07｜怎么设计属于我们自己的虚拟机和字节码？

2021-08-23 宫文学

《手把手带你写一门编程语言》 课程介绍 >

讲述：宫文学
时长 19:23 大小 17.75M

你好，我是宫文学。

到目前为止，我们的语言看上去已经有点像模像样了。但是有一个地方，我还一直是用比较凑合的方式来实现的，这就是**解释器**，这节课我想带你把它升级一下。

在之前的内容中，我们用的解释器都是基于 AST 执行的，而实际上，你所能见到的大多数解释执行的脚本语言，比如 Python、PHP、JavaScript 等，内部都是采用了一个虚拟机，用字节码解释执行的。像 Java 这样的语言，虽然是编译执行，但编译的结果也是字节码，JVM 虚拟机也能够解释执行它。

为什么这些语言都广泛采用虚拟机呢？这主要是由基于 AST 的解释器的不足导致的。今天这节课，我就带你先来分析一下 AST 解释器的缺陷，然后了解一下像 JVM 这样的虚拟机

的运行原理，特别是栈机和寄存器机的差别，以便确定我们自己的虚拟机的设计思路。

看上去任务有点多，没关系，我们一步一步来，我们先来分析一下基于 AST 的解释器。

基于 AST 的解释器的缺陷

其实，我们目前采用的解释器，是一种最简单的解释器。它有时被称为“树遍历解释器”(Tree-walking Interpreter)，或者更简单一点，也被叫做“AST 解释器”。

为什么我刚刚会说我们这个基于 AST 的解释器有点凑合呢？你可能会想通过遍历 AST 来执行程序不是挺好的吗？

确实，AST 解释器虽然简单，但很有用。比如，最常见的就是对一个表达式做求值，实现类似公式计算的功能，这在电子表格等系统里很常见。甚至在 MySQL 中，也是基于 AST 做表达式的计算，还有一些计算机语言的早期版本（如 Ruby），以及一些产品中自带的脚本功能，也是用 AST 解释器就足够了！

不过，虽然 AST 解释器很有用，但它仍然有明显的缺陷。**最主要的问题，就是性能差，在运行时需要消耗更多的 CPU 时间和内存。**在上一节课里，你可能使用过我们的函数特性计算过斐波那契数列，在参数值比较大的情况下（比如 n 大于 30 以后），你会看到程序的运行速度确实比较慢。

为什么会这样呢？你再来看我们的解释器，会发现它哪怕只是做一个表达式求值，也要层层做很多次的函数调用。比如，简单的计算 2+3*5，需要做的调用包括：

复制代码

```
visitBlock()
  visitStatement()
    visitExpressionStatement()
      visitBinary()   //+号
        visitIntegerLiteral() //2
        visitBinary() //*号
          visitIntegerLiteral() //3
          visitIntegerLiteral() //5
```

从表面上看起来，这只是做了 8 次的函数调用。其实，如果你仔细看我们代码的细节，就会发现，由于我们的程序采用了 Visitor 模式，每一次调用还都有一个

Visitor.visit(AstNode) 和 AstNode.accept(Visitor) 来回握手的过程，所以加起来一共做了 24 次函数调用。

这样的函数调用的开销是很大的。在上一节课，你已经知道了，每次函数调用都需要创建一个栈桢，这会导致内存的消耗。调用函数和从函数返回，也都需要耗费额外的 CPU 时间。在后面的课程里，等我们对程序的运行时机制的细节了解得更清楚以后，你会更加理解这些额外的开销发生在什么地方。

除了性能问题，AST 解释器还有其他的问题。比如，我们已经看到，在实现 Return 语句的时候，需要额外的冗余处理，以便跳过 Return 后面的语句，类似的情况还发生在 Break、Continue 等语句中。总的来说，在控制流的跳转方面，用 AST 都不方便。

还有，我们执行函数调用的时候，需要从函数调用的 AST 节点跳到函数声明的节点，这让人有点眼花缭乱。如果我们后面支持类、Lambda 等更加丰富的特性，需要运行类的构造函数、进行类成员的初始化，查找并执行正确的父类或子类的方法，那么程序的执行过程会更加让人难以理解。

而且，从根本上来说，AST 这种数据结构，比较忠实地体现了高级语言的语法特征。而高级语言呢，是设计用来方便人类理解的，并不是适合机器执行的。把计算机语言从适合人类阅读，变成适合机器执行，本来就是编译器要做的事情。不过，把高级语言变成 AST，我们叫做解析（Parse），还不能称上是编译（Compile）。要称得上编译，要对程序的表示方式做更多的变换才行。

那接下来呢，我们就探讨一下如何把程序编译成对机器更友好的方式。按照循序渐进的学习原则，我们不会一下子就编译成机器码，而是先编译成字节码，并试着实现一个虚拟机。

初步了解虚拟机

说到虚拟机，我们大多数人都不陌生。比如 Java 语言最显著的特征之一，就是运行在虚拟机上，而不是像 C 语言或 Go 语言那样，编译成可执行文件直接运行；.NET 也是用了类似的架构。就算现在 Java 支持 AOT（Ahead of Time）编译方式了，它的可执行文件中仍然打包了一个小型的虚拟机，以便支持某些特别的语言特性。

至于其他语言，如 Python、JavaScript 等，虽然我们不怎么提及它们的虚拟机，但它们通常都是基于虚拟机来运行的。

虚拟机如此流行，不是偶然现象，因为它提供了一些明显的优点。其中最值得注意的，就是**程序可以独立于具体的计算机硬件和操作系统而运行**。所以，我们在任何设备上，无论是 Mac 电脑、Windows 电脑、安卓手机还是 iPad，都可以用浏览器打开一个页面，并运行里面的 JavaScript。而如果采用 C 语言那样的运行方式，那么针对每种不同的 CPU 和操作系统的组合，都要生成不同的可执行文件，这对于像浏览器一样的很多应用场景，显然都是很麻烦的。

虚拟机还能提供其他好处，比如通过托管运行提供更高的安全性，还有通过标准的方式使用不同计算机上的文件资源、网络资源、线程资源和图形资源等等。

实际上，虚拟化是计算机领域的一个基本思路。比如，云计算平台能够虚拟出很多不同的操作系统，在基于 ARM 芯的 Mac 上可以仿真运行基于 X86 的 Windows 等等，都是不同角度的虚拟化。

所以说，要实现一门现代计算机语言，就不能忽视虚拟机方面的知识。

那语言的虚拟机都要包含哪些功能呢？又是由哪些部分构成的呢？

介绍虚拟机的文章很多，特别是针对像 JVM 和安卓的 ART 这样广泛使用的平台，有专门的书籍和课程来深入剖析。在我们的课程里，由于我们自己要实现一下虚拟机，因此我也会简单介绍一下虚拟机的原理和构成，当然更多的就要靠你动手实践来掌握虚拟机的精髓了。

总的来说，虚拟机可以看做是一台虚拟的计算机，它能像物理计算机一样，给程序提供一个完整的运行环境。

首先，虚拟机像物理计算机一样，会支持一个指令集。我们的程序按照这个指令集编译成目标代码后，就可以在虚拟机上执行了。

第二，虚拟机也像物理计算机一样，提供内存资源。比如在 JVM 中，你可以使用栈和堆两种内存。根据不同语言的内存管理机制的不同，在虚拟机里通常还要集成垃圾收集机制。

第三，虚拟机要像物理计算机一样，能够加载程序的代码并执行。程序的目标代码文件中，除了可执行的代码（也就是虚拟机指令），还会包含一些静态的数据，比如程序的符号表，这会让你的程序支持元编程功能，比如运行时的类型判断、基于反射来运行代码等等。静态数据还包括程序中使用的一些常量，比如字符串常量、整数常量等等。对于代码和静态数据，会被虚拟机放在特定的区域，并且能够被指令访问。

此外，虚拟机还要在 IO、并发等方面提供相应的支持。这样，我们才可以实现像在终端打印字符这样的功能。

好了，我们已经大概了解了虚拟机相关的概念。不过，不同的虚拟机在运行代码的机制方面是有所区别的，这也会影响到字节码的设计和算法的实现，所以我们现在展开介绍一下。

你可能听说过寄存器机和栈机，这就是比较流行两种程序运行机制。

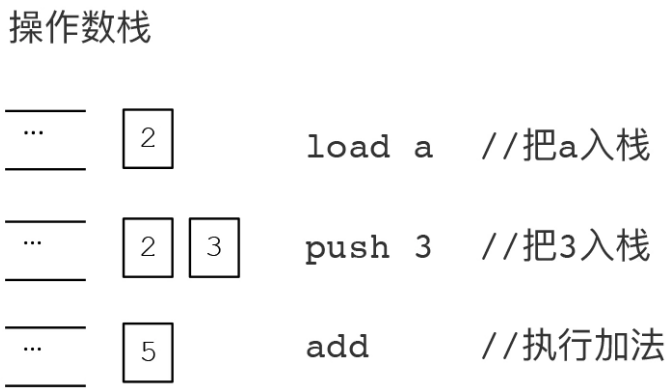
栈机和寄存器机

使用栈机的典型代表，就是 JVM，它能够运行 Java 的字节码。Web Assembly 是为浏览器设计的字节码，它的设计也是栈机的架构。

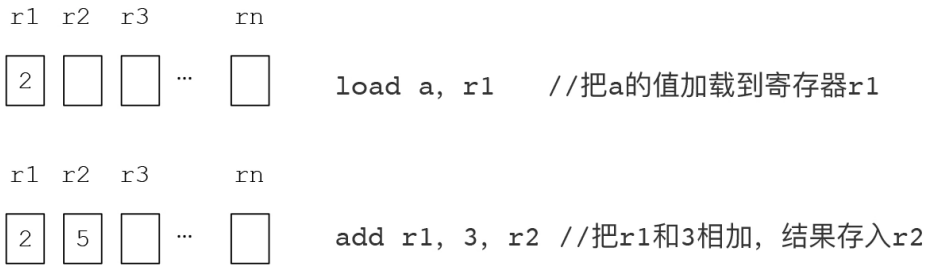
使用寄存器机的典型代表是能够运行 JavaScript 的 V8，V8 里面有一个基于寄存器机的字节码解释器。而 Lua 和 Erlang 内部也是采用了寄存器机作为程序的运行机制，其实我们现在使用物理计算机，也是寄存器机。

栈机和寄存器机的主要区别，是获取操作数的方式不同：栈机是从操作数栈里获取操作数，而寄存器机是从寄存器里获取。比如要计算“a+3”这个表达式，虚拟机通常都提供了一个指令用来做加法。a 和 3 呢，则是加法指令的操作数，其中 a 是一个本地变量，其值为 2，另一个操作数是常量 3。那怎么完成这个加法操作呢？

栈机的运行方式，是先把 a 的值从内存取出来，压到一个叫做操作栈的区域（使用 load 指令），然后把常量 3 压到操作数栈里（使用 push 指令），接着执行 add 指令。在执行 add 指令的时候，就从操作数栈里弹出 a 和 3，做完加法以后，再把结果 5 压到栈里。



而寄存器机的运行方式，是先把 a 的值加载到寄存器，在执行 add 指令的时候，从这寄存器取数，加上常量 3 以后，再把结果 5 放回到寄存器。



总结起来，栈机和寄存器机的区别有这三个方面：

第一，操作数的存储机制不同。栈机的操作数放在操作栈，操作数栈的大小几乎不受限制；而寄存器机的操作数是放在寄存器里，寄存器的数量是有限的。需要说明的是，对于物理机来说，寄存器指的是物理寄存器；而对于虚拟机来说，寄存器通常只是几个本地变

量。但因为这些变量被频繁访问，根据寄存器分配算法，它们有比较大的概率被映射成物理寄存器，从而大大提高运行性能。

第二，指令的格式不同。寄存器机的指令，需要在操作数里指定运算所需的源和目的操作数。而栈机的运算性的指令，比如加减乘除等，是不需要带操作数的，因为操作数就在栈顶；而像 push、load 这样的指令，是把数据压到栈里，也不需要指定目的地，因为这个数据也一定是存到栈顶的。

第三，生成字节码的难度不同。从 AST 生成栈机的代码是比较容易的，你在后面就可以体会到。而生成寄存器机的代码的难度就更高一些，因为寄存器的数量是有限的，我们必须要添加寄存器分配算法。

了解了栈机和寄存器机的这些差别以后，我们就可以根据自己的需求做取舍。比如，如果要更关注运行性能，就选用寄存器机；而如果想实现起来简单一点，并且指令数量更少，便于通过网络传输，我们就可以用栈机。

好了，现在我们已经初步了解了两种运行机制和两类指令集的特点了，是时候设计我们自己的虚拟机和字节码了！

设计我们自己的虚拟机

设计一个虚拟机是一项挺有挑战的工作，不过，如果我们仍然采取先迈出一小步，然后慢慢迭代的思路，就没那么复杂了。

在实现一个虚拟机之前，有些关键的技术决策是要确定一下的，这些决策影响到虚拟机的特性和我们所采用的技术。

决策 1：选择栈机还是寄存器机？

其实，栈机和寄存器都能满足我们对于程序运行的核心需求，因为我们目前对性能、字节码的大小都没有什么特别的要求。不过，经过思考，**我最终选择了栈机**，主要有这几个考虑：

首先，Java 的 JVM 用的就是栈机，而且讲述 JVM 的资料也很多，方便我们借鉴和学习它成熟的设计思路。

第二，由于我们最后是要生成面向物理机的机器码，而物理机就是寄存器机，所以我们肯定会学到这方面的知识。从扩大知识面的角度，我们在虚拟机层面熟悉一下栈机就更好了。

第三，Web Assembly 是一个很有前途的技术领域，目前各门语言都在添加编译成 Web Assembly 的工作。而自己动手实现一个栈机的经验，有助于我们理解 Web Assembly，为未来支持 Web Assembly 打下基础。

决策 2：字节码如何设计？

这个决策也简单。**既然已经选择了栈机的机制，那就参考最成熟的 Java 的字节码就好了。**这样，我们只要能生成类似于 Java 的字节码，就能保证编译程序的正确性。

说到这里，你其实还想到一个可能性：是不是可以把 TypeScript 编译以后，放在 JVM 上运行呢？

这是完全可能的，这样的工作也很有意义，你将来如果有需要可以做这样的一个实现。作为参考，我前一阵在技术会议上认识的一个极客朋友，做了一项很有意思的工作，就是把游戏领域的 DSL 编译成.NET 的字节码，从而驱动游戏引擎。通过这个核心技术，他开发了一个低代码游戏开发平台，大大降低了开发游戏的成本，提高了开发效率。不过由于 JVM 不是我们这个课程的重点，我们这个课程就不去专门生成能跑在 JVM 上的字节码文件了。

决策 3：最先支持哪些数据类型？

字节码的设计，跟语言特性是紧密相关的。比如，在 Python 的字节码中，所有操作都是针对对象的，因为在 Python 中，所有的数据都是对象化的。而在 Java 语言中，数据类型包括基础数据类型和对象两类，所以相应的指令也分为了两类，一类用于操作整型、浮点型等基础数据类型，一类用于操作对象。

TypeScript/JavaScript 语言提供了 number、string、boolean 等内置数据类型，也支持自定义类型。但我们目前还没有探讨到像 class 这样的自定义类型机制。不过，到目前为止，我们的语言特性基本上只支持数值运算。

所以第一步，我们就先只支持 number 类型就好了。甚至为了简单起见，我们就只支持整型运算就好了。一旦通过整型把整个实现机制跑通，再去支持其他数据类型也就没那么复杂了。

决策 4：采用什么语言来实现虚拟机？

到目前为止，我们的课程采用的语言一直是 TypeScript 和 JavaScript，但虚拟机却对运行性能的要求比较高，所以最好能够进行一些底层的控制，比如如何为栈桢分配内存、如何最高效地存储和访问常量等等，所以真实世界的虚拟机，大多是采用 C 和 C++ 实现的。

所以在这节课中，我们采取这样的策略：**一方面，采用 TypeScript 语言做原型**，这样可以帮助我们迅速理清楚设计思路，掌握原理。**另一方面，我们又会用 C 语言做一个参考实现**，这样我们能够充分享受到采用基于字节码的虚拟机所带来的优点。好消息是，大部分同学都或多或少接触过 C 语言，所以理解和使用起来并不会特别困难。

课程小结

今天这节课，我们迈入了一个新的领域，也就是采用虚拟机作为新的运行机制。

相比我们之前运行的 AST 解释器，运行字节码的虚拟机的性能更高，更容易运行流程控制语句。而且，字节码也是更加面向机器的代码表示方式，因此也更便于机器运行，但不利于人类阅读。

而虚拟机中有一个重要部分，就是程序的运行机制。对运行机制的了解是你实现一门语言时所需要的基本功。今天我们初步介绍了栈机和虚拟机两种比较流行的程序运行机制的区别，主要是在操作数的存储机制、指令的格式和生成字节码的难度这三点上，在后面的课程里，你会对这两种运行机制有更加深入的了解。

最后，我们对虚拟机做了一些设计决策。我们选择了栈机，决定参考最成熟的 Java 的字节码，最先实现 number 类型和整型运算，而且我还会带你用 TypeScript 和 C 语言实现两种虚拟机。虽然这节课还没有做具体的实现，但理清设计思路可能比实现还重要，特别是面对虚拟机这样一个比较新的话题时。不要着急，我们下一节就会实现出一个简单的栈机！

思考题

你所使用的语言有没有内置一个虚拟机？有什么特点？里面有没有字节码解释器？它是栈机还是寄存器机？欢迎你和我分享一下。

另外，如果你是带着实际的项目需求来学习这门课程的。那么，对于你的需求，AST 解释器是否够用，还是要必须采用其他的运行机制？为什么？

感谢你和我一起学习，如果你觉得我这节课讲得还不错，也欢迎你把它分享给更多对编程语言感兴趣的朋友。我是宫文学，我们下节课见。

上一篇 06 | 怎么支持条件语句和循环语句？

下一篇 08 | 基于TypeScript的虚拟机（一）：实现一个简单的栈机

精选留言 (4)

孤星可
2021-08-23

小弟用 Java 实现了 Java 虚拟机， 有兴趣可了解。
https://github.com/guxingke/mini-jvm

2

大豆
2021-08-23

V8居然也是寄存器机，也难怪，它跟Android同一个爸爸。

写点啥呢
2021-08-23

请问老师，lua的字节码执行我看到是每次执行压栈并且将结果值保存在栈顶（参考的5.4.

2），我理解这算是栈机的实现，为什么说lua是基于寄存器虚拟机呢？

展开

2

quanee

2021-08-23

老师, 想问一下 Go 的 runtime 和 Plan9 与 Java 的 JVM 和 字节码有什么区别, 总感觉功能上差不多?

2