自己动手实现解释器

译自 *Robert Nystrom* 的《*Crafting Interpreters*》

左元

2022

目录

第一部分

欢迎大家

这可能将会是一个伟大旅程的开始。编程语言这个领域有着巨大的探索和玩耍的空间。你可以在这个大房子里尽情的创造，可以把你做的东西分享给别人，也可以仅仅是娱乐自己。很多伟大的计算机科学家和软件工程师将他们一生的精力都投入到了这个领域，而这个领域却还远远没有到头。如果这本书是你踏入这个领域所接触的第一本书，欢迎你！

这本书将带着你在编程语言的世界里旅行一番。但在我们穿好登山靴出去旅行之前，我们需要先熟悉一下我们要旅行的地点的整个地图。这一部分的章节将会带我们学习一些编程语言中用到的基本概念，以及这些概念的组织方式。

我们也会对 Lox 这门语言非常熟悉。因为我们将要在书中剩下的部分实现这门语言（要实现两遍！）。

第一章 简介

> 童话故事是无比真实的：不是因为它告诉我们龙的存在，而是因为它告诉我们龙可以被击败。
>
> ——盖曼

我真的很兴奋我们能一起踏上这段旅程。这是一本关于为编程语言实现解释器的书。它也是一本关于如何设计一种值得实现的语言的书。我刚开始接触编程语言的时候就希望我可以写出这本书，这本书我在脑子里已经写了将近十年了[1]。

在本书中，我们将一步一步地介绍一种功能齐全的语言的两个完整的解释器实现。我假设这是您第一次涉足编程语言，因此我将介绍构建一个完整、可用、快速的语言所需的每个概念和代码。

为了在一本书中塞进两个完整的实现，而且避免这变成一个门槛，本文在理论上比其他文章更轻。在构建系统的每个模块时，我将介绍它背后的历史和概念。我会尽力让您熟悉这些行话，即便您在充满 PL（编程语言）研究人员的鸡尾酒会中，也能快速融入其中。

但我们主要还是要花费精力让这门语言运转起来。这并不是说理论不重要。在学习一门语言时，能够对语法和语义进行精确而公式化的推理是一项至关重要的技能。但是，就我个人而言，我在实践中学习效果最好。对我来说，要深入阅读那些充满抽象概念的段落并真正理解它们太难了。但是，如果我（根据理论）编写了代码，运行并调试完成，那么我就明白了。

这就是我对您的期望。我想让你们直观地理解一门真正的语言是如何生活和呼吸的。我的希望是，当你以后阅读其他理论性更强的书籍时，这些概念会牢牢地留在你的脑海中，依附于这个有形的基础之上。

1.1 为什么要学习这些东西？

每一本编译器相关书籍的前言似乎都有这一节。我不知道为什么编程语言会引起这种存在性的怀疑。我认为鸟类学书籍作者不会担心证明它们的存在。他们假设读者喜欢鸟，然后就开始讲授内容。

但是编程语言有一点不同。我认为，对我们中的任何一个人来说，能够创建一种广泛成功的通用编程语言的可能性都很小，这是事实。设计世界通用语言的设计师们，一辆汽车就能装得下。如果加入这个精英群体是学习语言的唯一原因，那么就很难证明其合理性。幸运的是，事实并非如此。

[1]这里要和我的家人和朋友们说声抱歉，抱歉这些年我是如此的不着调，如此的心不在焉（脑子里一直在写书）！

1.1.1 小型编程语言无处不在

对于每一种成功的通用语言，都有上千种成功的小众语言。我们过去称它们为“小语言”，但术语泛滥的今天它们有了“领域特定语言（即 DSL）”的名称。这些是为特定任务量身定做的洋泾浜语言，如应用程序脚本语言、模板引擎、标记格式和配置文件。

几乎每个大型软件项目都需要一些这样的工具。如果可以的话，最好重用现有的工具，而不是自己动手实现。一旦考虑到文档、调试器、编辑器支持、语法高亮显示和所有其他可能的障碍，自己实现就成了一项艰巨的任务。

但是，当现有的库不能满足您的需要时，您仍然很有可能发现自己需要一个解析器或其他东西。即使当您重用一些现有的实现时，您也不可避免地需要调试和维护，并在其内部进行探索。

1.1.2 自己实现编程语言是一种很好的锻炼

长跑运动员有时会在脚踝上绑上重物，或者在空气稀薄的高海拔地区进行训练。当他们卸下自己的负担以后，轻便的肢体和富氧的空气带来了新的相对舒适度，使它们可以跑得更快，更远。

实现一门语言是对编程技能的真正考验。代码很复杂，而性能很关键。您必须掌握递归、动态数组、树、图和哈希表。您在日常编程中至少使用过哈希表，但您对它们的理解程度有多高呢？嗯，等我们从头完成我们的作品之后，我相信您会理解的。

虽然我想说明解释器并不像您想的那样令人生畏，但实现一个好的解释器仍然是一个挑战。学会了它，您就会成为一个更强大的程序员，并且在日常工作中也能更加聪明地使用数据结构和算法。

1.1.3 一个额外的原因

这最后一个原因我很难承认，因为它是很私密的理由。自从我小时候学会编程以来，我就觉得语言有种神奇的力量。当我第一次一个键一个键地输入 BASIC 程序时，我无法想象 BASIC 语言本身是如何制作出来的。

后来，当我的大学朋友们谈论他们的编译器课程时，脸上那种既敬畏又恐惧的表情足以让我相信，语言黑客是另一种人，某种获得了通向神秘艺术的特权的巫师。

这是一个迷人的形象，但它也有黑暗的一面。我感觉自己不像个巫师，所以我认为自己缺乏加入秘社所需的先天品质。尽管自从我在学校笔记本上拼写关键词以来，我一直对语言着迷，但我花了数十年的时间鼓起勇气尝试真正地学习它们。那种“神奇”的品质，那种排他性的感觉，将我挡在门外。

当我最终开始拼凑我自己的小编译器时，我很快意识到，根本就没有魔法。它只是代码，而那些掌握语言的人也只是人。

有一些技巧您在语言之外不会经常遇到，而且有些部分有点难。但不会比您克服的其他障碍更困难。我希望，如果您对语言感到害怕，而这本书能帮助您克服这种恐惧，也许我会让您比以前更勇敢一点。

而且，说不准，你也许会创造出下一个伟大的语言，毕竟总要有人做。

1.2 本书的组织方式

这本书分为三个部分。您现在正在读的是第一部分。这部分用了几章来让您进入状态，教您一些语言黑客使用的行话，并向您介绍我们将要实现的语言 Lox。

其他两个部分则分别构建一个完整的 Lox 解释器。在这些部分中，每个章节的结构都是相同的。每一章节挑选一个语言功能点，教您背后对应的概念，并逐步介绍实现方法。

我花了不少时间去试错，但我还是成功地把这两个解释器按照章节分成了一些小块，每一小块的内容都会建立在前面几章的基础上，但不需要后续章节的知识。从第一章开始，你就会有一个可以运行和使用的工作程序。随着章节的推移，它的功能越来越丰富，直到你最终拥有一门完整的语言。

除了大量妙趣横生的英文段落，章节中还会包含一些其它的惊喜：

1.2.1 代码

本书是关于制作解释器的，所以其中会包含真正的代码。所需要的每一行代码都需要包含在内，而且每个代码片段都会告知您需要插入到实现代码中的什么位置。

许多其他的语言书籍和语言实现都使用 Lex 和 Yacc 这样的工具，也就是所谓的**编译器-编译器**，可以从一些更高层次的（语法）描述中自动生成一些实现的源文件。这些工具有利有弊，而且双方都有强烈的主张-有些人可能将其说成是信仰。

我们这里不会使用这些工具。我想确保魔法和困惑不会藏在黑暗的角落，所以我们会选择手写所有代码。正如您将看到的，这并没有听起来那么糟糕，因为这意味着您将真正理解每一行代码以及两种解释器的工作方式。

一本书和“真实世界”的条件是有区别的，因此这里的代码风格可能并不是可维护生产软件的最佳方式。可能我对某些写法是无所谓的，比如省略 *private* 或者声明全局变量，请理解我这样做是为了让您更容易看懂代码。书页不像 IDE 窗口那么宽，所以每一个字符都很珍贵。

另外，代码也不会有太多的注释。这是因为每一部分代码前后，都使用了一些真的很简洁的文字来对其进行解释。当你写一本书来配合你的程序时，欢迎你也省略注释。否则，你可能应该比我使用更多的//。

虽然这本书包含了每一行代码，并教授了每一行代码的含义，但它没有描述编译和运行解释器所需的机制。我假设您可以在 IDE 中选择一个 makefile 或一个项目导入，以使代码运行。这类说明很快就会过时，我希望这本书能像 XO 白兰地一样醇久，而不是像家酿酒（一样易过期）。

1.2.2 代码片段

因为这本书包含了实现所需的每一行代码，所以代码片段相当精确。此外，即使是在缺少主要功能的时候，我也尝试将程序保持在可运行状态。因此我们有时会添加临时代码，这些代码将在以后的代码段中替换。

一个完整的代码片段可能如下所示：

lox/Scanner.java，在 scanToken() 中替换 1 行

```
default:
  if (isDigit(c)) {
    number();
  } else {
    Lox.error(line, "Unexpected character.");
  }
  break;
```

中间是要添加的新代码。这部分代码的上面或下面可能有一些淡出的行，以显示它在周围代码中的位置。还会附有一小段介绍，告诉您在哪个文件中以及在哪里放置代码片段。如果简介说要 “replace x lines”，表明在浅色的行之间有一些现有的代码需要删除，并替换为新的代码片段。

1.2.3 题外话

题外话中包含传记简介、历史背景、对相关主题的引用以及对其他要探索的领域的建议。您无需深入了解就可以理解本书的后续部分，因此可以根据需要跳过它们。我不会批评你，但我可能会有些难过。【注：由于排版原因，在翻译的时候，将旁白信息作为脚注附在章节之后】

1.2.4 挑战

每章结尾都会有一些练习题。不像教科书中的习题集那样用于回顾已讲述的内容，这些习题是为了帮助您学习更多的知识，而不仅仅是本章中的内容。它们会迫使您走出文章指出的路线，自行探索。它们将要求您研究其他语言，弄清楚如何实现功能，换句话说，就是使您走出舒适区。

克服挑战，您将获得更广泛的理解，也可能遇到一些挫折。如果您想留在旅游巴士的舒适区内，也可以跳过它们。都随你便。

1.2.5 设计笔记

大多数编程语言书籍都是严格意义上的编程语言实现书籍。他们很少讨论如何设计正在实现的语言。实现之所以有趣，是因为它的定义是很精确的。我们程序员似乎很喜欢黑白、1 和 0 这样的事物。

就个人而言，我认为世界只需要这么多的 FORTRAN 77 实现。在某个时候，您会发现自己正在设计一种新的语言。一旦开始这样做，方程式中较柔和，人性化的一面就变得至关重要。诸如哪些功能易于学习，如何在创新和熟悉度之间取得平衡，哪种语法更易读以及对谁有帮助。

所有这些都会对您的新语言的成功产生深远的影响。我希望您的语言取得成功，因此在某些章节中，我以一篇“设计笔记”结尾，这些是关于编程语言的人文方面的一些文章。我并不是这方面的专家——我不确定是否有人真的精通这些，因此，请您在阅读这些文字的时候仔细评估。这样的话，这些文字就能成为您思考的食材，这也正是我的目标。

1.3 第一个解释器

我们将用 Java 编写第一个解释器 jlox。（这里的）主要关注点是概念。我们将编写最简单，最干净的代码，以正确实现该语言的语义。这样能够帮助我们熟悉基本技术，并磨练对语言表现形式的确切理解。

Java 是一门很适合这种场景的语言。它的级别足够高，我们不会被繁琐的实现细节淹没，但代码仍是非常明确的。与脚本语言不同的是，它的底层没有隐藏太过复杂的机制，你可以使用静态类型来查看正在处理的数据结构。

我选择 Java 还有特别的原因，就是因为它是一种面向对象的语言。这种范式在 90 年代席卷了整个编程世界，如今已成为数百万程序员的主流思维方式。很有可能您已经习惯了将代码组织到类和方法中，因此我们将让您在舒适的环境中学习。

虽然学术语言专家有时瞧不起面向对象语言，但事实上，它们即使在语言工作中也被广泛使用。GCC 和 LLVM 是用 c++ 编写的，大多数 JavaScript 虚拟机也是这样。面向对象的语言无处不在，并且针对该语言的工具和编译器通常是用同一种语言编写的。

最后，Java 非常流行。这意味着您很有可能已经了解它了，所以你要学习的东西就更少了。如果您不太熟悉 Java，也请不要担心。我尽量只使用它的最小子集。我使用 Java 7 中的菱形运算符使代码看起来更简洁，但就“高级”功能而言，仅此而已。如果您了解其它面向对象的语言（例如 C ＃或 C ++），就没有问题。

在第二部分结束时，我们将得到一个简单易读的实现。但是我们得到的不会是一个快速的解释器。它还是利用了 Java 虚拟机自身的运行时工具。我们想要学习 Java 本身是如何实现这些东西的。

1.4 第二个解释器

所以在下一部分，我们将从头开始，但这一次是用 C 语言。C 语言是理解实现编译器工作方式的完美语言，一直到内存中的字节和流经 CPU 的代码。

我们使用 C 语言的一个重要原因是，我可以向您展示 C 语言特别擅长的东西，但这并不意味着您需要非常熟练地使用它。您不必是丹尼斯 · 里奇（Dennis Ritchie）的转世，

但也不应被指针吓倒。

如果你（对 C 的掌握）还没到那一步，找一本关于 C 的入门书，仔细阅读，读完后再回来。作为回报，从这本书中你将成为一个更优秀的 C 程序员。可以想想有多少语言实现是用 C 完成的: Lua、CPython 和 Ruby 的 MRI 等，这里仅举几例。

在我们的 C 解释器 clox 中，我们不得不自己实现那些 Java 免费提供给我们的东西。我们将编写自己的动态数组和哈希表。我们将决定对象在内存中的表示方式，并构建一个垃圾回收器来回收它。

我们的 Java 版实现专注于正确性。既然我们已经完成了，那么我们就要变得越来越快。我们的 C 解释器将包含一个编译器，该编译器会将 Lox 转换为有效的字节码形式（不用担心，我很快就会讲解这是什么意思）之后它会执行对应的字节码。这与 Lua，Python，Ruby，PHP 和许多其它成功语言的实现所使用的技术相同。

我们甚至会尝试进行基准测试和优化。到最后，我们将为 lox 语言提供一个强大，准确，快速的解释器，并能够不落后于其他专业水平的实现。对于一本书和几千行代码来说已经不错了。

1.5 挑战

1. 在我编写的这个小系统中，至少有六种特定领域语言（DSL），它们是什么?
2. 使用 Java 编写并运行一个“Hello, world!”程序，设置你需要的 makefile 或 IDE 项目使其正常工作。如果您有调试器，请先熟悉一下，并在程序运行时对代码逐步调试。
3. 对 C 也进行同样的操作。为了练习使用指针，可以定义一个堆分配字符串的双向链表。编写函数以插入，查找和删除其中的项目。测试编写的函数。

1.6 设计笔记：名字是什么？

写这本书最困难的挑战之一是为它所实现的语言取个名字。我翻了好几页的备选名才找到一个合适的。当你某一天开始构建自己的语言时，你就会发现命名是非常困难的。一个好名字要满足几个标准：

1. **尚未使用**。如果您不小心使用了别人的名字，就可能会遇到各种法律和社会上的麻烦。
2. **容易发音**。如果一切顺利，将会有很多人会说和写您的语言名称。超过几个音节或几个字母的任何内容都会使他们陷入无休止的烦恼。
3. **足够独特，易于搜索**。人们会 Google 你的语言的名字来了解它，所以你需要一个足够独特的单词，以便大多数搜索结果都会指向你的文档。不过，随着人工智能搜索引擎数量的增加，这已经不是什么大问题了。但是，如果您将语言命名为“for”，那对用户基本不会有任何帮助。
4. **在多种文化中，都没有负面的含义**。这很难防范，但是值得深思。Nimrod 的设计师

最终将其语言重命名为“Nim”，因为太多的人只记得 Bugs Bunny 使用“Nimrod”作为一种侮辱（其实是讽刺）。

如果你潜在的名字通过了考验，就保留它吧。不要纠结于寻找一个能够抓住你语言精髓的名称。如果说世界上其他成功的语言的名字教会了我们什么的话，那就是名字并不重要。您所需要的只是一个相当独特的标记。

第二章 全书地图

> 你必须要有一张地图，无论它是多么粗糙。否则你就会到处乱逛。在《指环王》中，我从未让任何人在某一天走得超出他力所能及的范围。
>
> ——托尔金

我们不想到处乱逛，所以在我们开始之前，让我们先浏览一下以前的语言实现者所绘制的领土。它能帮助我们了解我们的目的地和其他人采用的备选路线。

首先，我先做个简单说明。本书的大部分内容都是关于语言的实现，它与语言本身这种柏拉图式的理想形式有所不同。诸如“堆栈”，“字节码”和“递归下降”之类的东西是某个特定实现中可能使用的基本要素。从用户的角度来说，只要最终产生的装置能够忠实地遵循语言规范，它内部的都是实现细节。

我们将会花很多时间在这些细节上，所以如果我每次提及的时候都写“语言实现”，我的手指都会被磨掉。相反，除非有重要的区别，否则我将使用“语言”来指代一种语言或该语言的一种实现，或两者皆有。

2.1 语言的各部分

自计算机的黑暗时代以来，工程师们就一直在构建编程语言。当我们可以和计算机对话的时候，我们发现这样做太难了，于是我们寻求电脑的帮助。我觉得很有趣的是，即使今天的机器确实快了一百万倍，存储空间也大了几个数量级，但我们构建编程语言的方式几乎没有改变。

尽管语言设计师所探索的领域辽阔，但他们所走过的路却很少。并非每种语言都采用完全相同的路径（有些采用一种或两种捷径），但除此之外，从海军少将 Grace Hopper 的第一个 COBOL 编译器，一直到一些热门的新移植到 JavaScript 的语言，它们的“文档”完全是由 Git 仓库中一个编辑得很差的 README 组成的。

我把一个语言实现可能选择的路径网络类比为爬山。你从最底层开始，程序是原始的源文本，实际上只是一串字符。每个阶段都会对程序进行分析，并将其转换为更高层次的表现形式，从而使语义（作者希望计算机做什么）变得更加明显。

最终我们达到了峰顶。我们可以鸟瞰用户的程序，可以看到他们的代码含义是什么。我们开始从山的另一边下山。我们将这个最高级的表示形式转化为连续的较低级别的形式，从而越来越接近我们所知道的如何让 CPU 真正执行的形式。

让我们沿着每一条路线和每一个感兴趣的地方走一遍。我们的旅程从左边的用户源代码的纯文本开始。

2.1.1 扫描

第一步是**扫描**，也就是所谓的**词法**，或者说（如果你想给别人留下深刻印象）**词法分析**。它们的意思都差不多。我喜欢“lexing”，因为这听起来像是一个邪恶的超级大坏蛋会做的事情，但我还是用“scanning”，因为它似乎更常见一些。

扫描器 (或词法解析器) 接收线性字符流，并将它们组合成一系列更类似于“单词”的东西。在编程语言中，这些词的每一个都被称为**标记**。有些标记是单个字符，比如 (和,。其他的可能是几个字符长的，比如数字（123）、字符串字元（"hi!"）和标识符（min）。

源文件中的一些字符实际上没有任何意义。空格通常是无关紧要的，而注释，从定义就能看出来，会被语言忽略。扫描仪通常会丢弃这些字符，留下一个干净的有意义的标记序列。

2.1.2 语法分析

下一步是**解析**。这就是我们从句法中得到**语法**的地方——语法能够将较小的部分组成较大的表达式和语句。你在英语课上画过句子图吗？如果有，你就做了解析器所做的事情，区别在于，英语中有成千上万的“关键字”和大量的歧义，而编程语言要简单得多。

解析器接受标记的平面序列，并构建反映语法嵌套本质的树结构。这些树有两个不同的名称：**解析树**或**抽象语法树**，这取决于它们与源语言的语法结构有多接近。在实践中，语言黑客通常称它们为“**语法树**”、“**AST**”，或者干脆直接说“**树**”。

解析在计算机科学中有着悠久而丰富的历史，它与人工智能界有着密切的联系。今天用于解析编程语言的许多技术最初是由人工智能研究人员设想的，他们试图让计算机与我们对话，以解析人类语言。

事实证明，相对那些解析器能够处理的严格语法来说，人类语言太混乱了；但对于编程语言中更简单的人工语法来说，人类语言却是完美的。唉，可惜我们这些有缺陷的人类仍然会错误地使用这些简单的语法，因此解析器的工作还包括通过报告**语法错误**让我们知道出错了。

2.1.3 静态分析

在所有实现中，前两个阶段都非常相似。现在，每种语言的个性化特征开始发挥作用。至此，我们知道了代码的语法结构（诸如哪些表达式嵌套在其他表达式中）之类的东西，但是我们知道的也就仅限于此了。

在 a+b 这样的表达式中，我们知道我们要把 a 和 b 相加，但我们不知道这些名字指的是什么。它们是局部变量吗？全局变量？它们在哪里被定义？

大多数语言所做的第一点分析叫做**绑定**或**解析**。对于每一个**标识符**，我们都要找出定义该名称的地方，并将两者连接起来。这就是**作用域**的作用—在这个源代码区域中，某个名字可以用来引用某个声明。

如果语言是静态类型的，这时我们就进行类型检查。一旦我们知道了 a 和 b 的声明位置，我们也可以弄清楚它们的类型。然后如果这些类型不支持互相累加，我们就会报告一个**类型错误**。

深吸一口气。我们已经到达了山顶，并对用户的程序有了全面的了解。所有这些从分析中可见的语义信息都需要存储在某个地方。我们可以把它藏在几个地方：

- 通常，它会被直接存储在语法树本身的**属性**中—属性是节点中的额外字段，这些字段在解析时不会初始化，但在稍后会进行填充。
- 有时，我们可能会将数据存储在外部的查找表中。通常，该表的关键字是标识符，即变量和声明的名称。在这种情况下，我们称其为**符号表**，并且其中与每个键关联的值告诉我们该标识符所指的是什么。
- 最强大的记录工具是将树转化为一个全新的数据结构，更直接地表达代码的语义。这是下一节的内容。

到目前为止，所有内容都被视为实现的**前端**。您可能会猜至此以后是**后端**，其实并不是。在过去的年代，当“前端”和“后端”被创造出来时，编译器要简单得多。后来，研究人员在两个半部之间引入了新阶段。威廉 · 沃尔夫（William Wulf）和他的同伴没有放弃旧术语，而是新添加了一个迷人但有点自相矛盾的名称“**中端**”。

2.1.4 中间表示

你可以把编译器看成是一条流水线，每个阶段的工作是把代表用户代码的数据组织起来，使下一阶段的实现更加简单。管道的前端是针对程序所使用的源语言编写的。后端关注的是程序运行的最终架构。

在中间阶段，代码可能被存储在一些**中间表示**（**intermediate representation**，也叫**IR**）中，这些中间表示与源文件或目标文件形式都没有紧密的联系（因此叫作“中间”）。相反，IR 充当了这两种语言之间的接口。

这可以让你更轻松地支持多种源语言和目标平台。假设你想实现 Pascal、C 和 Fortran 编译器，并且你的目标平台的体系结构是：x86、ARM，还有 SPARC。通常情况下，这意味着你需要写九个完整的编译器：Pascal → x86，C → ARM，以及其他各种组合。

一个共享的中间表示可以大大减少这种情况。你为每个产生 IR 的源语言写一个前端。然后为每个目标平台写一个后端。现在，你可以将这些混搭起来，得到每一种组合。

还有一个重要的原因是，我们可能希望将代码转化为某种形式，使语义更加明确.....。

2.1.5 优化

一旦我们理解了用户程序的含义，我们就可以自由地用另一个具有相同语义但实现效率更高的程序来交换它—我们可以对它进行**优化**。

一个简单的例子是**常量折叠**：如果某个表达式求值得到的始终是完全相同的值，我们可以在编译时进行求值，并用其结果替换该表达式的代码。如果用户输入：

```
pennyArea = 3.14159 * (0.75 / 2) * (0.75 / 2);
```

我们可以在编译器中完成所有的算术运算，并将代码更改为：

```
pennyArea = 0.4417860938;
```

优化是编程语言业务的重要组成部分。许多语言黑客把他们的整个职业生涯都花在了这里，竭尽所能地从他们的编译器中挤出每一点性能，以使他们的基准测试速度提高一个百分点。这可能会成为一种困扰。

我们通常会跳过本书中的棘手问题。许多成功的语言令人惊讶地很少进行编译期优化。例如，Lua 和 CPython 生成相对未优化的代码，并将其大部分性能优化工作集中在运行时上。

2.1.6 代码生成

我们已经将所有可以想到的优化应用到了用户程序中。最后一步是将其转换为机器可以实际运行的形式。换句话说，**生成代码**（或**代码生成**），这里的“代码”通常是指 CPU 运行的类似于汇编的原始指令，而不是人类可能想要阅读的“源代码”。

最后，我们到了**后端**，从山的另一侧开始向下。从现在开始，随着我们越来越接近于思维简单的机器可以理解的东西，我们对代码的表示变得越来越原始，就像逆向进化。

我们需要做一个决定。我们是为真实 CPU 还是虚拟 CPU 生成指令？如果我们生成真实的机器代码，则会得到一个可执行文件，操作系统可以将其直接加载到芯片上。原生代码快如闪电，但生成它需要大量工作。当今的体系结构包含大量指令，复杂的流水线和足够塞满一架 747 行李舱的历史包袱。

使用芯片的语言也意味着你的编译器是与特定的架构相绑定的。如果你的编译器以 x86 机器代码为目标，那么它就无法在 ARM 设备上运行。一直到 60 年代，在计算机体系结构的寒武纪大爆发期间，这种缺乏可移植性的情况是一个真正的障碍。

为了解决这个问题，像 BCPL 的 Martin Richards 和 Pascal 的 Niklaus Wirth 这样的黑客，让他们的编译器生成虚拟机代码。他们不是为真正的芯片编写指令，而是为一个假设的、理想化的机器编写代码。Wirth 称这种 **p-code** 为“可移植代码”，但今天，我们通常称它为**字节码**，因为每条指令通常都是一个字节长。

这些合成指令的设计是为了更紧密地映射到语言的语义上，而不必与任何一个计算机体系结构的特性和它积累的历史错误绑定在一起。你可以把它想象成语言底层操作的密集二进制编码。

2.1.7 虚拟机

如果你的编译器产生了字节码，你的工作还没有结束。因为没有芯片可以解析这些字节码，因此你还需要进行翻译。同样，您有两个选择。您可以为每个目标体系结构编写一个小型编译器，将字节码转换为该机器的本机代码。您仍然需要针对您支持的每个芯片做一些工作，但最后这个阶段非常简单，你可以在你支持的所有机器上重复使用编译器管道的其余部分。你基本上是把你的字节码作为一个中介码。

或者，您可以编写**虚拟机（VM）**，该程序可在运行时模拟支持虚拟架构的虚拟芯片。在虚拟机中运行字节码比提前将其翻译成本地代码要慢，因为每条指令每次执行时都必须在运行时模拟。作为回报，你得到的是简单性和可移植性。用比如说 C 语言实现你的虚拟机，你就可以在任何有 C 编译器的平台上运行你的语言。这就是我们在本书中构建的第二个解释器的工作原理。

2.1.8 运行时

我们终于将用户程序锤炼成可以执行的形式。最后一步是运行它。如果我们将其编译为机器码，我们只需告诉操作系统加载可执行文件，然后就可以运行了。如果我们将它编译成字节码，我们需要启动 VM 并将程序加载到其中。

在这两种情况下，除了最基本的底层语言外，我们通常需要我们的语言在程序运行时提供一些服务。例如，如果语言自动管理内存，我们需要一个垃圾收集器去回收未使用的比特位。如果我们的语言支持 instance of 测试，这样你就可以看到你有什么类型的对象，那么我们就需要一些表示方法来跟踪执行过程中每个对象的类型。

所有这些东西都是在运行时进行的，所以它被恰当地称为，**运行时**。在一个完全编译的语言中，实现运行时的代码会直接插入到生成的可执行文件中。比如说，在 Go 中，每个编译后的应用程序都有自己的一份 Go 的运行时副本直接嵌入其中。如果语言是在解释器或虚拟机内运行，那么运行时将驻留于虚拟机中。这也就是 Java、Python 和 JavaScript 等大多数语言实现的工作方式。

2.2 捷径和备选路线

这是一条漫长的道路，涵盖了你要实现的每个可能的阶段。许多语言的确走完了整条路线，但也有一些捷径和备选路径。

2.2.1 单遍编译器

一些简单的编译器将解析、分析和代码生成交织在一起，这样它们就可以直接在解析器中生成输出代码，而无需分配任何语法树或其他 IR。这些**单遍编译器**限制了语言的设计。您没有中间数据结构来存储程序的全局信息，也不会重新访问任何之前解析过的代码部分。这意味着，一旦您看到某个表达式，就需要足够的知识来正确地对其进行编译。

Pascal 和 C 语言就是围绕这个限制而设计的。在当时，内存非常珍贵，一个编译器可能连整个源文件都无法存放在内存中，更不用说整个程序了。这也是为什么 Pascal 的语法要求类型声明要先出现在一个块中。这也是为什么在 C 语言中，你不能在定义函数的代码上面调用函数，除非你有一个明确的前向声明，告诉编译器它需要知道什么，以便生成调用后面函数的代码。

2.2.2 树遍历解释器

有些编程语言在将代码解析为 AST 后就开始执行代码（可能应用了一点静态分析）。为了运行程序，解释器每次都会遍历语法树的一个分支和叶子，并在运行过程中计算每个节点。

这种实现风格在学生项目和小型语言中很常见，但在通用语言中并不广泛使用，因为它往往很慢。有些人使用“解释器”仅指这类实现，但其他人对“解释器”一词的定义更宽泛，因此我将使用没有歧义的“**树遍历解释器**”来指代这些实现。我们的第一个解释器就是这样工作的。

2.2.3 转译器

为一种语言编写一个完整的后端可能需要大量的工作。如果您有一些现有的通用 IR 作为目标，则可以将前端转换到该 IR 上。否则，您可能会陷入困境。但是，如果您将某些其他*源语言*视为中间表示，该怎么办？

您需要为您的语言编写一个前端。然后，在后端，您可以生成一份与您的语言级别差不多的其他语言的有效源代码字符串，而不是将所有代码降低到某个原始目标语言的语义。然后，您可以使用该语言现有的编译工具作为逃离大山的路径，得到某些可执行的内容。

人们过去称之为**源到源编译器**或**转换编译器**。随着那些为了在浏览器中运行而编译成 JavaScript 的各类语言的兴起，它们有了一个时髦的名字—**转译器**。

虽然第一个编译器是将一种汇编语言翻译成另一种汇编语言，但现今，大多数编译器都适用于高级语言。在 UNIX 病毒式地传播到各种各样的机器之后，开始了一个悠久的编译器传统，即编译器以 C 作为输出语言。只要 UNIX 存在，就可以使用 C 编译器，并生成有效的代码，因此，以 C 为目标是让语言在许多体系结构上运行的好方法。

Web 浏览器是今天的“机器”，它们的“机器代码”是 JavaScript，所以现在似乎几乎所有的语言都有一个以 JS 为目标的编译器，因为这是让你的代码在浏览器中运行的主要方式。

编译器的前端（扫描器和解析器）看起来跟其他编译器相似。然后，如果源语言只是在目标语言之上包装的简单语法外壳，则它可能会完全跳过分析，并直接输出目标语言中的类似语法。

如果两种语言的语义差异较大，那么你就会看到完整编译器的更多典型阶段，包括分析甚至优化。然后，在代码生成阶段，无需输出一些像机器代码一样的二进制语言，而是在目标语言中生成一串语法正确的源码（好吧，目标代码）。

不管是哪种方式，你再通过目标语言已有的编译管道运行生成的代码，就可以了。

2.2.4 即时编译

最后一个与其说是捷径，不如说是危险的高山争霸赛，最好留给专家。执行代码最快的方法是将代码编译成机器代码，但你可能不知道你的最终用户的机器支持什么架构。

该怎么做呢?

您可以做和 HotSpot JVM、Microsoft 的 CLR 和大多数 JavaScript 解释器相同的事情。在终端用户的机器上，当程序加载时（无论是从 JS 中还是从源代码加载，或者是 JVM 和 CLR 的平台无关的字节码），都可以将其编译为对应的本地代码，以适应本机支持的体系结构。自然地，这被称为**即时编译**。大多数黑客只是说 “JIT”，其发音与 “fit” 押韵。

最复杂的 JIT 将性能分析钩子插入到生成的代码中，以查看哪些区域对性能最为关键，以及哪些类型的数据正在流经其中。然后，随着时间的推移，它们将通过更高级的优化功能自动重新编译那些热点部分。

2.3 编译器和解释器

现在我已经向你的脑袋里塞满了一大堆编程语言术语，我们终于可以解决一个自远古以来一直困扰着程序员的问题: 编译器和解释器之间有什么区别?

事实证明，这就像问水果和蔬菜的区别一样。这看上去似乎是一个非此即彼的选择，但实际上 “水果” 是一个植物学术语，“蔬菜” 是烹饪学术语。严格来说，一个并不意味着对另一个的否定。有不是蔬菜的水果（苹果），也有不是水果的蔬菜（胡萝卜），也有既是水果又是蔬菜的可食用植物，比如西红柿。

好，回到语言上:

- **编译**是一种实现技术，其中涉及到将源语言翻译成其他语言-通常是较低级的形式。当你生成字节码或机器代码时，你就是在编译。当你移植到另一种高级语言时，你也在编译。
- 当我们说语言实现 “是**编译器**” 时，是指它会将源代码转换为其他形式，但不会执行。用户必须获取结果输出并自己运行。
- 相反，当我们说一个实现 “是一个**解释器**” 时，是指它接受源代码并立即执行它。它 “从源代码” 运行程序。

像苹果和橘子一样，某些实现显然是编译器，而不是解释器。GCC 和 Clang 接受您的 C 代码并将其编译为机器代码。最终用户直接运行该可执行文件，甚至可能永远都不知道使用了哪个工具来编译它。所以这些是 C 的*编译器*。

在 Matz 的旧版本的 Ruby 规范实现中，用户从源代码中运行 Ruby。该实现通过遍历语法树对其进行解析并直接执行。期间都没有发生其他的转换，无论是在实现内部还是以任何用户可见的形式。所以这绝对是一个 Ruby 的*解释器*。

但是 CPython 呢? 当你使用它运行你的 Python 程序时，代码会被解析并转换为内部字节码格式，然后在虚拟机内部执行。从用户的角度来看，这显然是一个解释器-他们是从源代码开始运行自己的程序。但如果你看一下 CPython 的内部，你会发现肯定有一些编译工作在进行。

答案是两者兼而有之。CPython 是一个解释器，但他也有一个编译器。实际上，大多数脚本语言都以这种方式工作，如您所见:

中间那个重叠的区域也是我们第二个解释器所在的位置，因为它会在内部编译成字

节码。所以，虽然本书名义上是关于解释器的，但我们也会涉及一些编译的内容。

2.4 我们的旅程

一下子有太多东西要消化掉。别担心。这一章并不是要求你理解所有这些零碎的内容。我只是想让你们知道它们是存在的，以及大致了解它们是如何组合在一起的。

当您探索本书本书所指导的路径之外的领域时，这张地图应该对你很有用。我希望你自己出击，在那座山里到处游走。

但是，现在，是我们自己的旅程开始的时候了。系好你的鞋带，背好你的包，走吧。从这里开始，你需要关注的是你面前的路。

2.5 挑战

1. 选择一个你喜欢的语言的开源实现。下载源代码，并在其中探索。试着找到实现扫描器和解析器的代码，它们是手写的，还是用 Lex 和 Yacc 等工具生成的？（存在.l 或.y 文件通常意味着后者）
2. 实时编译往往是实现动态类型语言最快的方法，但并不是所有的语言都使用它。有什么理由不采用 JIT 呢？
3. 大多数可编译为 C 的 Lisp 实现也包含一个解释器，该解释器还使它们能够即时执行 Lisp 代码。为什么？

第二部分

树遍历解释器

第三部分

字节码解释器

> 如果你发现你几乎把所有的时间都花在了理论上，那就开始把一些注意力转向实际的东西；这会提高你的理论水平。如果你发现你几乎把所有的时间都花在了实践上，那就开始把一些注意力转向理论上的东西；这将改善你的实践。
>
> ——高德纳

我们已经有了一个 Lox 的完整实现 jlox，那么为什么这本书还没有结束呢？部分原因是 jlox 依赖 JVM 为我们做很多事情。如果我们想要了解一个解释器是如何工作的，我们就需要自己构建这些零碎的东西。

jlox 不够用的一个更根本的原因在于，它太慢了。树遍历解释器对于某些高级的声明式语言来说是不错的，但是对于通用的命令式语言—即使是 Lox 这样的“脚本”语言—这是行不通的。以下面的小脚本为例：

```
fun fib(n) {
  if (n < 2) return n;
  return fib(n - 1) + fib(n - 2);
}

var before = clock();
print(fib(40));
var after = clock();
print(after - before);
```

在我们的笔记本电脑上，jlox 大概需要 72 秒的时间来执行。一个等价的 C 程序在半秒内可以完成。我们的动态类型的脚本语言永远不可能像手动管理内存的静态类型语言那样快，但我们没必要满足于慢两个数量级以上的速度。

我们可以把 jlox 放在性能分析器中运行，并进行调优和调整热点，但这也只能到此为止了。它的执行模型（遍历 AST）从根本上说就是一个错误的设计。我们无法将其微优化到我们想要的性能，就像你无法将 AMC Gremlin 打磨成 SR-71 Blackbird 一样。

我们需要重新考虑核心模型。本章将介绍这个模型—字节码，并开始我们的新解释器，clox。

第三章 字节码块

3.1 字节码？

在工程领域，很少有选择是不需要权衡的。为了更好地理解我们为什么要使用字节码，让我们将它与几个备选方案进行比较。

3.1.1 为什么不遍历抽象语法树？

我们目前的解释器有几个优点：

- 嗯，首先我们已经写好了，它已经完成了。它能完成的主要原因是这种风格的解释器实现起来非常简单。代码的运行时表示直接映射到语法。从解析器到我们在运行时需要的数据结构，几乎都毫不费力。
- 它是可移植的。我们目前的解释器是使用 Java 编写的，可以在 Java 支持的任何平台上运行。我们可以用同样的方法在 C 语言中编写一个新的实现，并在世界上几乎所有平台上编译并运行我们的语言。

这些是真正的优势。但是，另一方面，它的内存使用效率不高。每一段语法都会变成一个 AST 节点。像 1+2 这样的 Lox 表达式会变成一连串的对象，对象之间有很多指针，就像：

每个指针都会给对象增加 32 或 64 比特的开销。更糟糕的是，将我们的数据散布在一个松散连接的对象网络中的堆上，会对空间局部性造成影响。

现代 CPU 处理数据的速度远远超过它们从 RAM 中提取数据的速度。为了弥补这一点，芯片中有多层缓存。如果它需要的一块存储数据已经在缓存中，它就可以更快地被加载。我们谈论的是 100 倍以上的提速。

数据是如何进入缓存的？机器会推测性地为你把数据塞进去。它的启发式方法很简单。每当 CPU 从 RAM 中读取数据时，它就会拉取一块相邻的字节并放到缓存中。

如果我们的程序接下来请求一些在缓存行中的数据，那么我们的 CPU 就能像工厂里一条运转良好的传送带一样运行。我们真的很想利用这一点。为了有效的利用缓存，我们在内存中表示代码的方式应该像读取时一样紧密而有序。

现在抬头看看那棵树。这些子对象可能在任何地方。树遍历器的每一步都会引用子节点，都可能会超出缓存的范围，并迫使 CPU 暂停，直到从 RAM 中拉取到新的数据块（才会继续执行）。仅仅是这些树形节点及其所有指针字段和对象头的开销，就会把对象彼此推离，并将其推出缓存区。

我们的 AST 遍历器在反射等方面还有其它开销，但仅仅是局部性问题就足以证明使用更好的代码表示是合理的。

3.1.2 为什么不编译成本地机器的汇编代码？

如果你想真正快，就要摆脱所有的中间层，一直到最底层—机器码。听起来就很快，机器码。

最快的语言所做的是直接把代码编译为芯片支持的本地指令集。从早期工程师真正用机器码手写程序以来，以本地代码为目标一直是最有效的选择。

如果你以前从来没有写过任何机器码，或者是它略微讨人喜欢的近亲汇编语言，那我给你做一个简单的介绍。本地代码是一系列密集的操作，直接用二进制编码。每条指令的长度都在一到几个字节之间，而且几乎是令人头疼的底层指令。“将一个值从这个地址移动到这个寄存器”“将这两个寄存器中的整数相加”，诸如此类。

通过解码和按顺序执行指令来操作 CPU。没有像 AST 那样的树状结构，控制流是通过从代码中的一个点跳到另一个点来实现的。没有中间层，没有开销，没有不必要的跳转或指针寻址。

闪电般的速度，但这种性能是有代价的。首先，编译成本地代码并不容易。如今广泛使用的大多数芯片都有着庞大的拜占庭式架构，其中包含了几十年来积累的大量指令。它们需要复杂的寄存器分配、流水线和指令调度。

当然，你可以把可移植性抛在一边。花费几年时间掌握一些架构，但这仍然只能让你接触到一些流行的指令集。为了让你的语言能在所有的架构上运行，你需要学习所有的指令集，并为每个指令集编写一个单独的后端。

3.1.3 什么是字节码？

记住这两点。一方面，树遍历解释器简单、可移植，而且慢。另一方面，本地代码复杂且特定于平台，但是很快。字节码位于中间。它保留了树遍历型的可移植性—在本书中我们不会编写汇编代码，同时它牺牲了一些简单性来换取性能的提升，虽然没有完全的本地代码那么快。

结构上讲，字节码类似于机器码。它是一个密集的、线性的二进制指令序列。这样可以保持较低的开销，并可以与高速缓存配合得很好。然而，它是一个更简单、更高级的指令集，比任何真正的芯片都要简单。（在很多字节码格式中，每条指令只有一个字节长，因此称为“字节码”）

想象一下，你在用某种源语言编写一个本地编译器，并且你可以全权定义一个尽可能简单的目标架构。字节码就有点像这样，它是一个理想化的幻想指令集，可以让你作为编译器作者的生活更轻松。

当然，幻想架构的问题在于它并不存在。我们提供编写模拟器来解决这个问题，这个模拟器是一个用软件编写的芯片，每次会解释字节码的一条指令。如果你愿意的话，可以叫它**虚拟机**。

模拟层增加了开销，这是字节码比本地代码慢的一个关键原因。但作为回报，它为我们提供了可移植性。用像 C 这样的语言来编写我们的虚拟机，它已经被我们所关心的所有机器所支持，这样我们就可以在任何我们喜欢的硬件上运行我们的模拟器。

这就是我们的新解释器 clox 要走的路。我们将追随 Python、Ruby、Lua、OCaml、Erlang 和其它主要语言实现的脚步。在许多方面，我们的 VM 设计将与之前的解释器结构并行。

当然，我们不会严格按照顺序实现这些阶段。像我们之前的解释器一样，我们会反复地构建实现，每次只构建一种语言特性。在这一章中，我们将了解应用程序的框架，并创建用于存储和表示字节码块的数据结构。

3.2 开始

除了 main() 还能从哪里开始呢？启动你的文本编辑器，开始输入。

```
#include "common.h"

int main(int argc, const char* argv[]) {
  return 0;
}
```

从这颗小小的种子开始，我们将成长为整个 VM。由于 C 提供给我们的东西太少，我们首先需要花费一些时间来培育土壤。其中一部分就在下面的 header 中。

```
#ifndef clox_common_h
#define clox_common_h

#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

#endif
```

在整个解释器中，我们会使用一些类型和常量，这是一个方便放置它们的地方。现在，它是古老的 NULL、size_t，C99 中的布尔类型 bool，以及显式声明大小的整数类型—uint8_t 和它的朋友们。

3.3 指令块

接下来，我们需要一个模块来定义我们的代码表示形式。我一直使用“chunk”指代字节码序列，所以我们把它作为该模块的正式名称。

```
#ifndef clox_chunk_h
#define clox_chunk_h
```

```
#include "common.h"

#endif
```

在我们的字节码格式中，每个指令都有一个字节的**操作码**（通常简称为 **opcode**）。这个数字控制我们要处理的指令类型—加、减、查找变量等。我们在这块定义这些：

```
#include "common.h"
#endif
```

现在，我们从一条指令 OP_RETURN 开始。当我们有一个全功能的 VM 时，这个指令意味着“从当前函数返回”。我承认这还不是完全有用，但是我们必须从某个地方开始下手，而这是一个特别简单的指令，原因我们会在后面讲到。

3.3.1 保存指令的动态数组

字节码是一系列指令。最终，我们会与指令一起存储一些其它数据，所以让我们继续创建一个结构体来保存所有这些数据。

chunk.h，在枚举 OpCode 后添加

```
} OpCode;
typedef struct {
  uint8_t* code;
} Chunk;
#endif
```

目前，这只是一个字节数组的简单包装。由于我们在开始编译块之前不知道数组需要多大，所以它必须是动态的。动态数组是我最喜欢的数据结构之一。动态数组提供了：

- 缓存友好，密集存储
- 索引元素查找为常量时间复杂度
- 数组末尾追加元素为常量时间复杂度

这些特性正是我们在 jlox 中以 ArrayList 类的名义一直使用动态数组的原因。现在我们在 C 语言中，可以推出我们自己的动态数组。如果你对动态数组不熟悉，其实这个想法非常简单。除了数组本身，我们还保留了两个数字：数组中已分配的元素数量（容量，capacity）和实际使用的已分配元数数量（计数，count）。

chunk.h，在结构体 Chunk 中添加代码

```
typedef struct {
  int count;
  int capacity;
  uint8_t* code;
} Chunk;
```