequence @ 0x740890
free [class StringCodePointSequence 0x740840] @ 0x740840
free StringCodePointSequence metaclass @ 0x740760
free [class StringCodePointSequence metaclass 0x740710] @ 0x740710
free [closure 0x740190] @ 0x740190
free [closure 0x740160] @ 0x740160
free [closure 0x740130] @ 0x740130
free [closure 0x7401f0] @ 0x7401f0
free [closure 0x7401c0] @ 0x7401c0
free [closure 0x740920] @ 0x740920
free StringByteSequence @ 0x7406d0
free [class StringByteSequence 0x740680] @ 0x740680
free StringByteSequence metaclass @ 0x7400e0
free [class StringByteSequence metaclass 0x740090] @ 0x740090
free [closure 0x740060] @ 0x740060
free [closure 0x740030] @ 0x740030
free [closure 0x740240] @ 0x740240
free String @ 0x73f9b0
free [class String 0x73ffe0] @ 0x73ffe0
free String metaclass @ 0x73ff10
free [class String metaclass 0x73fec0] @ 0x73fec0
free [closure 0x73f980] @ 0x73f980
free [closure 0x73f950] @ 0x73f950
free [closure 0x73fc80] @ 0x73fc80
free [closure 0x73f380] @ 0x73f380
free WhereSequence @ 0x73fa30
free [class WhereSequence 0x73f9e0] @ 0x73f9e0
free WhereSequence metaclass @ 0x73f910
free [class WhereSequence metaclass 0x73f8c0] @ 0x73f8c0
free [closure 0x73f350] @ 0x73f350
free [closure 0x73f320] @ 0x73f320
free [closure 0x73f680] @ 0x73f680
free [closure 0x73f650] @ 0x73f650
free MapSequence @ 0x73f400
free [class MapSequence 0x73f3b0] @ 0x73f3b0
free MapSequence metaclass @ 0x73f2e0
free [class MapSequence metaclass 0x73f290] @ 0x73f290
free [closure 0x73f260] @ 0x73f260
free [closure 0x73f230] @ 0x73f230
free [closure 0x73f200] @ 0x73f200
```

异步社区 www.epubit.com.cn

```
free [closure 0x73f1d0] @ 0x73f1d0
free [closure 0x73ef40] @ 0x73ef40
free [closure 0x73ef10] @ 0x73ef10
free [closure 0x73eee0] @ 0x73eee0
free [closure 0x73eeb0] @ 0x73eeb0
free [closure 0x73ee80] @ 0x73ee80
free [closure 0x73ef90] @ 0x73ef90
free [closure 0x73ee20] @ 0x73ee20
free [closure 0x73edf0] @ 0x73edf0
free [closure 0x73edc0] @ 0x73edc0
free [closure 0x73ee50] @ 0x73ee50
free Sequence @ 0x73ed80
free [class Sequence 0x73ed30] @ 0x73ed30
free Sequence metaclass @ 0x73ec60
free [class Sequence metaclass 0x73ec10] @ 0x73ec10
free Thread @ 0x73eb50
free [class Thread 0x73eb00] @ 0x73eb00
free Thread metaclass @ 0x73ea30
free [class Thread metaclass 0x73e9e0] @ 0x73e9e0
free Fn @ 0x73e920
free [class Fn 0x73e8d0] @ 0x73e8d0
free Fn metaclass @ 0x73e800
free [class Fn metaclass 0x73e7b0] @ 0x73e7b0
free Num @ 0x73e6f0
free [class Num 0x73e6a0] @ 0x73e6a0
free Num metaclass @ 0x73e5d0
free [class Num metaclass 0x73e580] @ 0x73e580
free Bool @ 0x73e4c0
free [class Bool 0x73e470] @ 0x73e470
free Bool metaclass @ 0x73e3a0
free [class Bool metaclass 0x73e350] @ 0x73e350
free Null @ 0x73dcc0
free [class Null 0x73e270] @ 0x73e270
free Null metaclass @ 0x73de20
free [class Null metaclass 0x73e190] @ 0x73e190
free [invalid toString] @ 0x73def0
free [fn 0x73dc60] @ 0x73dc60
free [fn 0x73db40] @ 0x73db40
free [fn 0x73d8c0] @ 0x73d8c0
free
 @ 0x73d950
free [fn 0x73d7e0] @ 0x73d7e0
free
 @ 0x73d6f0
free [fn 0x739550] @ 0x739550
free
 @ 0x7394b0
free [fn 0x7393b0] @ 0x7393b0
free [fn 0x739220] @ 0x739220
free [fn 0x738b10] @ 0x738b10
```

异步社区 www.epubit.com.cn

```
free [fn 0x73cd80] @ 0x73cd80
free [fn 0x73cc70] @ 0x73cc70
free [fn 0x73cb00] @ 0x73cb00
free [fn 0x73c9d0] @ 0x73c9d0
free [fn 0x73c8e0] @ 0x73c8e0
free [fn 0x73c7f0] @ 0x73c7f0
free } @ 0x73c7c0
free  @ 0x73c450
free :  @ 0x73c420
free  @ 0x73c4a0
free ,  @ 0x73c360
free { @ 0x738a50
free [fn 0x738970] @ 0x738970
free [fn 0x738820] @ 0x738820
free [fn 0x73afd0] @ 0x73afd0
free Count must be a non-negative integer. @ 0x73b0f0
free [fn 0x73aea0] @ 0x73aea0
free [fn 0x73ace0] @ 0x73ace0
free ] @ 0x73aba0
free , @ 0x73aa80
free [ @ 0x73a850
free [fn 0x73a7f0] @ 0x73a7f0
free [fn 0x73a700] @ 0x73a700
free [fn 0x73a5e0] @ 0x73a5e0
free [fn 0x73a4d0] @ 0x73a4d0
free [fn 0x73a3c0] @ 0x73a3c0
free [fn 0x73a290] @ 0x73a290
free [fn 0x73a1d0] @ 0x73a1d0
free [fn 0x73a0f0] @ 0x73a0f0
free [fn 0x739f70] @ 0x739f70
free [fn 0x739e60] @ 0x739e60
free [fn 0x739d30] @ 0x739d30
free [fn 0x739be0] @ 0x739be0
free [fn 0x739b20] @ 0x739b20
free [fn 0x7376c0] @ 0x7376c0
free  @ 0x737720
free Count must be a non-negative integer. @ 0x739790
free [fn 0x7375b0] @ 0x7375b0
free [fn 0x7362f0] @ 0x7362f0
free [fn 0x738db0] @ 0x738db0
free [fn 0x738ca0] @ 0x738ca0
free [fn 0x736cc0] @ 0x736cc0
free [fn 0x736bd0] @ 0x736bd0
free [fn 0x735940] @ 0x735940
free [fn 0x738230] @ 0x738230
free [fn 0x738120] @ 0x738120
free [fn 0x738030] @ 0x738030
free [fn 0x737f20] @ 0x737f20
free [fn 0x737bf0] @ 0x737bf0
free  @ 0x737b30
```

异步社区 www.epubit.com.cn

```
free [fn 0x737a90] @ 0x737a90
free  @ 0x737300
free [fn 0x736040] @ 0x736040
free Can't reduce an empty sequence. @ 0x737330
free [fn 0x7371b0] @ 0x7371b0
free [fn 0x736f70] @ 0x736f70
free [fn 0x736df0] @ 0x736df0
free [fn 0x7353b0] @ 0x7353b0
free [fn 0x7368d0] @ 0x7368d0
free [fn 0x7366a0] @ 0x7366a0
free [fn 0x736450] @ 0x736450
free [fn 0x735690] @ 0x735690
free [fn 0x735e70] @ 0x735e70
free [fn 0x735c80] @ 0x735c80
free [fn 0x735580] @ 0x735580
free objectMeta @ 0x734f50
free [class objectMeta 0x734c70] @ 0x734c70
free class @ 0x734c40
free [class class 0x734cc0] @ 0x734cc0
free object @ 0x734b00
free [class object 0x734ab0] @ 0x734ab0
free [module 0x734250] @ 0x734250
free [map 0x734110] @ 0x734110
[work@work b]$
```

异步社区 www.epubit.com.cn