09｜基于TypeScript的虚拟机（二）：丰富特性，支持跳转语句

2021-08-27 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 12:41　大小 11.62M

你好，我是宫文学。

在上一节课里，我们已经实现了一个简单的虚拟机。不过，这个虚拟机也太简单了，实在是不够实用啊。

那么，今天这节课，我们就来增强一下当前的虚拟机，让它的特性更丰富一些，也为我们后续的工作做好铺垫，比如用 C 语言实现一个更强的虚拟机。

我们在这一节课有两项任务要完成：

首先，要支持 if 语句和 for 循环语句。这样，我们就能熟悉与程序分支有关的指令，并且还能让虚拟机支持复杂一点的程序，比如我们之前写过的生成斐波那契数列的程序。

第二，做一下性能比拼。既然我们已经完成了字节码虚拟机的开发，那就跟 AST 解释器做一些性能测试，看看性能到底差多少。

话不多说，开干！首先，我们来实现一下 if 语句和 for 循环语句。而实现这两个语句的核心，就是要支持跳转指令。

了解跳转指令

if 语句和 for 循环语句，有一个特点，就是让程序根据一定的条件执行不同的代码。这样一个语法，比较适合我们人类阅读，但是对于机器执行并不方便。机器执行的代码，都是一条条指令排成的直线型的代码，但是可以根据需要跳转到不同的指令去执行。

针对这样的差异，编译器就需要把 if 和 for 这样结构化编程的代码，转变成通过跳转指令跳转的代码，其中的关键是**计算出正确的跳转地址**。

我们举个例子来说明一下，下面是我用 Java 写的一个示例程序，它有一个 if 语句。

复制代码

```
public static int foo3(int a){
    int b;
    if (a > 10){
        b =  a + 8;
    }
    else{
        b = a - 8;
    }
    return b;
}
```

我们先用 javac 命令编译成.class 文件，然后再用 javap 命令以文本方式显示生成的字节码：

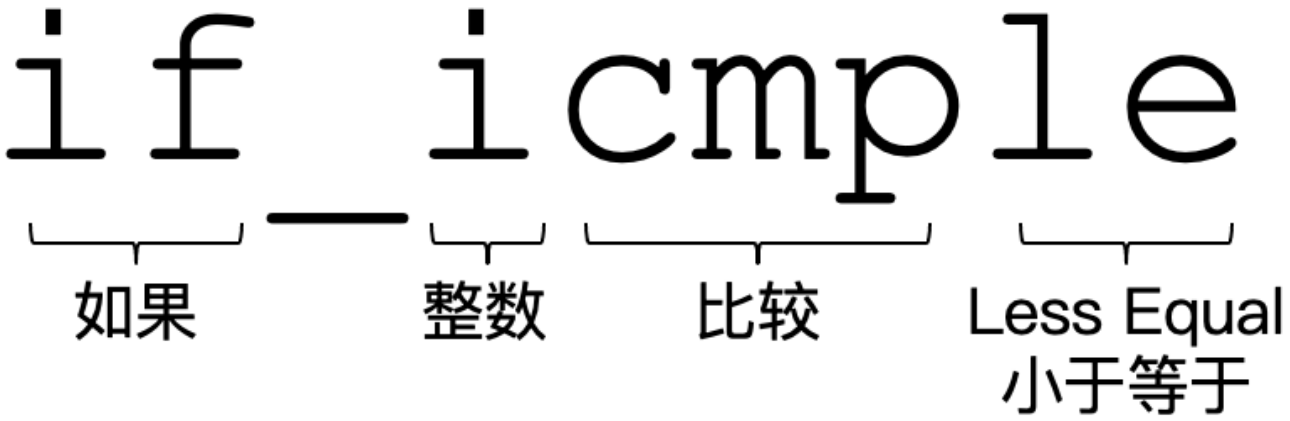
```
public static int foo3(int);
  descriptor: (I)I
  flags: (0x0009) ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=2, args_size=1
       0: iload_0
       1: bipush        10
       3: if_icmple     14
       6: iload_0
       7: bipush        8
       9: iadd
      10: istore_1
      11: goto          19
      14: iload_0
      15: bipush        8
      17: isub
      18: istore_1
      19: iload_1
      20: ireturn
```

栈顶的两个值做<=运算，如果为false，则跳转到地址14。

无条件跳转到地址19。

我主要是想通过这个示例程序，给你展现两条跳转指令。

一条是 if_icmple，它是一条有条件的跳转指令。它给栈顶的两个元素做 <=（less equal，缩写为 le）运算。如果计算结果为真，那么就跳转到分支地址 14。



你可能会发现一个问题，为什么我们的源代码是 > 号，翻译成字节码却变成了 <= 号了呢？没错，虽然符号变了，但其实我们的语义并没有发生变化。

我给你分析一下，在源代码里面，程序用 > 号计算为真，就执行 if 下面的块；那就意味着如果 > 号计算为假，或者说 <= 号为真，则跳转到 else 下面的那个块，这两种说法是等价的。

但现在我们要生成的是跳转指令，所以用 <= 做判断，然后再跳转，就是比较自然了。具体你可以看看我们下文中为 if 语句生成代码的逻辑和相关的图，就更容易理解了。

你在字节码中还会看到另一个跳转指令，是 goto 指令，它是一个无条件跳转指令。

在计算机语言发展的早期，人们用高级语言写程序的时候，也会用很多 goto 语句，导致程序非常难以阅读，程序的控制流理解起来困难。虽然直到今天，C 和 C++ 语言里还保留了 goto 语句。不过，一般不到迫不得已，你不应该使用 goto 语句。

这种迫不得已的情况，我指的是使用 goto 语句实现一些奇特的效果，这些效果是用结构化编程方式（也就是不用 goto 语句，而是用条件语句和循环语句表达程序分支）无法完成的。比如，采用 goto 语句能够从一个嵌套很深的语句块，一下子跳到外面，然后还能再跳进去，接着继续执行！这相当于能够暂停一个执行到一半的程序，然后需要时再恢复上下文，接着执行。

我在说什么呢？这可以跟协程的实现机制关联起来，协程要求在应用层把一个程序停止下来，然后在需要的时候再继续执行。那么利用 C/C++ 的 goto 语句的无条件跳转能力，你其实就可以实现一个协程库，如果你想了解得更具体一些，可以看看我之前的《编译原理实战课》。

总结起来，goto 的这种跳转方式，是更加底层的一种机制。所以，在编译程序的过程中，我们会多次变换程序的表达方式，让它越来越接近计算机容易理解的形式，这个过程叫做**Lower 过程**。而 Lower 到一定程度，就会形成线性代码加跳转语句的代码格式，我们有时候就会把这种格式的 IR 叫做“**goto 格式（goto form）**”。

好了，刚才聊的关于 goto 语句的这些知识点，是为了加深大家对它的认识，希望能够对你的编程思想有所启发。

回到正题，现在我们已经对跳转指令有了基本的认识，那么我就把接下来要用到的跳转指令列出来，你可以看看下面这两张表：

助记符	操作码（16进制）	操作码（2进制）	其他字节	描述
lcmp	94	1001 0100		弹出栈顶的两个值。如果相等，则压栈0；如果value1>value2,则压栈1；其他情况，压栈-1
ifeq	99	1001 1001	2个字节：分支地址	如果栈顶的值等于0，跳转
ifne	9a	1001 1010		如果栈顶的值不等于0，跳转
iflt	9b	1001 1011		如果栈顶的值小于0，跳转
ifge	9c	1001 1100		如果栈顶的值大于等于0，跳转
ifgt	9d	1001 1101		如果栈顶的值大于0，跳转
ifle	9e	1001 1110		如果栈顶的值小于等于0，跳转

助记符	操作码（16进制）	操作码（2进制）	其他字节	描述
if_icmpeq	9f	1001 1111	2个字节：分支地址	弹出栈顶的两个值。如果value1==value2，则跳转到分支地址
if_icmpne	a0	1010 0000		value1!=value2，则跳转
if_icmplt	a1	1010 0001		value1<value2，则跳转
if_icmpge	a2	1010 0010		value1>=value2，则跳转
if_icmpgt	a3	1010 0011		value1>value2，则跳转
if_icmple	a4	1010 0100		value1<=value2，则跳转
goto	a7	1010 0111		无条件跳转到分支地址
iinc	84	1000 0100	2个字节，第一个字节是变量的下标，第二个字节是需要增加的值	把指定下标的变量，增加指定的值（可以是负数）

如果后面要增加对浮点数和对象引用的比较功能，我们可以再增加一些指令。但由于目前我们还是只处理整数，所以这些指令就够了。

接着，我们就修改一下字节码生成程序和虚拟机中的执行引擎，让它们能够支持 if 语句和 for 语句。

为 if 语句和 for 循环语句生成字节码

让 if 语句生成字节码的代码你可以参考 visitIfStatement 方法。在这个方法里，我们首先为 if 条件、if 后面的块、else 块分别生成了字节码。

```
//条件表达式的代码
let code_condition:number[] = this.visit(ifstmt.condition);
//if块的代码
let code_ifBlock:number[] = this.visit(ifstmt.stmt);
//else块的代码
let code_elseBlock:number[] = (ifstmt.elseStmt == null) ? []: this.visit(ifstm
```

接下来，我们要把这三块块代码拼接到一起，这就需要计算出每块代码的起始地址，以便跳转语句能够做出正确地跳转。比如，if 块的起始位置是代码条件块的长度加 3，这个 3 是我们生成的跳转指令的长度。

```
let offset_ifBlock:number = code_condition.length + 3;                    //if语
let offset_elseBlock:number = code_condition.length + code_ifBlock.length + 6;
let offset_nextStmt:number = offset_elseBlock + code_elseBlock.length;    //if语
```

计算出每块代码的起始地址后，接下来我们就说说如何生成跳转指令。

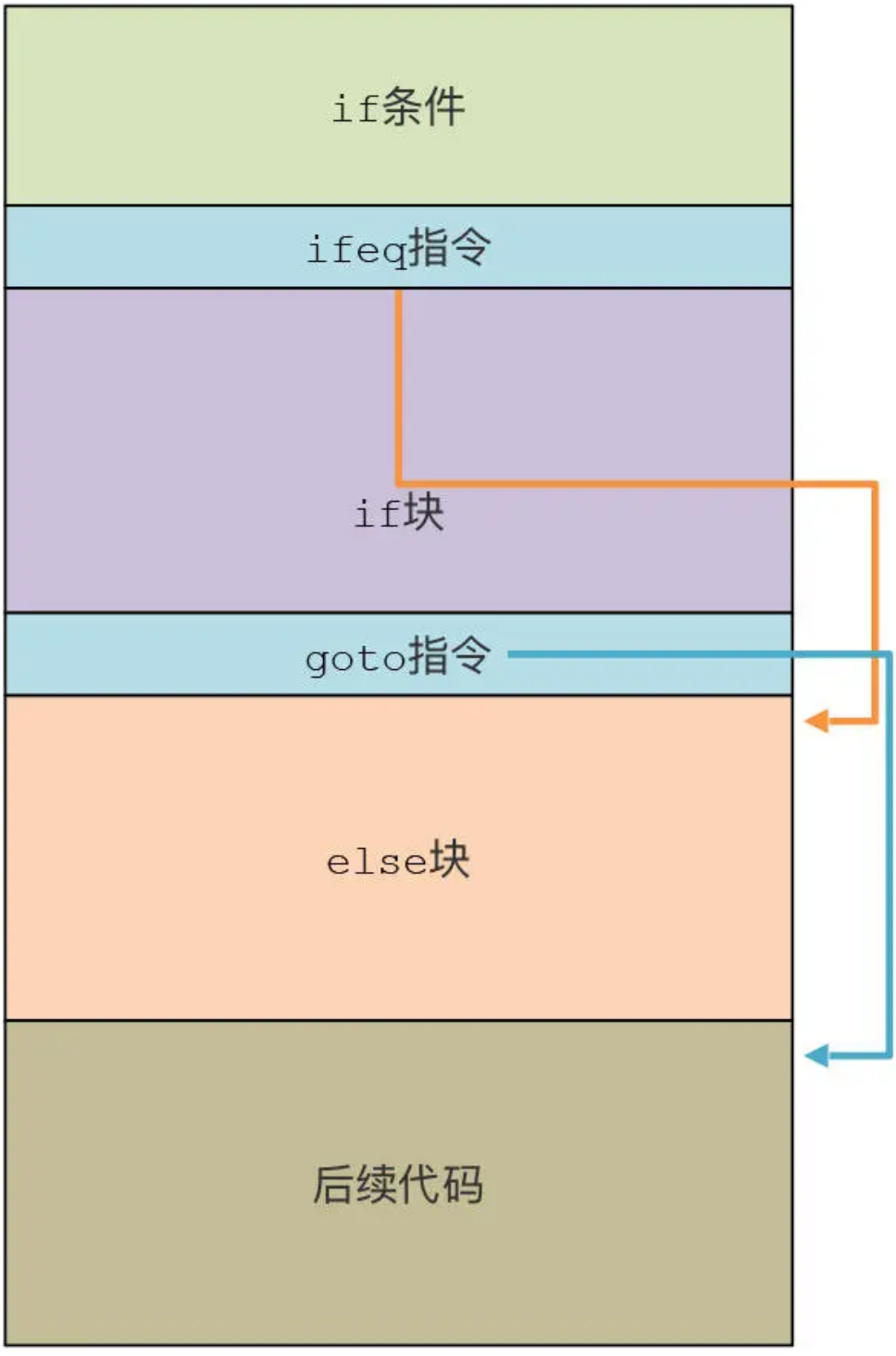
在前面 Java 生成的字节码中，我们看到编译器对于“a>10”生成了一条“if_icmple”指令。但我为了生成代码的逻辑更简单，采用了另一个实现思路。也就是先对 a>10 做计算，如果结果为真，就在栈顶放一个 1，否则就放一个 0。

接下来，我们再生成一条 ifeq 指令，根据栈顶的值是 1 还是 0 来做跳转。那么这个 ifeq 指令就要占据 3 个字节，其中操作码占 1 个字节，操作数占两个字节。我的算法生成了两条指令，你也可以思考一下如何能够像 Java 编译器那样只生成一条指令。

计算好偏移量以后，我们就可以把 condition 块、if 块和 else 块的代码拼接成完整的 if 语句的代码了：

```
//条件
code = code.concat(code_condition);

//跳转:去执行else语句块
code.push(OpCode.ifeq);
code.push(offset_elseBlock>>8);
code.push(offset_elseBlock);

//条件为true时执行的语句
code = code.concat(code_ifBlock);

//跳转：到整个if语句之后的语句
code.push(OpCode.goto);
code.push(offset_nextStmt>>8);
code.push(offset_nextStmt);

//条件为false时执行的语句
code = code.concat(code_elseBlock);
```

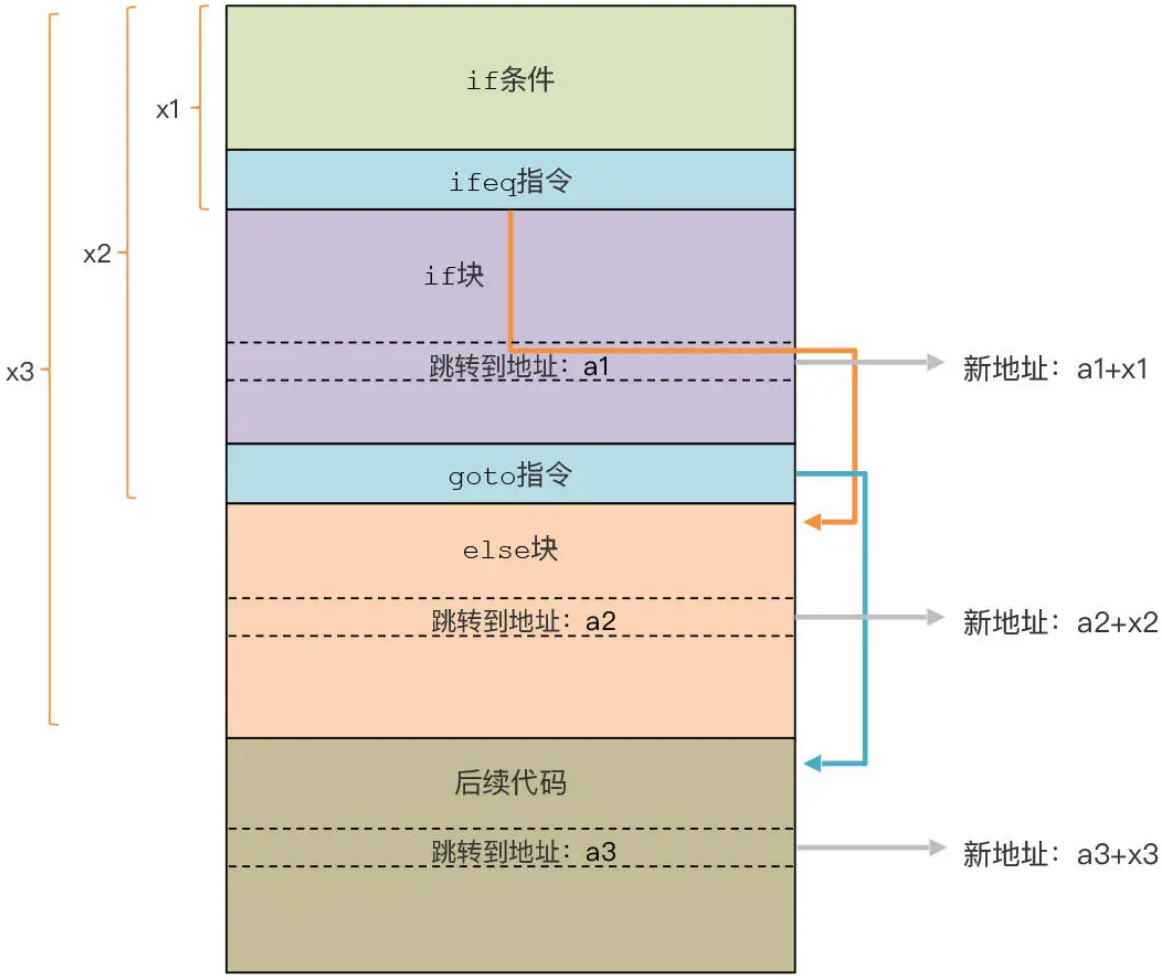
你也看到了，在示例代码中，我们生成了两个跳转指令，分别是 ifeq 指令和 goto 指令，这样就能保证所有语句块之间正确的跳转关系。


if条件
ifeq指令
if块
goto指令
else块
后续代码

那到这里，是不是就大功告成了，为 if 语句生成了正确的字节码了呢？

不是的。你仔细体会一下，是否有点什么不对劲？哪里不对劲呢？

你看，在我们生成的跳转指令中，一定要设置好正确的跳转地址，否则代码就错误了。可是，在 if 块和 else 块的代码中可能也包含了跳转指令呀。现在你把 if 语句中的各个块拼成了一个整体的字节码数组，那么各个块中包含的跳转指令的地址就错了呀。原来的跳转地址，都是基于各自的字节码数组中的位置，现在拼成一个新数组，这个位置也就变了。



所以，这些跳转指令的跳转地址，都需要做一下调整，加上我们计算出来的偏移量，来保证整体的跳转不出问题。

复制代码

```
1 this.addOffsetToJumpOp(code_ifBlock, offset_ifBlock);
```

```
this.addOffsetToJumpOp(code_elseBlock, offset_elseBlock);
```

其实，这个调整跳转指令地址的过程，是每次做代码拼接的时候都会被调用的。比如，当前 if 语句生成的代码，在上一级的 AST 节点中做拼接的时候，也需要做地址的调整，这样就会保证为函数最终生成的字节码中，每个跳转指令的地址都是正确的。

实现完了 If 语句，你可以再实现一下 for 循环语句。for 循环语句的实现思路也是差不多的，都是先生成各个部分的代码，然后再正确地拼接到一起。

好了，到目前为止，我们已经扩展了字节码编译器的功能，让它能够为 if 语句和 for 循环语句正确地生成字节码了。接下来，我们再扩展一下字节码解释器的功能，让它能够正确地运行这些跳转指令就行了。

升级字节码虚拟机

对于 ifeq、ifneq、if_cmplt 等跳转指令，解释器的处理逻辑差不多都是一样的。它在遇到这些指令的时候，都会计算一下栈顶的值是否符合跳转条件。如果符合，就计算出一个新的指令地址并跳转过去。否则，就顺序执行下一条指令就可以了。

复制代码

```
case OpCode.ifeq:
    byte1 = code[++codeIndex];
    byte2 = code[++codeIndex];
    //如果栈顶的值是0，那么就计算一个新的指令地址，以便跳转过去
    if(frame.oprandStack.pop() == 0){
        codeIndex = byte1<<8|byte2;
        opCode = code[codeIndex];
    }
    //否则，就继续执行下一条指令
    else{
        opCode = code[++codeIndex];
    }
```

在增加了对跳转指令的支持以后，我们就完成了对虚拟机的升级工作。**现在，我们已经拥有两个运行时了，一个是 AST 解释器，一个是字节码虚拟机。**在我们启动字节码虚拟机的设计之前，我们曾经希望它能提升程序运行的性能。那现在是否达到了这个目标呢？让我们测试一下吧！

做一下性能比拼

我们仍然采用斐波那契数列的例子来做性能测试。这个例子的好处，就是随着数列的变大，所需要的计算量会指数级地上升，所以更容易比较出性能差异来。现在，我们用 AST 解释器和字节码虚拟机都来执行下面的代码：

复制代码

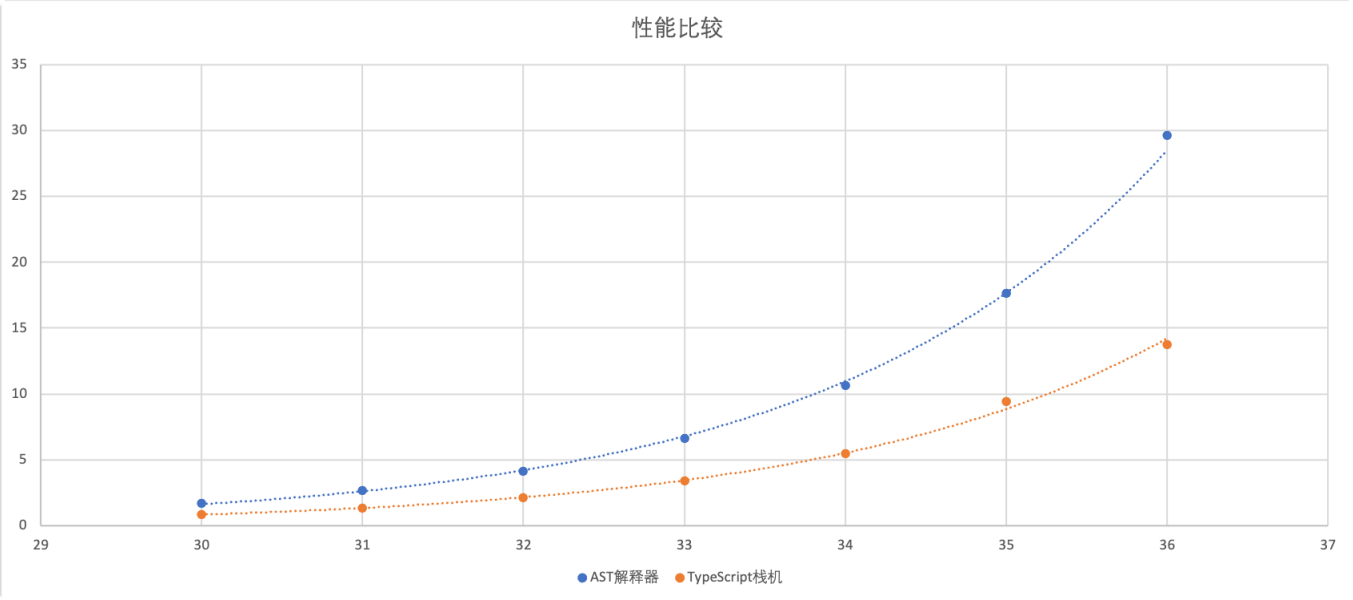
```
function fibonacci(n:number):number{
    if (n <= 1){
        return n;
    }
    else{
        return fibonacci(n-1) + fibonacci(n-2);
    }
}

for (let i:number = 0; i< 26; i++){
    println(fibonacci(i));
}
```

我让 n 依次取不同的数值，分别测量出两个运行时所花费的时间，做成了一个表格。

n	AST解释器	TypeScript栈机
30	1.66	0.846
31	2.649	1.324
32	4.104	2.126
33	6.604	3.374
34	10.654	5.477
35	17.664	9.427
36	29.613	13.752

你可以从表格中看到，TypeScript 栈机所花费的计算时间，基本上只是 AST 解释器的一半。所以，我们确实按照预期获得了性能上的提升。

如果把表格画成图表，就能更清晰地显示出运行速度变化的趋势。你能看出，这两条线都是指数上升的，并且栈机花费的时间始终只有 AST 解释器的一半。



好了，我们的第一个版本的虚拟机已经完成了。在性能方面，也确实如我们所愿，获得了重大的提升，可以说比较圆满地完成了我们这节课的目标！

课程小结

这节课，我们主要实现了对 if 语句和 for 循环语句的支持。在这个过程中，我们学习了一类新的指令，也就是跳转指令。你如果多接触几种指令集就会发现，包括物理 CPU 的指令集在内的各种指令集，都有跳转指令，并且名称也都差不多。所以，我们这里学到的知识，也有助于后面降低你学习汇编语言的难度。

在为跳转指令生成代码的时候，关键点是**要正确地计算跳转地址，**地址是相对于这个函数的字节码数组的开头位置的偏移量。只有在为整个函数都生成了字节码以后，我们才能计算出所有的基本块的准确起始地址。

最后，我们还对虚拟机的性能做了测试。现在的性能只提高了一倍，当生成的斐波那契数列比较大的时候，我还是觉得挺慢的。那么，我们能否有办法进一步提升虚拟机的性能呢？像 JVM 和 V8 这样的虚拟机，都是用 C++ 写的。那么，如果采用 C/C++ 这样的系统级语言写一个新版本的虚拟机，会不会获得性能上更大的提升呢？那么在下一节课，我们就动手实现一个 C 语言版本的虚拟机吧。

思考题

又来到了我们例行的思考题环节，我想问下，现在你对我们虚拟机的性能测试结果满意吗？你觉得我们还可以做哪些改进，来提升 TypeScript 版本的虚拟机的性能呢？欢迎在留言区留言。

感谢你和我一起学习，也欢迎你把这节课分享给更多对字节码虚拟机感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

这节课的示例代码在这里！

上一篇 08 | 基于TypeScript的虚拟机（一）：实现一个简单的栈机

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。