□ 极客头条 (/) □ (/search) □ □

□ 中间语言和虚拟机漫谈

8

□ 编程语言 (http://www.csdn.net/tag/编程语言/news) 虚拟机 (http://www.csdn.net/tag/虚拟机/news) □

文 / 徐宥

编程语言的发展历史，总的来说，是一个从抽象机器操作逐步进化为抽象人的思维的过程。机器操作和人的思维如一枚硬币的两面，而语言编译器就像是个双面胶，将这两面粘在一起，保证编程语言源程序和机器代码在行为上等价。当然，人本身并不是一个完美的编译器，不能无错的将思维表达为高级语言程序，这种偏差，即Bug。因为编译器的帮助，我们可以脱离机器细节，只关心表达思维和程序行为这一面。

编程语言的发展日新月异。特别是随着对问题的深入理解，新的设计思想，语法构建和新的领域相关语言（DSL）层出不穷。而硬币的另一面似乎一直波澜不惊。这是自然的——无需关心底层架构的变化，或者目标代码生成优化等技术的进化，正是编译器带给我们的好处，因为这些细节和要解决的问题往往关系不大。

尽管所受的关注度不高，这些底层的技术一直在持续地进步。特别是这十年来，一场大的变革正在悄悄发生。这场变革，就是中间语言和虚拟机几乎成为了编程语言的标配——编译器不再以机器的CPU指令集作为编译目标，而是生成针对某种中间语言或虚拟机指令集的目标代码。这场变化是深刻的，它意味着编程语言的设计者自此完全脱离了具体硬件平台的束缚，语言如何设计和如何执行成为了两个完全正交的系统。这个变革大幅度降低了创造一个新语言的成本，一下子把我们推入了一个语言井喷的时代。

从抽象语法树到中间语言

熟悉编译器设计的读者都知道，编译的第一步是构建一个叫抽象语法树(AST)的数据结构 语法树这个概念来源于LISP)。有了这样的数据结构后，解释器和编译器在此分野。以AST为起点，解释器完全可以遍历语法树，递归执行每个子结点。IEEE POSIX (或称标准UNIX) 规定的 AWK 语言，其经典实现就是一个生成和遍历语法树的过程：

```
...

 syminit();
 compile_time = 1;
 Node    *winner ;    /* root of parse tree */
 yyparse();               /* generate parse tree */

 ...

 if (errorflag == 0) {
   compile_time = 0;  /* switch to execution */
   run(winner);   /* execution of parse tree starts here */
 }

...
```

Awk这样的传统解释器的优点在于结构简单，开发便利。事实上许多领域专用语言都采取这种方式实现，如PostScript、Matlab、R等。

解释执行的缺点也是显而易见的。首要的一点就是每次执行都需要重新生成语法树。领域专用语言或许可以忍受每次零点几秒的重复解释过程，而对于可以开发大型应用的通用编程语言来说，这一点是致命的。每次重新生成语法树也意味着这样的语言难以用于资源受限系统，因为语言本身语法结构复杂，布置一个解释模块的代价往往非常高昂。为了避免解释执行的这些弊端，传统的编译器致力于只解释一次，将通用语言的语法树，直接转变为目标机器的CPU指令。传统的FORTRAN和C编译器就是如此设计的。有些编程构建，如C语言中的C++，甚至是直接受CPU指令影响的产物。

]认识到，其实传统
[为一种中间指令格
从生成目标指令的编
语法树生成，而直接

较早在中间语言上进行探索的是MIT的LISP机器。如Thomas Knight，他的研究集中在如何在硬件上实现一个高效的LISP环境。显然，没有一个硅片可以直接运行mapcar，但设计一个支持mapcar的中间语言并不困难，只需要支持一些基本的列表操作即可。这种设计思想影响了很多后来的系统。流行的GCC编译器，从结构上来说分前端和代码生成端两部分。连接两者的中间语言RTL的基本一些指令，都可以追溯到LIPS机器的指令集。

中间语言和虚拟机

中间语言可用于程序优化的原因是显而易见的：这种中间格式既贴近机器代码，又保存了原有程序的结构。程序优化并不是一门魔术。像循环展开，死代码消除等技术，都依赖于程序控制结构，而中间语言可以保持这样的控制结构。事实上，目前我们所知的编译优化技术，无一不是建立在结构分析之上。中间语言的出现让程序优化成为了一个独立的问题。原本单列的C程序优化, FORTRAN程序优化如今统一归结为RTL程序优化。编译器前端可以千差万别支持许多语言，但负责优化和翻译为目标代码的后端均归为一个，就此一点，就大大简化了语言编译器的设计门槛。现如今，几乎没有一个语言设计者需要考虑如何生成高效目标代码了。

当然，中间语言的作用并不仅限于目标代码优化。如果我们把中间语言也当作一种语言的话，不难发现中间语言甚至比原语言更加普及。 比如，Java虚拟机(JVM) 语言实际上是一个比Java语言成功许多倍的产品。JVM存在于众多Java语言不存在的地方。像Jython、Scala和JRuby这样的语言，均依赖于JVM，而非Java语言本身。

语言的虚拟机的本质，是一个可以运行中间语言的机器。在实际硬件上，程序和数据是两个截然不同的概念；而对于虚拟机来讲，中间语言程序，只是虚拟机程序的输入数据罢了。这种将程序当作数据的处理方式，带来了我们熟知的许多虚拟机的优点，如跨平台特性，安全性等等。因为程序即是数据，为虚拟机读取中间语言程序方便，其指令往往都是以字节为单位，故称为字节码 (bytecode)。 相比之下，计算机的CPU指令则可长度不一，也不一定占据整数个字节。

程序是数据这个特性使得虚拟机可以做到跨平台和沙箱安全；反过来，数据是程序又使得虚拟机可以用在一些意想不到的地方，使数据更加灵活。目前通行的轮廓字体描述语言TrueType就是成功运用虚拟机来更加灵活地处理字体的一个例子。

TrueType是一种采用数学函数描述字体的矢量字体。矢量字体在理论上可以自由缩放。而实践中，因为显示器本质上是点阵的，所有的矢量字形都要经过栅格化（rasterization），将矢量轮廓近似转化为像素点的透明度。然而，这种近似并不是随意的。以汉字 “中” 为例，为保证其对称美观，我们必须约束栅格化程序，保证任何时候左右两个竖线与中间一竖的距离相等，哪怕为此不惜将此字缩减或放宽一两个像素。这类约束又被称作提示(hinting)。它对字体至关重要——缺少提示的矢量字体在字形较小时不可避免地会出现失真，变形和锯齿等现象。不难理解，本质上“提示”是一个以字体轮廓和字形大小为输入，以栅格数据为输出的程序。因为此，TrueType包含了一套虚拟机指令，方便字体设计者表达这种提示。可以想象，如果没有这个虚拟机的存在，设计灵活的矢量字体是不可能完成的任务。实际上，我们所见到的几乎所有的矢量字体文件，都是一个数据和程序的混合物。 从另一方面来说，每个字形都需要一个专门的“提示”，也从一个侧面说明了设计高质量的中文字体之难度。

基于栈，还是基于寄存器

凡提到虚拟机，绕不过去的第一个问题就是这个虚拟机是基于栈的，还是基于寄存器的（有些虚拟机，如LISP机器，可以同时有栈和寄存器）？尽管这里“寄存器”和“栈”，都不一定直接对应到机器CPU的寄存器或者内存里的栈。这个问题之所以重要，因为它直接决定了虚拟机的应用场景。一般说来，基于栈的虚拟机结构相对简单，且更加适合资源受限系统。 比如上文我们说的TrueType虚拟机，结构简单，功能专一，就是基于栈的。

尽管所有的计算机的存储模型都是构建在图灵机的无穷纸带模型上，实践中所有语言都或多或少依赖于栈模型。特别的，函数调用就等价于栈的推入和弹入操作，其他操作均可抽象为对栈顶元素进行。相比之下，寄存器模型虽然贴近真实机器，却并不够直接：很少有高级语言直接制定寄存器如何分配的，因此编译器的作者需要考量寄存器分配问题。而基于栈的虚拟机的所有指令都可默认为对栈顶元素操作，结构简单，且暂时绕开了寄存器分配难题。

基于栈的虚拟机更加适合内存和CPU处理速度等方面有限的系统。同样的源程序，在目标代码的体积上，面向栈虚拟机上生成的代码更加小。这是容易理解的：基于栈的虚拟机的指令默认对栈顶元素操作，因此指令只需为OP格式，无需OP Reg1, Reg2, Reg3等额外指定寄存器。这个设计也绕开了指令解码问题。平均上说，基于寄存器的虚拟机生成的指令的体积比基于栈的要大。我们见到的许多基于栈的虚拟机，都是为资源受限系统设计的。JVM的初衷是一个运行在电视机顶盒中的小系统，后来精简版本的JVM甚至可以放到智能卡上；Forth语言的虚拟机是要用在计算机固件(Open Firmware)，航空系统和嵌入式系统中；控制打印的PostScript用于高品质打印机中。很显然，机顶盒，引导固件和打印机都是资源受限的系统，这些系统中的虚拟机，不约而同都是基于栈的。值得一提的是，因为实现简单，许多并非用于受限系统的通用语言的虚拟机也是基于栈的，如Python、Ruby、.NET 的CLR等。

基于寄存器的虚拟机，是为性能所生。引入寄存器假设固然关上了用于资源受限系统的门，却也打开了一扇通向进一步性能优化的窗。栈虚拟机的一大缺点就是要不停地将操作数在堆和栈之间来回拷贝。比方说一个简单的三个参数的函数调用，在传递参数上就需要至少三次入栈和出栈操作，而在寄存器上只要指定三个寄存器即可。现代处理器提供的通用寄存器支持，本身就是为了减少这类值的来回拷贝。尽管有Hotspot这样的技术能够将一段栈虚拟机指令转化为基于寄存器的机器指令，可毕竟没有直接从支持寄存器的中间语言翻译直接。前面说过，保持程序的结构是优化的先决条件。失去了“指定三个值”这样的结构的栈虚拟机，需要运行时间接的推断这个操作。而直接指定这些访问结构，将值直接映射到 CPU 的寄存器，正是这类虚拟机运行效率高的要点所在。Android的Dalvik, Perl的Parrot都是基于寄存器的虚拟机，而LLVM则是基于寄存器假设的中间语言。其中，为了让Android程序更加快的运行，

Google不惜放弃JVM指令集，而选择将JVM指令转化为基于有限个寄存器的Dalvik指令集。Parrot和LLVM则更加自由一些，假设了无穷多个寄存器。无论是有限还是无限个寄存器，省却不必要的值拷贝是这类中间语言的最大优点。

JIT和直接执行

JIT（Just-in-time）是运行时的动态编译技术。不难看出，JIT是针对中间语言的——将原语言的编译推迟到运行时并无意义，将中间语言的解释，部分转化为编译后的机器代码，则可以优化运行效率。JIT之所以可行，一个基本假设是程序大多存在热点。D. E. Knuth三十年前观察到的一个现象: 一段FORTRAN程序中不到4%的部分往往占用超过50%的运行时间。因此，在运行时识别这样的热点并优化，可以事半功倍地提高执行效率。

按照Jython作者Jim Hugunin的观测，JIT技术出现后，同样功能的程序，运行于Java虚拟机上的字节码和直接编译成二进制代码的C程序几乎一样快，有的甚至比C快。乍一看虚拟机比原生代码快，理论上是不可能的。而实践中，因为JIT编译器可以识别运行时热点做出特别优化。相比之下，静态编译器的代码优化并不能完全推断出运行时热点。而且，有些优化技术，如将虚函数调用静态化，只有在运行时才能做到。在对热点深度优化的情况下，JIT比直接生成的机器代码执行效率高并不是一件神奇的事情。引入了JIT的，以Python书写的Python执行器pypy, 运行速度要比以C实现的CPython解释器快一到五倍，就是 JIT 技术魅力的一个明证。

尽管JIT技术看上去很炫，实践中也能够做到几乎和原生二进制代码速度相近，我们必须承认，这只是一种补救相对慢的中间语言解释的一种措施罢了。设计语言平台时，设计者可能因为这样那样的原因而选择中间语言／虚拟机解决方案，或因为针对嵌入式系统(Java)，或因为跨平台要求(Android Dalvik)，或者仅仅因为设计者想偷懒不愿写一个从语言到CPU指令的编译器(Python/Ruby)。无论原因为何，当最初的原因已经不存在或不重要，而性能又成为重要考量的话，采用中间语言就显得舍近求远。JavaScript 引擎的进化就是一个生动的例子。

JavaScript语言最初只是一种协助HTML完成动态客户端内容的小语言。Netscape浏览器中的JS引擎，最初只是一个简单的解释器。自2004年Google发布Gmail之后, Ajax 技术的发展对JS引擎的速度提出了更高的挑战。JavaScript引擎的速度被当成一个浏览器是否领先于对手的关键指标。在此情况下，众多浏览器厂商纷纷卷入了一轮JS引擎速度的军备竞赛。

最先挑起这场战争的是Firefox，目标是当时占据90%市场的IE。Firefox 3于2008年6月登场，其JS引擎TraceMonkey在栈虚拟机的基础上首次采用了JIT技术，在当时众多标准评测中超越了IE7。就在当月，WebKit开发小组宣布了基于寄存器的Squirrelfish 引擎，殊途同归，也是基于中间语言，尽管两者互相不兼容。

到9月，Google发布了第一个版本的Chrome浏览器以及新的JS引擎——V8。V8一反使用中间语言的设计套路，力求将JS直接编译到本地代码。Google毫不掩饰V8在标准评测上比其他浏览器快的结果，因此造成了Firefox和Safari开发者对各自JS引擎速度评测的一场恶战。到了9月的时候，Firefox和Safari各自的引擎都比6月份的结果快到20%到60%不等。而V8也赢得了许多眼球，催生了之后的Node.js项目。

这场军备竞赛的一个结果，就是V8以外的引擎，也开始探索绕过中间语言从JavaScript直接生成二进制的可能性。SquirrelFish Extreme就是自Squirrelfish衍生出来的一步本地代码的引擎。值得注意的是，尽管都是生成本地代码，V8和SquirrelFish Extreme这样的编译器，并不是退回到传统的编译器技术上，因为他们已经吸收了许多对JIT编译器性能的研究成果。

Google也将Android执行环境，从原来的Dalvik虚拟机，换成可以直接生成机器代码的ART架构。ART负责在App安装后一次将跨平台的字节码分发格式，编译成原生机器代码。20多年前，为了跨平台，Java采取了虚拟机的设计方案。如今，中间语言的跨平台的部分依然保留，但作为已经不直接参与执行了。硬件的进步带来的中间语言和虚拟机设计的进化，是当时的设计者如何也想不到的事情了。

(http://geek.csdn.net/user/publishlist/u012396362)
lowtech (http://geek.csdn.net/user/publishlist/u012396362)
发布于 GEEKNEWS (http://geek.csdn.net/forum/1) 3小时前

评论

已有1条评论 最新

rrdawlx (http://geek.csdn.net/user/publishlist/wlx1991) 1小时前
(http://geek.csdn.net/user/publishlist/wlx1991)
沙发，好文章!
0 回复 投诉

加载到底了