



下载APP



## 33 | 函数式编程第2关：实现闭包特性

2021-10-29 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 16:24 大小 15.02M



你好，我是宫文学。

上节课，我们实现了函数式编程的一个重要特性：高阶函数。今天这节课，我们继续来实现函数式编程的另一个重要特性，也就是闭包。

闭包机制能实现信息的封装、缓存内部状态数据等等，所以被资深程序员所喜欢，并被广泛用于实现各种框架、类库等等。相信很多程序员都了解过闭包的概念，但往往对它的内在机制不是十分清楚。




今天这节课，我会带你了解闭包的原理和实现机制。在这个过程中，你会了解到闭包数据的形成机制、词法作用域的概念、闭包和面向对象特性的相似性，以及如何访问位于其他栈帧中的数据。

首先，让我们了解一下闭包的技术实质，从而确定如何让我们的语言支持闭包特性。

## 理解闭包的实质

我们先通过一个例子来了解闭包的特点。在下面的示例程序中有一个 ID 的生成器。这个生成器是一个函数，但它把一个内部函数作为返回值来返回。这个返回值被赋给了函数类型的变量 `id1`，然后调用这个函数。

 复制代码

```
1 function idGenerator():number{//()=>number{
2     let nextId = 0;
3
4     function getId(){
5         return nextId++; //访问了外部作用域的一个变量
6     }
7
8     return getId;
9 }
10
11 println("\nid1:");
12 let id1 = idGenerator();
13 println(id1()); //0
14 println(id1()); //1
15
16 //新建一个闭包，重新开始编号
17 println("\nid2:");
18 let id2 = idGenerator();
19 println(id2()); //0
20 println(id2()); //1
21
22 //闭包可以通过赋值和参数传递，在没有任何变量引用它的时候，生命周期才会结束。
23 println("\nid3:");
24 let id3 = id1;
25 println(id3()); //2
```

然后神奇的事情就发生了。每次你调用 `id1()`，它都会返回一个不同的值，依次是 0、1、2.....

为什么每次返回的值会不一样呢？

你看这个代码，内部函数 `getId()` 访问了外部函数的一个本地变量 `nextId`。当外部函数退出以后，内部函数仍然可以使用这个本地变量，并且每次调用内部函数时，都让 `nextId` 的值加 1。这个现象，就体现了闭包的特点。

总结起来，闭包是这么一种现象：**一个函数可以返回一个内部函数，但这个内部函数使用了它的作用域之外的数据**。这些作用域之外的数据会一直伴随着该内部函数的生命周期，内部函数一直可以访问它。

说得更直白一点，就是**当内部函数被返回时，它把外部作用域中的一些数据打包带走了，随身携带，便于访问**。

这样分析之后你就明白了。为了支持闭包，你需要让某些函数有一个私有的数据区，用于保存一些私有数据，供这个函数访问。在我们这门课里，我们可以把这个函数专有的数据，叫做闭包数据，或者叫做闭包对象。

然后我们再运行这个示例程序，并分析它的输出结果：

```
id1:
```

```
0
```

```
1
```

```
id2:
```

```
0
```

```
1
```

```
id3:
```

```
2
```

你会发现这样一个事实：id1 和 id2 分别通过调用 idGenerator() 函数获得了一个内部函数，而它们各自拥有自己的闭包数据，是互不干扰的。

从这种角度看，**闭包有点像面向对象特性。每次 new 一个对象的时候，都会生成不同的对象实例。**实际上，在函数式语言里，我们确实可以用闭包来模拟某些面向对象编程特性。不过这里你要注意，并不是函数所引用的外部数据，都需要放到私有的数据区中的。我们可以再通过一个例子来看一下。

我把前面的示例程序做了一点修改。这一次，我们的内部函数可以访问两个变量了。

[复制代码](#)

```
1 //编号的组成部分
2 let segment:number = 1000;
3
4 function idGenerator():()=>number{
5     let nextId = 0;
6
7     function getId(){
8         return segment + nextId++; //访问了2个外部变量
9     }
10
11     //在与getId相同的作用域中调用它
12     println("getId:" + getId());
13     println("getId:" + getId());
14
15     //恢复nextId的值
16     nextId = 0;
17
18     return getId;
19 }
20
21 println("\nid1:");
22 let id1 = idGenerator();
23 println(id1()); //1000
24 println(id1()); //1001
25
26 //修改segment的值，会影响到id1和id2两个闭包。
27 segment = 2000;
28
29 //新建一个闭包，重新开始编号
30 println("\nid2:");
31 let id2 = idGenerator();
32 println(id2()); //2000
33 println(id2()); //2001
34
35 //闭包可以通过赋值和参数传递，在没有任何变量引用它的时候，生命周期才会结束。
```

```
36 println("\nid3:");  
37 let id3 = id1;  
38 println(id3()): //2002
```

你看，我们增加了一个全局变量 `segment`，而内部函数最后生成的 `id` 等于 `segment + nextId`。比如，当 `segment=1000`，`nextId=8` 的情况下，生成的 `id` 就是 1008。

并且，我们在两个不同的地方调用了这个内部函数。一个地方是在 `idGenerator` 内部，也就是声明 `getId()` 的函数作用域，而另一个调用则在全局程序中。在这两个情况下，函数访问变量数据的方式是不同的。

你能看到，在 `idGenerator` 内部调用函数 `getId()` 时，我们不需要为它设置私有数据区。为什么呢？因为 `getId()` 运行时所处的作用域和声明它的作用域是一样的，所以可以直接访问 `nextId` 和 `segment` 这两个变量。并且，`nextId` 和 `segment` 的生存期都超过了调用 `getId()` 的时间区间，所以也不会出现像在第一个示例程序中，`idGenerator` 中的 `nextId` 变量会随着 `idGenerator` 运行结束而消失的情形。

但在外部来调用 `id1()`、`id2()` 的时候，我们就需要把 `nextId` 变量放到私有数据区了。这是因为，调用 `id1()` 和 `id2()` 所在的作用域，与声明 `getId()` 的时候是不同的。在运行时的作用域中，已经“看不到” `nextId` 变量了，所以必须把它放到私有数据区。不过，在这个作用域仍然是能够“看到” `segment` 变量的，所以 `segment` 变量不需要放到私有数据区。

这样看起来，**闭包现象跟作用域密切相关**。所以这里我再引申一下，介绍一个术语，叫做**词法作用域 (Lexical Scope)**，它的意思是声明一个符号时所在的作用域。对于函数来说，它总是在函数声明的作用域中绑定它所引用的各种变量。这样，我们以后不管在哪里执行这个函数，都会“记住”它们，也就是访问这个作用域中的变量的值。现代大部分语言使用的都是词法作用域。

词法作用域又叫做静态作用域。与静态作用域相对的呢，是动态作用域。动态作用域中，函数中使用的变量不是在语义分析阶段绑定好的，而是可以在运行期动态地绑定。比如，如果 TypeScript 使用的是动态作用域，而我们在调用 `id1()`、`id2()` 之前，声明了一个叫做 `nextId` 的变量，那么 `id1()` 和 `id2()` 执行的时候，其函数内部的那个 `nextId` 变量，就会跟刚声明的这个 `nextId` 变量绑定。那这个时候，也就不需要闭包了。

通过对比词法作用域和动态作用域，我希望你能加深对闭包的理解。对于学习了这门课程的你来说，理解它们的差别应该更容易。因为你已经清晰地知道，变量引用的消解都是发生在语义分析阶段，不会在运行期来改变这种引用关系。

最后，在示例程序中，你还会看到 id3 这个函数类型的变量，并且我们把 id1 赋给了 id3。id1 和 id3 引用的是同一个闭包。所以虽然我们不再使用变量 id1 了，但闭包所占用的内存并不会释放。并且，你还可以把这个闭包以参数的方式传给其他函数，甚至再被包含在其他闭包里。只要还有变量在使用这个闭包，那它的内存就不会被释放。这一点跟面向对象特性中的对象也是完全一致的。

好了，我们花了不少的篇幅来分析闭包的实质。但这些分析不是白费的。基于这些分析，你就可以迅速拿出实现方案来。

现在，就让我们直接上手试试。按照惯例，我们仍然要先修改编译器的前端。

## 修改编译器前端

在编译器的前端方面，我们主要是做一些语义分析工作。

根据前面的分析，**首先我们要分析出一个函数里引用的变量中，哪些是在函数之外声明的**。这项工作比较简单，因为我们在引用消解的时候，已经能够把变量的引用和变量的声明建立起关联关系，并且能够知道每个变量是在哪个作用域里声明的。

你可以用 “node play example\_closure.ts -v” ，把示例程序的符号表打印出来，看看每个变量所属的作用域。这样，你就可以很容易地分析出内部函数所引用的外部变量。



符号表：

Scope of block

```
idGenerator{Function, local var count:1}
id1{Variable}
id2{Variable}
id3{Variable}
```

Scope of function: idGenerator

{empty}

Scope of block

变量符号

**nextId{Variable}**

getId{Function, local var count:0}

Scope of function: getId

{empty}

Scope of block

{empty}

nextId在外层函数的Scope中。

语义分析后的AST，注意变量和函数已被消解：

Prog

FunctionDecl idGenerator

Return type: FunctionType: ()number

Block:

VariableStatement

**VariableDecl nextId:integer**

0:integer

变量声明

FunctionDecl getId

Return type: void

Block:

ReturnStatement

PostFix Unary:Inc:0, constValue:0

**Variable: nextId:integer, LeftValue,**

变量引用

constValue:0, resolved

在完成第一项分析以后，我们还要做第二个分析。也就是说，我们要识别出哪些变量仍然能够被访问到，而哪些是不能的。

这次分析是在调用外部函数并返回闭包的时候进行的。对于那些能够访问到的变量，我们可以继续访问，比如示例程序中的 `segment` 变量。而对于哪些已经不能够访问到的变量，我们则要创建一个私有的数据区来保管它们，比如示例程序中的 `nextId` 变量。

语义分析的参考实现，你可以参见 [semantic-ts](https://github.com/geekbang/semantic-ts)。

在语义分析之后，我们继续来修改 AST 解释器，让它支持闭包特性。

## 修改 AST 解释器

在 AST 解释器里，为了让闭包正常运行，我们需要让函数能够正确访问外部的变量。根据我们前面的分析，这又分成两种情况。第一种情况，是要支持函数的私有数据区，用来保存 nextId 这样的变量。第二种情况，是能够访问其他函数栈帧中的变量，比如 segment 变量。

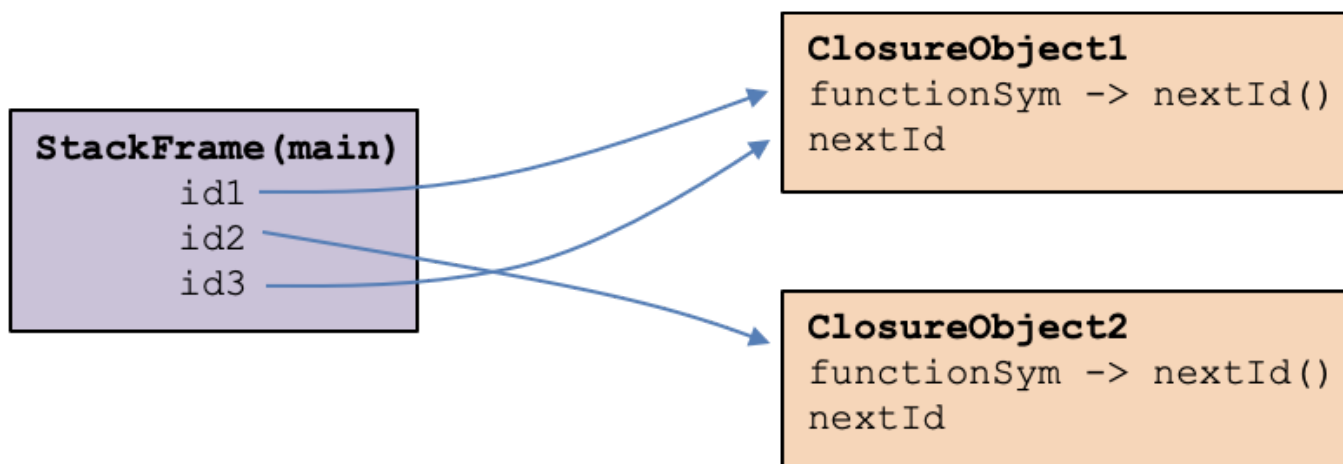
**首先我们看看第一种情况，就是为闭包设计私有的数据区。**

这里，我设计了一个 ClosureObject 数据结构，这个数据结构和 PlayObject、StackFrame 的设计相差不大，都用了一个 Map 来保存变量的数据，闭包函数可以从这里访问私有数据。

[复制代码](#)

```
1 //闭包对象
2 class ClosureObject{
3     functionSym:FunctionSymbol;
4     data:Map<Symbol,any> = new Map();
5     constructor(functionSym:FunctionSymbol){
6         this.functionSym = functionSym;
7     }
8 }
```

在示例程序的栈帧中，id1 和 id3 指向一个相同的 ClosureObject，而 id2 指向另一个 ClosureObject。当给函数变量赋值时，我们就把这个闭包对象赋给新的变量。闭包对象中包含了一个 FunctionSym 引用，这样解释器就知道要运行哪个函数的代码了。



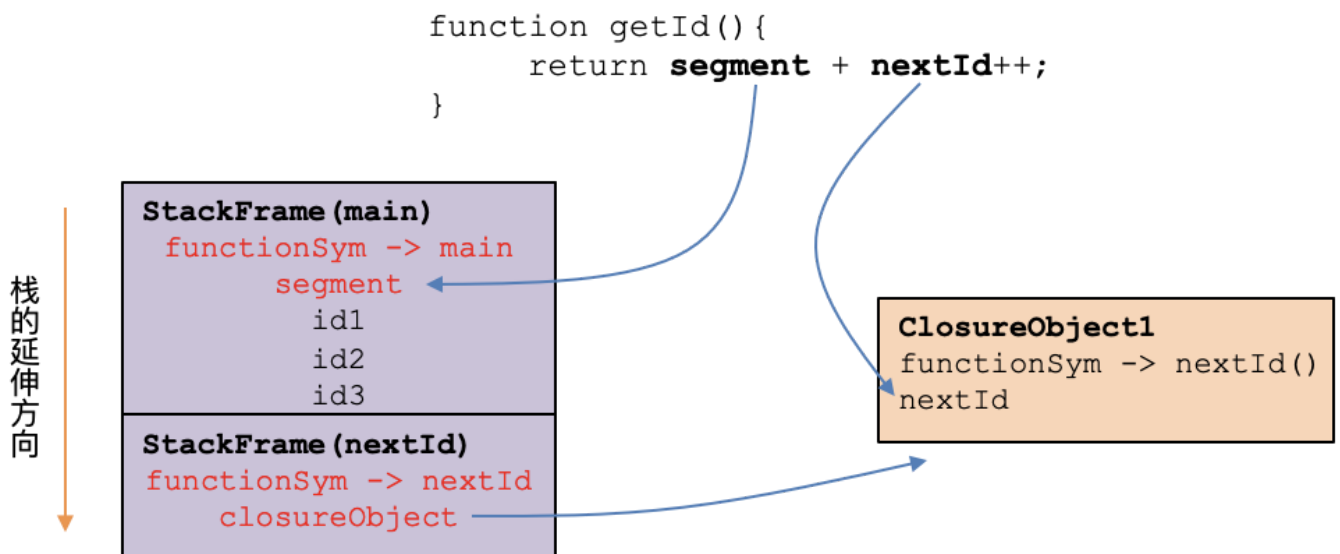
你能看出来，使用 ClosureObject 跟使用 PlayObject 是很相似的。对于函数中的数据，我们要区分出来哪些是位于函数作用域中的，哪些是位于 ClosureObject 中的。而在运行



对象方法时，我们也是要区分方法的本地变量和存放在 PlayObject 中的对象属性。

好了，现在我们设计完了闭包的数据对象。接下来我们看看如何让函数访问其他函数的栈帧中的变量，比如访问 `segment` 变量。

这个时候，函数的调用栈如下图所示。`segment` 位于全局函数的栈帧中。而 `id1()` 所调用的函数逻辑，需要先找到 `segment` 所在的函数的栈帧，然后才能访问 `segment` 变量的值。可是如何能找到 `segment` 变量所在的栈帧呢？



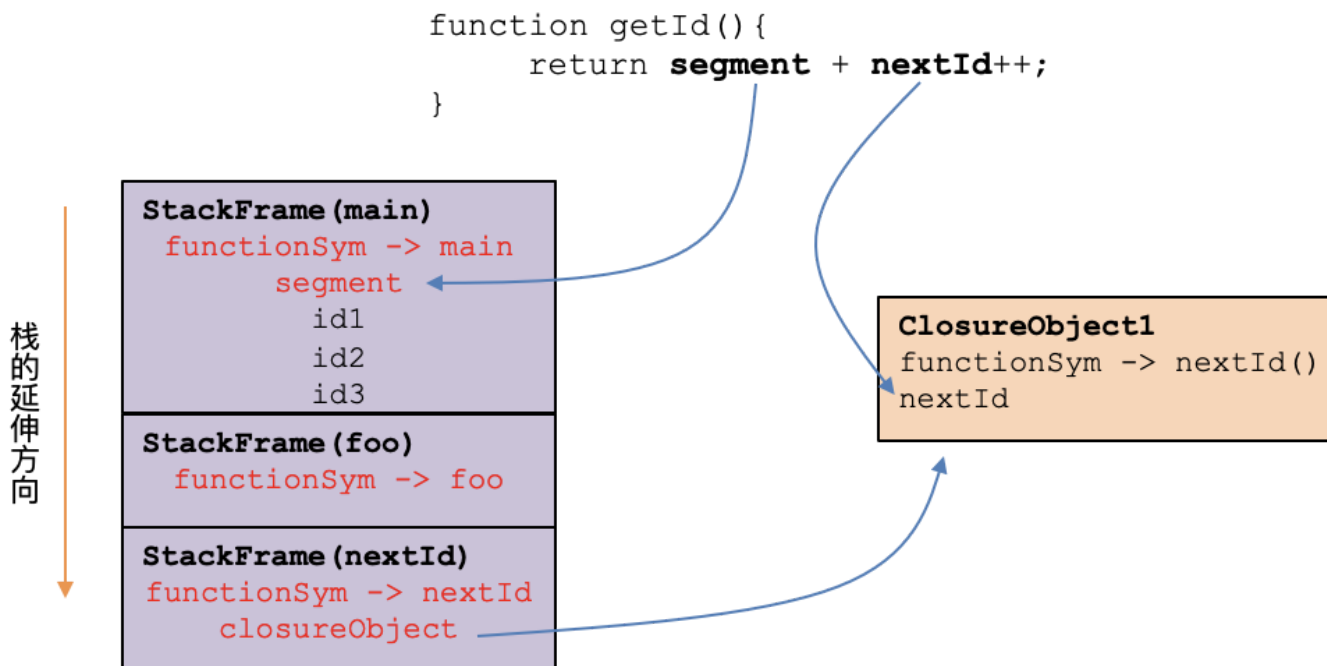
你可能会说，这还不简单，不就是当前栈帧的上一级栈帧吗？图中就是这么显示的呀。

不，事情没那么简单。你看，我在原来的示例函数的基础上再增加了一点代码。我们增加了一个函数 `foo()`，在 `foo` 里调用 `idGenerator()` 来生成闭包。

复制代码

```
1 function foo():()=>number{  
2     println("\nid4:");  
3     let id4 = idGenerator();  
4     println(id4());  
5     println(id4());  
6     return id4;  
7 }  
8  
9 let id5 = foo();  
10 println("\nid5:");  
11 println(id5());  
12 println(id5());
```

这个时候，调用栈就变成了下面这样，一共三级。你还可以再设计出更复杂的调用场景，产生更多级的栈帧。所以，`id1()` 的上一级栈帧并不是 `segment` 变量所在的栈帧。



那如何找到 `segment` 变量所在的栈帧呢？这就需要我们的栈帧能够跟函数关联起来，让算法知道哪个栈帧是由哪个函数产生的，然后就可以找到正确的栈帧了。

[复制代码](#)

```
1 class StackFrame{  
2     //存储变量的值  
3     values:Map<Symbol, any> = new Map();  
4  
5     //返回值，当调用函数的时候，返回值放在这里  
6     retVal:any = undefined;  
7  
8     //产生当前栈帧的函数  
9     functionSym:FunctionSymbol;  
10  
11     constructor(functionSym:FunctionSymbol){  
12         this.functionSym = functionSym;  
13     }  
14 }
```

好了，现在我们已经知道如何在 AST 解释器中支持闭包特性了。你可以查看 [play.ts](#) 中参考实现，研究一下其中的技术细节。

接下来我们再来分析一下，如何把闭包特性编译进可执行程序。

## 编译成可执行程序

**我们先来实现一下闭包的私有数据区。**

在这个技术特性上，我们完全可以借鉴面向对象特性的实现技术。我们可以把闭包函数，比作一个类的方法。至于闭包数据，我们就可以把它当做是对象的属性。

而当程序调用一个闭包函数的时候，它必须把闭包数据通过第一个参数传递进去，这个过程跟调用对象方法也是一样的。当调用对象方法的时候，对象引用也是第一个参数。

所以说，在有了实现面向对象特性的底子以后，我们再实现对闭包数据的管理，就会容易很多，可以充分借鉴原来的技术思路。

**不过，要实现另一个特性，也就是从别的函数的栈帧里找到所引用的外部变量的数据，就没那么简单了。**

按照 AST 解释器的实现逻辑，我们需要知道哪个栈帧是由哪个函数生成的。可是，我们马上就遇到了两个需要解决的技术点。

**第一个技术点，是我们并没有在可执行文件里保存函数符号的信息。**在编译后的可执行文件里，每个函数都只是一些机器码而已。在汇编代码中用来标识每个函数入口的标签，也消失不见了，因为它的作用已经完成了。它的作用，就是用于计算函数的入口地址而已。我们甚至可以说，在我们现在生成的机器码里，根本就没有任何关于函数的概念。

这个问题是可以解决的。我们可以在可执行程序的数据区保存程序的符号信息。在研究 C++ 程序生成的汇编代码中，我们已经看到，C++ 会在可执行文件里保存一些与类型有关的信息以及 vtable。我们可以采用相同的技术手段，保存我们自己想保存的信息。

实际上，如果我们要调试程序，那就要往可执行文件里保存很多符号信息和调试信息，否则没办法知道栈帧和寄存器里的哪个数据对应的是哪个变量。所以，当我们用 debug 模式生成可执行文件时，要比 release 模式的可执行文件大很多。

**第二个技术点，是我们并没有在原来的栈帧里保存对函数的引用信息。**所以，我们就难以搞清楚哪个栈帧是哪个函数生成的，也很难从中找到相应的变量。这个技术点也是可以解决的，这需要我们往栈帧里添加额外的信息，来建立栈帧和函数之间的对应关系。

不过，要实现这两个技术点，涉及的代码工作量有点大，我们在这节课的参考实现里就不包含这个特性了，你可以先自己动手试试。不用担心，这点我会在课程完结前补上。同时，在我们后面再迭代的、作为开源项目的 PlayScript 代码库中，也会添加这个特性，届时你再可以对照着看看。

## 课程小结

今天的内容就是这些。为了掌握闭包的特性，你需要记住以下几个知识点：

首先，闭包的产生，是由于声明时所引用的外部作用域中的变量，在运行时的作用域中并不存在，所以我们需要一个专门的数据区来保存这些数据。

第二，TypeScript 采用的是词法作用域，也就是在语义分析阶段就把变量的使用和声明做了绑定，并且不再改变。这是需要闭包机制的根本原因。

第三，闭包特性跟面向对象特性有很多相似之处，闭包数据就类似于对象数据。我们调用闭包函数的时候，也要把闭包对象的引用传递给函数，类似于调用对象的方法。

第四，如果要访问其他函数的栈帧内的数据，我们需要记录每个栈帧是由哪个函数生成的。

## 思考题

今天的思考题，我想让你分享一下你使用闭包特性的场景，以及为什么要使用闭包特性。欢迎你在留言区留言。

欢迎你把这节课分享给更多感兴趣的朋友。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    32 | 函数式编程第1关：实现高阶函数

### 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。