11 | 基于C语言的虚拟机（二）：性能增长10倍的秘密

2021-09-01 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 15:20 大小 14.06M

你好，我是宫文学。

上一节课，我们初步实现了一个 C 语言版本的虚拟机，让它顺利地跑起来了。你想想看，用 TypeScript 生成字节码文件，然后在一个 C 语言实现的虚拟机上去运行，这个设计，其实和 Java 应用、Andorid 应用、Erlang 应用、Lua 应用等的运行机制是一样的。也就是说，如果退回到智能手机刚诞生的年代，你完全可以像 Android 的发明人一样，用这种方式提供一个移动应用开发工具。

其实，我国最新的自主操作系统 HarmonyOS，也是采用了像我们这门课一样的虚拟机计机制，而且用的就是 TypeScript 语言，这也是我这门课采用 TypeScript 作为教学语言的原因之一。虽然我还没有看到 HarmonyOS 的虚拟机代码，但并不妨碍我去理解它的实

现原理。当然了，你在学完这门课以后，也会更容易理解 HarmonyOS 的开发方式，而且也有助于你阅读它的虚拟机的代码。

好了，对于我们当前成果的吹捧到此打住。让我们回到现实，现实有点残酷：**我们当前实现的基于 C 语言的虚拟机，在上一节课的性能测试中，竟然排名倒数第一。**这显然不正常，这也说明了在虚拟机的设计中，我们还有一些重要的设计考虑被忽视了。

那这一节课呢，我们就来分析一下导致我们虚拟机性能不高的原因，并且针对性地解决掉这个问题。在这个过程中，你会加深对计算机语言的运行时技术的理解，特别是对内存管理的理解。

那首先，让我们把产生性能问题的可能原因分析一下。

性能问题的分析

首先，我们应该了解到一点，现代的 JavaScript 引擎，性能确实挺高的。

在我们 TypeScript 版本的虚拟机中，TypeScript 被编译成了 JavaScript，并在 Node.js 中运行，而 Node.js 又是基于 V8 引擎的。

在互联网的早期，JavaScript 的运行效率比较低。但是后来，以 V8 为代表的 JavaScript 引擎，性能有了大幅度的提升，使得现在的 Web 前端可以实现很复杂的功能。

V8 在运行 JavaScript 的时候，会做即时编译（JIT）。V8 的即时编译器，能够根据运行时收集的信息对类型做推测，这也就避免了由于运行时的类型判断而产生的额外开销，从而生成了跟提前编译（AOT）差不多的代码。如果你想了解更多细节，你可以去看看我在《编译原理实战课》中对 V8 的剖析。

从原理上来说，运行时的推测机制，甚至会生成比提前编译（AOT）更高效的代码。因为它拥有运行时的统计信息，并通过某些优化算法（参考 JVM 的局部逃逸分析算法）实现了更好的编译优化。

换一句话说，V8 也是编译生成了机器码，甚至有时候会生成更高效的机器码。仅从这一点看，它并不会比 C 语言的提前编译差。

不过，JavaScript 毕竟是动态类型的语言，它的编译和运行过程会有一些额外的开销。

比如，编译后的目标代码总要留出一些口子，用来处理类型预测失效的情况。这个时候，它会从运行本地代码的状态退回到解释器去执行。

所以，平均来说，JavaScript 编写的程序，性能不会比 C/C++ 更高。它在某些场景下能接近 C/C++ 的性能，已经相当惊人了。

可是，在上一节中 TypeScript 版本虚拟机的性能居然是 C 语言版本的 2 倍半，这就太不正常了。一定还有别的因素在起作用。

所以，我们来看看第二方面的因素，就是运行时的设计。

在前面的讨论中，我们比较关注的是编译技术与性能的关系。不过，在一个虚拟机中，还会有其他影响性能的因素，这就是**语言的运行时**。运行时就是支撑我们的应用程序运行所需要的一些软件功能，**最常见的运行时功能就是内存管理机制和并发机制**。

在这里，我们重点要看一下内存管理机制。通常我们提到内存管理的时候，一下子就想到垃圾收集机制去了。其实，这只是内存管理的一半工作，完整的内存管理功能还要包括内存的申请机制。

在像 Java、JavaScript 这样的语言中，语言的运行时需要根据程序的指令，随时在内存中创建对象，然后在程序用不到这些对象的时候，再使用垃圾收集机制，把这些对象所占据的内存释放掉。

那么重点就来了：**申请和释放内存，有时会导致巨大的性能开销**。一个好的运行时，必须想办法降低这些开销。

我们初版的 C 语言虚拟机可能就存在这方面的问题。不过，计算机语言的运行时，都是从堆里申请和管理内存的。为了让你理解内存管理和性能的关系，更好地排查出影响 C 语言虚拟机性能的原因，我们首先回顾一下栈和堆这两种基础的内存管理机制。

两种内存管理机制：栈和堆

在现代的操作系统中，为了支持应用的运行，通常会提供栈和堆这两种内存管理机制。当我们在一个 C 语言的函数里使用本地变量时，这些本地变量所需的内存是在栈里申请的，这也就是这个函数所使用的栈桢。而当我们用 C 语言的 malloc 函数申请一块内存的时候，这块内存就是从堆里申请的。

不过，只有像 C/C++ 这样直接编译成本地代码的语言，才可以使用操作系统的栈来保存栈桢。在后面的课程中，我们也会生成与栈桢管理有关的汇编代码，管理栈桢通常需要修改特定寄存器的值，以及使用 push、pop 等辅助的指令。

在我们的解释器所使用的栈桢是自己管理的，本质都是从堆里申请的。**从栈里和堆里申请内存的开销是不一样的。**

从栈里申请内存很简单，基本上只需要修改栈顶指针，也就是某个特定寄存器的值就行了，栈就会自动地伸缩，整个栈的地址空间始终是连续的一整块内存。

而堆就不是了。从堆里申请的内存，由于每个对象的生存期是不一样的，所以就会形成很多的“空洞”，导致内存碎片化。这样，再次申请内存的时候，操作系统需要找到一块大小合适的自由内存空间。这个过程，就需要消耗一定的计算量。在内存碎片化越来越严重的情况下，找到一块可用内存空间的开销会越来越大。

另外，程序的并发也会为堆的内存申请带来额外的开销。在现代操作系统中，每个线程都有自己独享的栈，相互之间不会干扰，但堆却是各个线程所共享的。所以，在分配内存的时候，操作系统会进行线程间的同步，每次只能为一个线程分配内存，避免同一块内存被分配给多个线程。这显然也会降低系统的性能。

现在你再回头来看看我们的 C 语言虚拟机的实现。在栈桢和操作数栈这两个数据结构中，有好几个地方都是指针，比如本地变量的数组、操作数栈，以及操作数栈中的数据区。按照常规的编程方法，我们为每个指针都单独申请了内存。

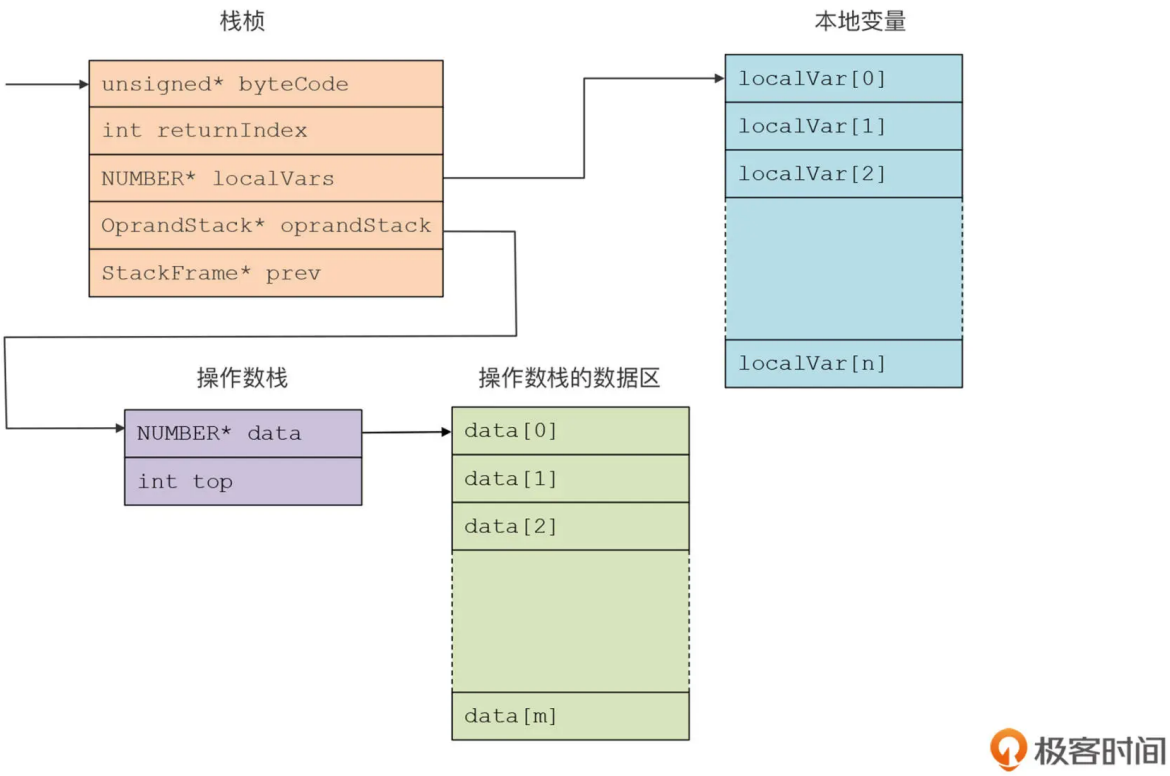
复制代码

```
typedef struct _StackFrame{
    //本栈桢对应的函数，用来找到代码
    FunctionSymbol* functionSym;
    //返回地址
    int returnIndex;
    //本地变量数组
    NUMBER* localVars;
```

```
    //操作数栈
    OprandStack* oprandStack;
    //指向前一个栈桢的链接
    struct _StackFrame * prev;
}StackFrame;

/**
 * 操作数栈
 * 当栈为空的时候，top = -1;
 * */
typedef struct _OprandStack{
    NUMBER * data; //数组
    int top;       //栈顶的索引值
}OprandStack;
```

这样就导致我们一个栈桢的内存布局被切成了 4 小块：

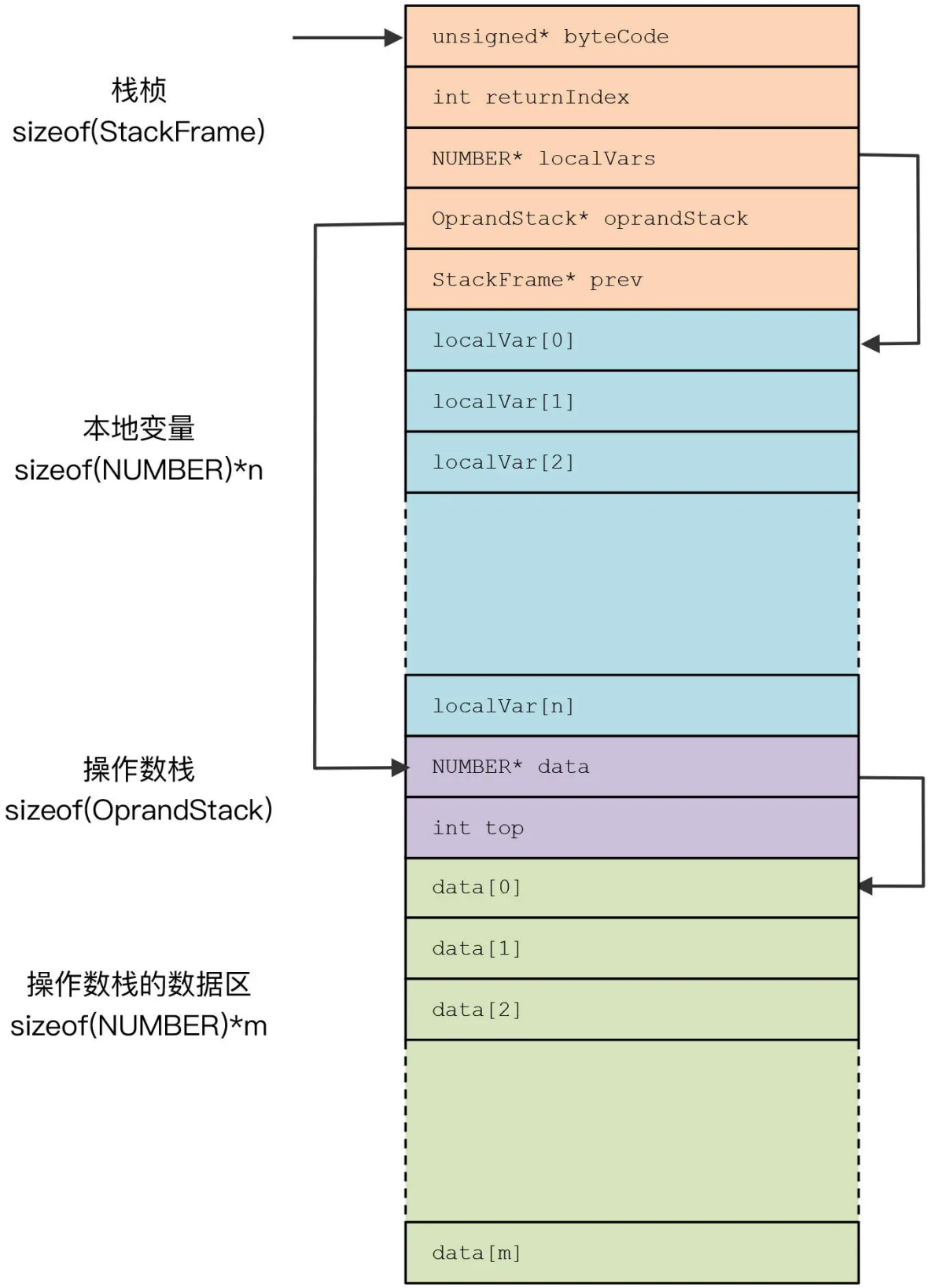


很显然，这是一个可以优化的设计。其实，我们可以一次就申请出整个栈桢所需的内存，而不是分成 4 次。这样减少了 malloc 的调用次数，并且还会更好地避免内存的碎片化。好，现在我们就来做第一次的优化看看。

第一次优化：把栈桢组合成一个整体

这个时候，我们重新编写一下为栈桢申请内存的程序。

首先我们需要计算整个栈桢的整体大小。在下图中，我列出了每块内存大小的计算公式。加到一起后就是整个栈桢的大小了。


栈桢
sizeof(StackFrame)
unsigned* byteCode
int returnIndex
NUMBER* localVars
OprandStack* oprandStack
StackFrame* prev
本地变量
sizeof(NUMBER)*n
localVar[0]
localVar[1]
localVar[2]
localVar[n]
操作数栈
sizeof(OprandStack)
NUMBER* data
int top
操作数栈的数据区
sizeof(NUMBER)*m
data[0]
data[1]
data[2]
data[m]

在申请了一整块内存以后，我们就获得了一个指针，指向内存块的起始位置。然后，我们可以安排内存的布局，并计算出本地变量的数组、操作数栈，以及操作数栈中的数据区这三个指针的值。

这样的话，你就可以在程序中使用这些指针来访问数据了，这跟单独为每个指针申请内存没啥区别。这就是 C/C++ 语言的一些便利之处：**你可以自己给每个指针赋值**。当然，这种操作要小心一点，如果地址计算错误，会引发程序执行的问题。

好了，这样做之后，我们成功地把内存申请和释放的次数降低到了原来的 1/4。那么，还有没有可以进一步优化的地方呢？还是有的，我们来开始第二次优化。

第二次优化：基于 Arena 机制来管理栈桢

在我们的虚拟机中，每一次函数调用，都要创建一次栈桢。这样的话，每一次函数调用都要做一次内存申请。

fibonacci(n) 函数的时间复杂度是指数级。随着 n 的增加，函数调用的次数会迅速增加。具体来说，fibonacci(n) 函数的时间复杂度是下面这个公式：

$$x = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

比如，当 n=30 时，需要做 832,040 次函数调用，当 n=40 时，那就需要做 1 亿多次函数调用。如果按照我们现有的内存管理机制，就要做 1 亿多次内存申请和释放的动作。

那有没有办法减少这个数值呢？当然有。

我们面临的问题，其实像 V8 和 JVM 这些虚拟机都会遇到。并且，它们不仅仅需要降低由于栈桢导致的内存分配和释放的开销，这些虚拟机在运行时还会产生很多小对象，也可能像我们这个例子那样，产生上亿次分配内存的请求，从而导致大量的性能开销。

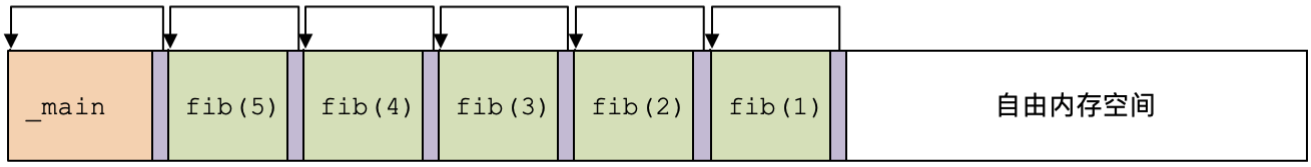
解决这类问题的一个思路，是采用 Arena 内存管理技术。Arena 机制的大致意思是，从操作系统申请一些比较大块的内存。当程序需要比较小块的内存时，从这些大块中自行分配

就行，就不去调用操作系统的接口了。这样，从操作系统的角度看，内存分配的请求会大大减少。

说白了，就是把内存申请从零售改为批发。在所获得的一整块内存里，可以根据软件自己的特点，去合理的管理内存，并提高分配和回收的性能。

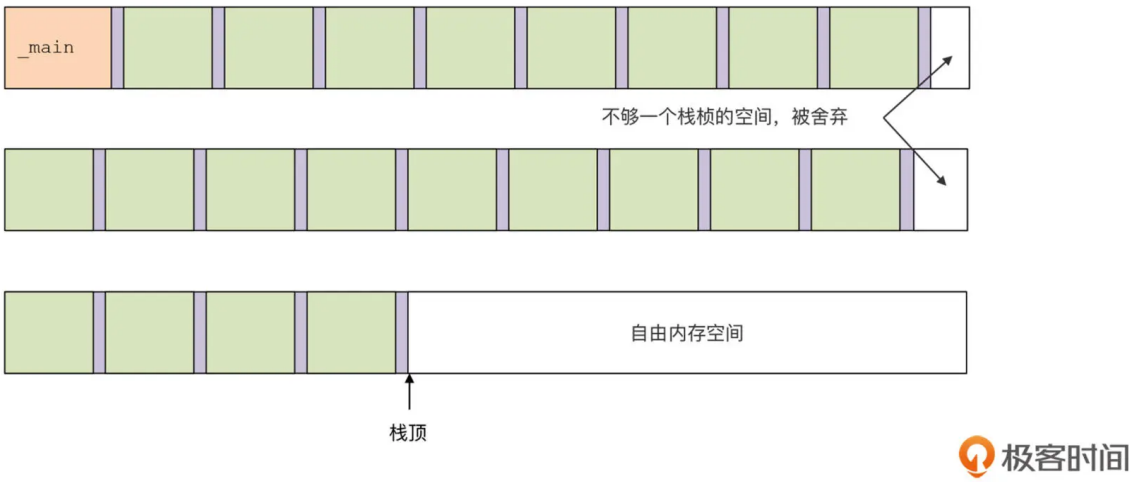
并且，我们如果用 Arena 的方式来管理栈桢的申请，那可以非常简单。因为栈肯定是连续伸缩的，要申请一个新的栈桢，我们只需要在内存块上移动栈顶指针就行。

具体的管理方案你可以看看下面这张图：



当计算 fib(5) 的时候，虚拟机会逐级产生多个栈桢。每次划分出一个栈桢来以后，我们要多占一个位置，用于保存前一个栈桢的栈顶位置。这样，在弹出栈桢的时候，我们直接把栈顶指针指到前一个栈桢的栈顶就可以了。

如果一个内存块不够怎么办？那就用多个内存块，一个块满了就去用下一个。



弹出栈桢的时候，则是一个逆向的过程，栈顶指针也会从一个块退回到前一个块。

好了，用 Arena 来管理栈桢的原理就是这样。具体的实现细节，你可以去看看这节课的示例代码中与 Arena 有关的算法。另外，我还在 playvm.h 里设置了一个叫做 USE_ARENA 的预编译开关。你如果注释掉“#define USE_ARENA”语句，就会回到我们前一个用幼稚的方法申请内存的版本，这样你就能比较它们的性能差异了。

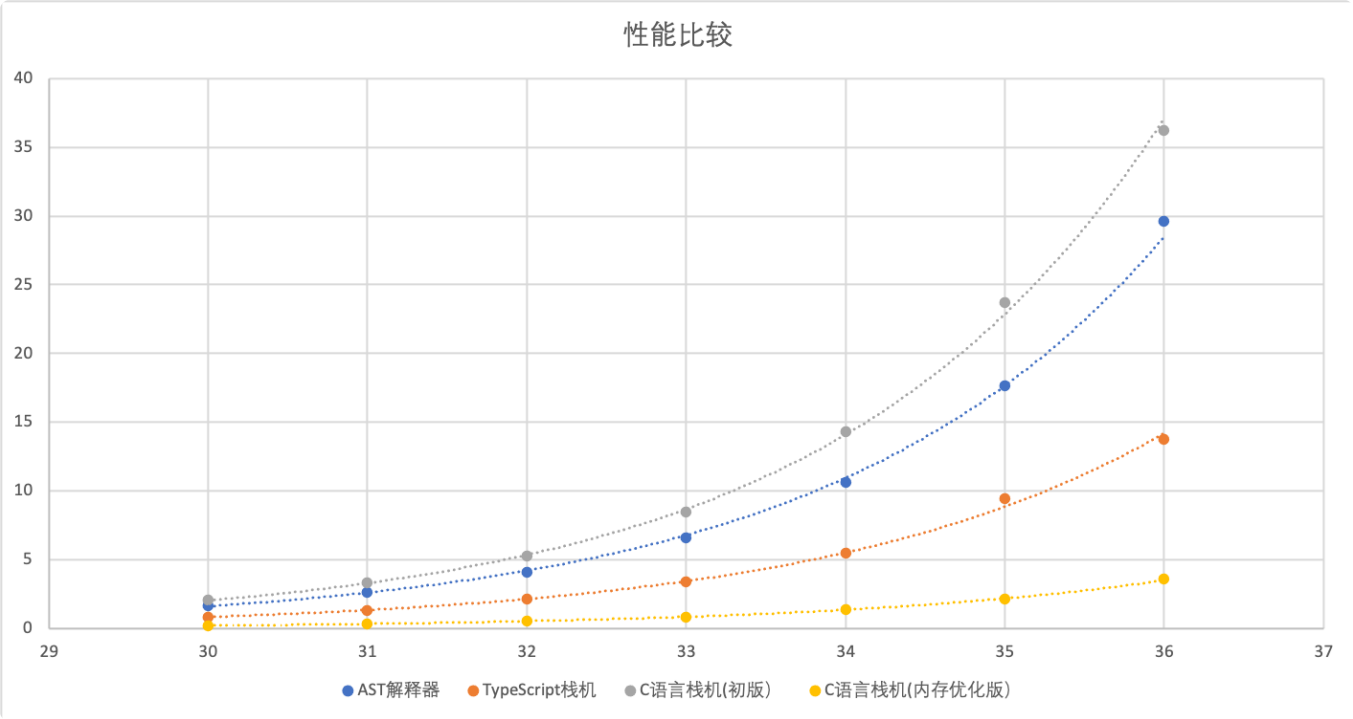
现在又到了检验成果的时候了。我们新的内存管理机制到底能否带来性能上的改进呢？会带来多少改进呢？我们还是赶紧上手测试一下吧！

再次做性能测试

我把测试结果放在了下面的表格中。你能发现，性能的提升是惊人的，**基本上比之前版本的性能提升了 10 倍**！

n	AST解释器（秒）	TypeScript栈机（秒）	C语言栈机(初版)（秒）	C语言栈机（内存优化版）（秒）
30	1.660	0.846	2.052	0.215
31	2.649	1.324	3.290	0.318
32	4.104	2.126	5.283	0.516
33	6.604	3.374	8.487	0.833
34	10.654	5.477	14.330	1.349
35	17.664	9.427	23.721	2.168
36	29.613	13.752	36.212	3.599

同样的，我也做了个图表，让性能比较数据更直观：代表新版虚拟机的线在最下方，性能最快；而原来的版本在最上方，性能最慢。



你看，这个测试结果证明，我们在内存管理方面做优化的思路是正确的。而基于 C 语言版的栈机也终于在性能上展露出优势，是 TypeScript 版栈机的 4 倍左右。

课程小结

今天这节课，我们通过分析和提升基于 C 语言的解释器的性能，初步接触了内存管理方面的知识。栈是一种低成本的获取内存的方式，而从堆中申请内存的开销就要大一些。我们可以通过合并小的内存、采用 Arena 技术等方式，大幅度降低从操作系统申请内存的次数，从而提升系统的整体性能。

除了内存管理机制上的优化外，其实还有一些技术来提升虚拟机的性能。我之前就提过，核心的字节码解释器，最好用一个大函数来实现，尽量减少函数调用的次数。像对操作数栈的压栈和出栈操作，我们目前是用两个小函数实现的，这两个小函数最好内联到主程序 execute 函数中，从而降低函数调用所导致的开销。不过，C 语言的编译器缺省就会对这样的小函数做内联处理。

另外，我们要让程序尽量访问物理寄存器而不是访问内存，这样也能大大提高性能。目前，我们的操作数都是放在内存中的，每一次计算都会对内存做操作。通过一些技术，是有可能把对操作数栈的操作转换成对物理的寄存器的操作的。你可以考虑一下怎样才能实现这个设想。我们后面会讲到寄存器的使用，你可以记着这个话题，到时候再前后印证一下。

在这个过程中，你应该也意识到了，要实现好一门语言，运行时是很重要的。而要实现好运行时，我们需要对操作系统、硬件架构等都有比较深入的了解才行，我们在下一节课会重点讨论这个话题。

思考题

这一节我们为栈桢设计了一个新的内存布局，请你仔细看一下，这个内存布局还有没有进一步优化的可能性，比如栈桢所占的空间是不是可以再缩小一点？

另外，这一节我们做优化的思路，能不能反过来再用在 TypeScript 版本的栈机上，提升一下那个版本的性能呢？你可以分析一下，然后试试看，这样能帮助你加深对 JavaScript 虚拟机的理解。

感谢你和我一起学习，也欢迎你将这节课分享给更多对虚拟机和内存管理机制感兴趣的朋友。我是宫文学，我们下节课再见。

资源链接

这节课的示例代码在这里！

上一篇 10 | 基于C语言的虚拟机（一）：实现一个简单的栈机

下一篇 12 | 物理机上程序运行的硬件环境是怎么样的？

精选留言 (4)

写留言

大豆

2021-09-01

原来如此，Java，dart中一开始为堆，栈分配固定的内存大小。就是采用的Arena内存管理技术，从而减少了系统调用。
从计算机硬件角度来看，Arena 内存管理技术能够增加l1，l2，l3的缓存命中率，减少从物理内存的读取次数，从而来提升效率。
老师，我的想法对吗？

展开

1

罗乾林
2021-09-03

发现代码中opStackSize的大小为固定值，是否可以通过数据流分析的方法，计算出实际需要的最大操作数栈大小，我想这样也能减小栈帧的大小

展开

奋斗的蜗牛
2021-09-02

想不到c语言版的栈帧还可以怎么减少空间

展开

qinsi
2021-09-01

目前这个vm的ts版本无论是字节码还是堆栈都是用Array存储的。入栈和出栈操作都是在Array尾部进行的，而Array是会动态调整大小的。一个可能的优化是事先固定Array大小并自己维护栈顶指针；也可以仿照Arena来尝试优化新创建Array的开销；还可以考虑用Buffer/ArrayBuffer替代Array。不过v8是个高度优化的引擎，上述这些优化究竟有没有作用还要看profile的结果。

展开