



下载APP



32 | 函数式编程第1关：实现高阶函数

2021-10-27 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 12:06 大小 11.10M



你好，我是宫文学。

前面三节课，我们探讨了怎么在现代语言中实现面向对象编程的特性。面向对象是一种重要的编程范式。还有另一种编程范式，也同样重要，并且近年来使用得很多，这就是函数式编程。从今天这节课开始，我们就来实现一下函数式编程。

函数式编程思想其实比面向对象编程思想的历史更长，早期的 Lisp 等语言都是函数式编程语言。像 JavaScript 等后来的语言，也继承了 Lisp 语言在函数式编程方面的思想，对函数式编程也有不错的支持。



近年，函数式编程思想得到了一定程度的复兴，部分原因是由于函数式编程能够更好地应对大规模的并发处理。我自己最近参与的项目，也在全面使用一门函数式编程语言，这也

是对函数式编程的优势的认可。此外，像 Erlang 这种能够开发高可靠性系统的函数式编程语言，也一直是感兴趣的研究对象。

对于函数式编程这个话题，很多书和文章都对它有过讲解。我在《编译原理实战课》的 [第 39 节](#)，也对函数式编程特性的一些技术点做了分析。在我们的这门课里，因为要动手实现出来，所以目标不能太大，我们就挑几个最核心的技术点来实现一下，让你对函数式编程的底层机制有一次穿透性的了解。

今天这节课，我们主要来实现高阶函数的特性。对于函数式编程来说，高阶函数是实现其他功能的基础，属于最核心的技术点。那么，我们就先分析一下什么是高阶函数。

高阶函数的例子

高阶函数的核心思想，是**函数本身可以当做数据来使用**，就像 number 数据和 string 数据那样。那既然可以当做数据使用，那自然可以用它来声明变量、作为参数传递给另一个函数，以及作为返回值从另一个函数中返回。如果一门计算机语言把函数和数据同等对待，这时候我们说函数是一等公民（First-class Citizen）。

我用 TypeScript 写了一个 reduce 函数的例子，带你来感受一下高阶函数的特性。这个函数能够遍历一个 number 数组，并且返回一个 number 值。

 复制代码

```
1 //reduce函数：遍历数组中的每个元素，最后返回一个值
2 function reduce(numbers:number[], fun:(prev:number,cur:number)=>number):number
3     let prev:number = 0;
4     for (let i = 0; i < numbers.length; i++){
5         prev = fun(prev, numbers[i]);
6     }
7     return prev;
8 }
9
10 //累计汇总值
11 function sum(prev:number, cur:number):number{
12     return prev + cur;
13 }
14
15 //累计最大值
16 function max(prev:number, cur:number):number{
17     if (prev >= cur)
18         return prev;
19     else
```

```
20         return cur;
21     }
22
23     let numbers = [2,3,4,5,7,4,5,2];
24
25     println(reduce(numbers, sum));
26     println(reduce(numbers, max));
```

这个 reduce 函数很有意思的一点，是它能接受一个函数作为参数。在每遍历一个数组元素的时候，都会调用这个传进来的函数。根据传入的函数不同，reduce 函数能完成不同的功能。当传入 max 函数的时候，reduce 函数能返回数组元素的最大值；而当传入 sum 函数的时候，则能返回数组元素的汇总值。

这个例子能够部分体现函数式编程的优势：**把系统的功能拆解成函数，再灵活组合。**

那这些高阶函数的特性具体怎么实现呢？按照惯例，我们还是先看看在编译器前端方面，我们要做什么工作。

修改编译器前端


要实现上面的功能，编译器前端需要增加新的语法规则，并做一些与函数类型有关的语义处理工作。

首先我们看看语法方面的工作。

我们需要要增加与函数类型有关的语法，支持示例程序中的 (prev:number, cur:number)=>number 这样的格式。

涉及的语法规则如下：

```
1 type_ : unionOrIntersectionOrPrimaryType | functionType;
2 functionType : '(' parameterList? ')' '=>' type_;
```

 复制代码

你看到，我们增加了新的类型表达式。与此相对应的，我们也要增加新的 AST 节点：FunctionTypeExp，用于记录解析出来的函数类型信息。

接着，我们再看看语义分析方面的工作。

在语义分析方面，我们需要扩展现在的类型系统，来支持函数类型。对函数类型表达式的AST节点进行解析后，我们就能够生成对应的函数类型了。函数类型的设计如下：

[复制代码](#)

```
1 export class FunctionType extends Type{
2     returnType:Type;    //返回值类型
3     paramTypes:Type[];  //参数的类型
4     ...
5 }
```

这样的话，变量、参数的类型，就可以设置为这种函数类型。接下来，我们需要对类型计算和类型检查的代码升级。比如，联合类型中也可以包含函数类型，给函数类型的变量赋值的时候，我们要检查类型是否匹配。

另外，我们在函数的引用消解方面也要做一些工作。比如，在示例程序中，我们使用了 `fun(prev, cur)` 这样的表达式。在这节课之前，我们肯定要把 `fun` 关联到一个具体的函数声明。但现在，`fun` 有可能是一个具体的函数，也有可能是一个函数类型的变量，这在引用消解的时候要区分开。`fun` 消解后，关联的符号是一个变量符号，而不是一个函数符号。

好了，编译器前端的工作就是这些。基本上没有太大的技术难度，其实也没有新的技术点，基本上是在原来的技术框架下做扩展，但工作量还是有的。

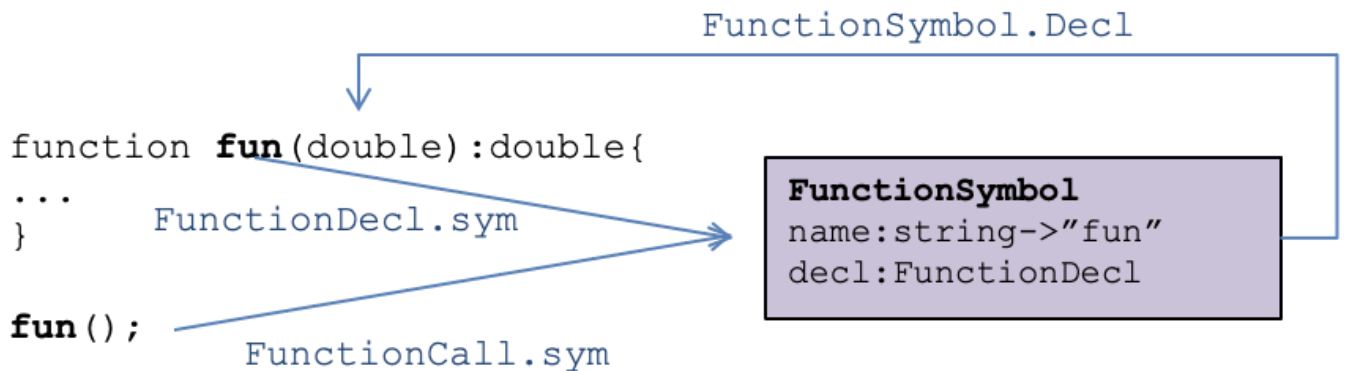
具体的实现，你可以参照 [parser.ts](#) 和 [semantic.ts](#)。

相比编译器前端而言，运行时中涉及的新技术点就更多一些了。我们先看看AST解释器的运行时。

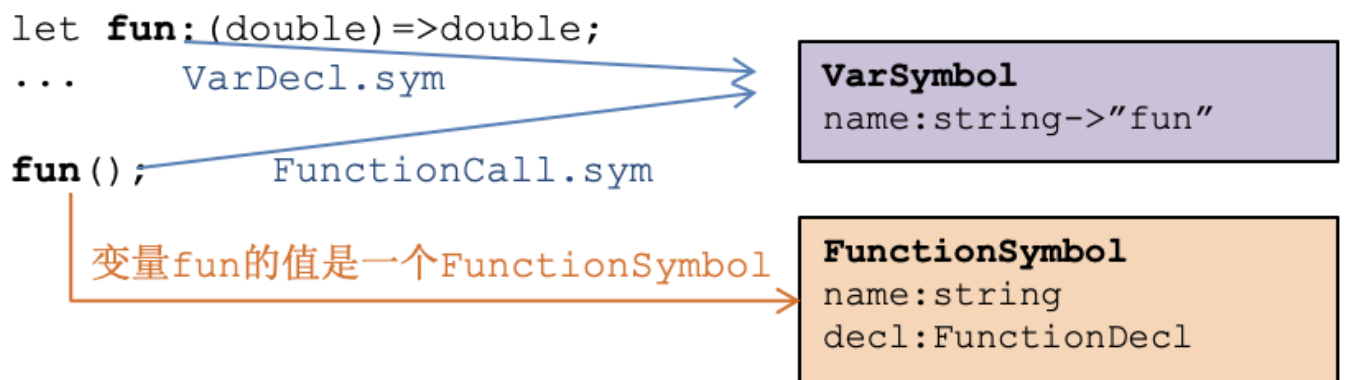
升级AST解释器

在AST解释器中，我们需要把函数当作值来传递。那这个值应该是什么呢？你可以思考一下。

在我们当前的实现中，其实可以直接把函数的符号当作变量值来传递就行了。你看，我们之前调用函数时候，这个 `FunctionCall` 表达式已经被消解，从而其 `sym` 属性就指向了一个具体的函数符号，通过这个函数符号就能找函数声明，从而解释执行这个函数。



现在我们可以用一个变量表示一个函数，那么这个变量的 `sym` 属性指向的是变量声明的地方，而变量的值才是该函数的符号。这个时候，我们可以取出变量的值，也就是一个函数符号，就可以解释执行这个函数了。



通过这样的分析，你应该可以弄清楚直接调用函数和调用一个函数变量的区别了。前者要访问其 `sym` 属性来获得函数符号，而后者是通过变量的值来获得函数符号，进而解释执行该符号所关联的函数声明的 AST。

AST 解释器的参考实现，我仍然放在了 [play.ts](#) 中。你可以运行一下 [example_fp.ts](#) 示例程序，看看能不能得到正确的运行结果。

好，接下来我们再看看在编译成可执行程序的情况下，如何使用函数式编程特性。

编译成可执行程序

我们刚刚已经说过，要支持函数式编程特性，最重要的是能够把函数当做值来传递。在 AST 解释器中，这个值是函数符号。那在编译成汇编代码的时候，我们用什么来代表一个函数呢？

你已经接触过很多汇编代码了，我相信你肯定知道，其实像函数这种高抽象度的语言要素，在 Lower 到汇编代码这个层面时，只是一个标签而已。在汇编代码转换成机器码的时候，这个标签就是代表了一段代码在程序文本段的一个地址而已。**所以说，传递一个函数，就是传递一个函数地址。**

当你在程序里显式地调用一个函数的时候，只要让我们生成的汇编代码，直接跳转到这个函数的标签就行了。但当我们调用一个函数型的变量的时候，实质上就是要跳转到这个变量所存储的地址中。也就是说，这个地址不是在编译时能够确定的，而是在运行时根据函数型变量的值来确定。

这种在运行时来确定被调用的函数的机制，我们其实在上一节已经部分接触过了。在调用类的方法的时候，具体的方法地址，要去查 vtable。不过，函数式编程中来获取函数地址就变得更灵活了，干脆是通过变量和参数来传递的。

我们还是用 C 语言写一个例子，看看这种地址传递是如何实现的。这里你可能会问了：我没听说过 C 语言是函数式编程语言呀？为什么可以用 C 语言写函数式编程的例子呢？

不着急。你看一下代码就知道了。在下面的示例程序中，我们用 C 语言重写了上面的示例程序，也包括 reduce、max 和 sum 这几个函数。

[复制代码](#)

```
1 #include "stdio.h"
2
3 double reduce(double* numbers, int length, double(*fun)(double, double)){ //使
4     int prev = 0;
5     for (int i = 0; i < length; i++){
6         prev = fun(prev, numbers[i]);
7     }
8     return prev;
9 }
10
11 double max(double prev, double cur){
12     if (prev >= cur)
13         return prev;
14     else
```

```

15         return cur;
16     }
17
18     double sum(double prev, double cur){
19         return prev + cur;
20     }
21
22     int main(){
23         double numbers[8] = {2,3,4,5,7,4,5,2};
24         printf("%lf\n", reduce(numbers, 8, sum));
25         printf("%lf\n", reduce(numbers, 8, max));
26     }
27

```

在 reduce 函数中，最后一个参数是一个函数指针，这个函数指针会接受两个 double 类型的输入参数，返回一个 double 值。在 C 语言中，函数指针能够被作为数值传递，而这个指针的值实际上就是一个函数的入口地址。

所以说，虽然 C 语言并不强调函数式编程能力，其实你仍然可以用它来体现函数式编程思想的。就像 C 语言也不是面向对象的语言，但你仍然可以用它来体现面向对象编程思想。

那我们现在看看这段 C 语言的代码编译成汇编代码是什么样子。你可以用 “clang -S fp.c -o fp.s” 编译成汇编代码。然后，你可以查看 main 函数中下面这几行代码，这几行代码代表了 “reduce(numbers, 8, sum)” 。其中第三行代码就是获取了 sum 函数的地址，并作为 reduce 函数的第三个参数。

```

movq    -88(%rbp), %rdi ← 参数1: 数组地址    ## 8-byte Reload
movl    $8, %esi ← 参数2: 数组长度
leaq    _sum(%rip), %rdx ← 参数3: sum函数的地址
callq   _reduce

```

你也可以再看看 reduce 函数中的代码片段。其中 “callq *%rax” 就是把 sum 或 max 函数的地址放在了 rax 寄存器里，然后跳转到这个地址去执行就行了。

```

_reduce:
    .cfi_startproc
## %bb.0:
    pushq   %rbp

```

```

pushq    %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq     %rsp, %rbp
.cfi_def_cfa_register %rbp
subq     $32, %rsp
movq     %rdi, -8(%rbp)
movl     %esi, -12(%rbp)
movq     %rdx, -24(%rbp) ← 参数3: fun
                           函数的地址
movl     $0, -28(%rbp)
movl     $0, -32(%rbp)

LBB0_1:
movl     -32(%rbp), %eax
cmpl     -12(%rbp), %eax
jge LBB0_4

## %bb.2:
movq     -24(%rbp), %rax ← 参数3: fun
                           函数的地址
cvtsi2sdl -28(%rbp), %xmm0
movq     -8(%rbp), %rcx
movslq   -32(%rbp), %rdx
movsd    (%rcx,%rdx,8), %xmm1
callq    *%rax ← 调用fun函数

```

好了，现在我们通过剖析，已经弄清楚了如何把函数作为值传递了。不过，为了能够编译这节课的示例程序，我们还要实现一个小的技术点，就是能够正确的获取数组的长度，也

就是示例代码中的“`numbers.length`”。这里要使用点符号表达式，访问数组对象的 `length` 属性。

这个话题我们在前几节课讨论过。我们目前已经从自定义的 `class` 对象中获取对象属性。而对于数组和字符串这样的内置数据类型，我们也可以访问它的一些属性。要访问这些属性，我们也是把它们翻译成内存地址就可以了，也就是在 `PlayObject` 的地址基础上加上一定的偏移量。

在实现了这个技术点之后，我们就可以编译并运行示例程序了。你也可以自己动手写几个程序，试一试高阶函数的特性。

具体的实现，我放在了 [@asm_x86-64.ts](#) 中。

课程小结

今天这节课，我们初步实现了函数式编程的一个特性，把函数变成了跟其他数据一样的等公民，从而可以支持高阶函数。这节课，我们记住这几个知识点就好了：

首先，为了把函数当做数据，我们需要支持函数类型，让我们可以声明函数类型的变量。

第二，在 AST 解释器中，函数类型的变量的值，可以表示为函数符号。

第三，在编译成可执行文件时，函数类型的变量的值，会被 Lower 成函数的地址。

在下一节课里，我们将会继续探究函数式编程的特性。

思考题

在函数式编程语言里，我们经常使用 `lambda` 表达式来表示一个函数。你能否分析一下，要支持 `lambda` 表达式，我们需要做些什么工作？

欢迎你把这节课分享给更多对函数式编程感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

[@这节课的代码目录在这里！](#)

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 31 | 面向对象编程第2步：支持继承和多态

1024 活动特惠

VIP 年卡直降 ¥2000

新课上线即解锁，享 365 天畅看全场

超值拿下 ¥999



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。