闭包与活动：解开寻址谜团

infinte

似乎某些程序员的集合是不相交的，就好像 JS 程序员和玩编译原理和 CPU 指令的汇编程序员就几乎没有交叉。前些日子讨论的火热的“作用域链”问题，说白了就是寻址问题，不过，这个在 C 中十分简单的问题却被 JS 这个动态语言弄得很复杂。

正是因为 JS 是动态语言，所以 JS 的寻址是现场寻址，而非像 C 一样，编译后确定。此外，JS 引入了 this 指针，这是一个很麻烦的东西，因为它“隐式”作为一个参数传到函数里面。我们先看“作用域链”话题中的例子：

```
var testvar = 'window 属性';
var o1 = {testvar:'1', fun:function(){alert('o1: '+this.testvar);}};
var o2 = {testvar:'2', fun:function(){alert('o2: '+this.testvar);}};
o1.fun(); // '1'
o2.fun(); // '2'
o1.fun.call(o2); //'2'
```

三次 alert 结果并不相同，很有趣不是么？其实，所有的有趣、诡异的概念最后都可以归结到一个问题上，那就是寻址。

JS 是静态还是动态作用域?

告诉你一个很不幸的消息，JS 是静态作用域的，或者说，变量寻址比 perl 之类的动态作用域语言要复杂得多。下面的代码是程序设计语言原理上面的例子：

```
function big(){
   var x = 1;
   eval('f1 = function(){echo(x)}');
   function f2(){var x = 2;f1()};
   f2();
};
big();
```

输出的是 1，和 pascal、ada 如出一辙，虽然 f1 是用 eval 动态定义的。另外一个例子同样来自《程序设计语言原理》：

```
function big2(){
```

```
    var x = 1;
    function f2(){echo(x)}; //用 x 的值产生一个输出
    function f3(){var x = 3;f4(f2)};
    function f4(f){var x = 4;f()};
    f3();
}
big2();//输出 1：深绑定；输出 4：浅绑定；输出 3：特别绑定
```

输出的还是 1，说明 JS 不仅是静态作用域，还是深绑定，这下事情出大了……

闭包===函数?

Wikipedia 对 Closure 的定义是“In computer science, a **closure** is a first-class function with free variables. Such a function is said to be "closed over" its free variables.”，翻译成中文就是“一个闭包是一个带有自己变量的（第一型）函数”。“第一”类型在 Aimingoo 的书中已经有论述，不过我还是要提一下：“第一型函数”说的是函数可以当做变量传递、作函数的返回值等。
函数被存储时，需要在计算机内存中包含一些东西——如函数的代码（字符串或 `OpCodes`）、参数表等。而函数**调用时**（注意，是调用时）需要的东西则包括变量表、实参表（JavaScript 中为 `arguments`）和被调用者（`arguments.callee`）。我们定义，如果某个函数被调用，则称这个函数发起了一次活动。而活动，就是“闭包”的重要成分。

活动的概念

为了解释函数（尤其是允许函数嵌套的语言中，比如 Ada）运行时复杂的寻址问题，《程序设计语言原理》一书中定义了活动记录实例（ARI，Activation record instance，以下简称活动）：它是堆栈上一些记录，包括：

1. 函数实例地址（`arguments.callee`）
2. 局部变量表
3. 实参表（arguments 表）
4. this 指针（也可看做实参的一个）
5. 返回值地址
6. 动态链接

这里，动态链接永远指向某个函数调用时所处的活动（如 b 活动时调用 a，则 a 的那次活动中，动态链接指向调用它的 b 的活动）。

静态链接

静态链接则描述了函数**定义时，引擎执行的活动**，因为函数的组织是有根树，所以所有的静态链接汇总后一定会指向宿主（如 window），我们可以看例子（注释后为输出）：

```
var x = 'x in host';
function a(){echo(x)};
function b(){var x = 'x inside b';echo(x)};
function c(){var x = 'x inside c';a()};
function d(){
    var x = 'x inside d,a closure-made function';
    return function(){echo(x)}
};

a();// x in host
b();// x inside b
c();// x in host
d()();//  x inside d,a closure-made function
```

在第一句调用时，我们可以视作“堆栈”上有下面的内容（左边为栈顶）：

```
[a 的活动] → [宿主]
  ↑当前活动
```

a 的静态链直直的戳向宿主，因为 a 中没有定义 x，解释器寻找 x 的时候，就沿着静态链在宿主中找到了 x；对 b 的调用，因为 b 的局部变量里记录了 x，所以最后 echo 的是 b 里面的 x：'x inside b'；

现在，c 的状况有趣多了，调用 c 时，可以这样写出堆栈信息：

```
动态链：[a]→[c]→[宿主]
静态链：[c]→[宿主]；[a]→[宿主]
```

因为对 x 的寻址在调用 a 后才进行，所以，静态链接还是直直的戳向宿主，自然 x 还是'x in host'咯！

d 的状况就更加有趣了，d 创建了一个函数作为返回值，而它紧接着就被调用了~因为 d 的返回值是在 d 的生命周期内创建的，所以 d 返回值的静态链接戳向 d，所以调用的时候，输出 d 中的 x：'x inside d,a closure-made function'。

静态链接的创建时机

前面说过，“闭包”是“带有局部变量”的函数，而“局部变量”就是静态链接之时的活动。不过和 Ada 等语言有些不同的是，《程序设计语言原理》里面的 ARI 保存在堆栈中，而且函数的生命周期一旦结束，ARI 就跟着销毁；而 JS 的活动却不是这样，活动被销毁，当且仅当没有指向它和它的成员的引用（或者说，任何代码都无法找到它）。我们可以简单地认为函数活动就是躲在函数背后一个对象，披上了“局部变量”的“衣服”而已。

《程序设计语言原理》描述的静态链是调用时创建的，不过，静态链的关系却是在代码编译的时候就确定了。比如，下面的代码：

```
PROCEDURE a;
  PROCEDURE b;
  END;
  PEOCEDURE c;
  END;
END;
```

b 和 c 的静态链戳向 a。如果调用 b，而 b 中某个变量又不在 b 的局部变量中时，编译器就生成一段代码，它希望沿着静态链向上搜堆栈，直到搜到变量或者 RTE。

和 Ada 之类的编译型语言不同的是，JS 是全解释性语言，而且函数可以动态创建，这就出现了“静态链维护”的难题。好在，JS 的函数不能直接修改，它就像 erlang 里面的符号一样，更改等于重定义。所以，静态链也就只需要在每次定义的时候更新一下。无论定义的方式是 function(){} 还是 eval 赋值，函数创建后，静态链就固定了。

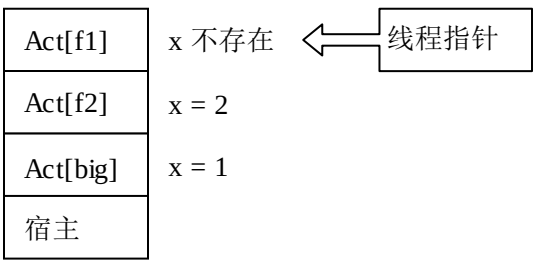
调用时到底发生了什么？

我们回到 big 的例子，当解释器运行到“function big(){.....}”时，它在内存中创建了一个函数实例，并连接静态链接到宿主。但是，在最后一行调用的时候，解释器在内存中画出一块区域，作为 big 的活动。我们不妨写作 Act[big]。执行指针移动到第 2 行。

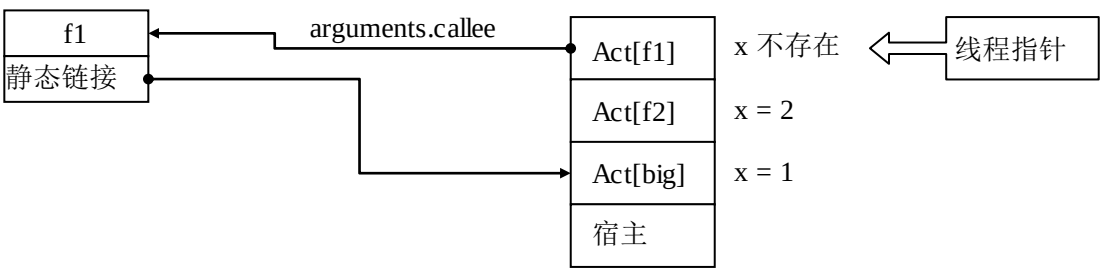
执行到第 3 行时，解释器创建了“f1”实例，保存在 Act[big]中，连接静态链到 Act [big]。下一行，解释器创建“f2”实例，连接静态链。接着，到了第 5 行，调用 f2，创建 Act [f2]；f2 调用 f1，创建 ARI[f1]；f1 要输出 x，就需要对 x 寻址。

变量的寻址

我们继续，现在要对 x 寻址，但 x 并不出现在 f1 的局部变量中，于是，解释器必须要沿着堆栈向上搜索去找 x，从输出看，解释器并不是沿着“堆栈”一层一层找，而是有跳跃的，因为此时“堆栈”为：



如果解释器真的沿着堆栈一层一层找的话，输出的就是 2 了。这就触及到 Js 变量寻址的本质：沿着静态链上搜。



继续上面的问题，执行指针沿着 f1 的静态链上搜，找到 big，恰好 big 里面有 x=1，于是输出 1，万事大吉。

那么，静态链是否会接成环，造成寻址“死循环”呢？大可不用担心，因为还记得函数是相互嵌套的么？换言之，函数组成的是有根树，所有的静态链指针最后一定能汇总到宿主，因此，担心“指针成环”是很荒谬的。（反而动态作用域语言寻址容易造成死循环。）

现在，我们可以总结一下简单变量寻址的方法：解释器现在当前函数的局部变量中寻找变量名，如果没有找到，就沿着静态链上溯，直到找到或者上溯到宿主仍然没有找到变量为止。

活动的生命

现在来正视一下活动，活动记录了函数执行时的局部变量（包括参数）、this 指针、动态链和最重要的——函数实例的地址。我们可以假想一下，ARI 有下面的结构：

```
Class Activity {
   List<Variable> variables,    //变量表
   List<Variable> arguments,    //参数表
   Variable pThis, //this 指针
   Activity dynamicLink, //动态链接
   Function arg_callee //函数实例
}
```

variables,arguments,pThis 包括所有局部变量、参数和 this 指针；dynamicLink 指向 ARI 被它的调用者；arg_callee 指向函数实例。在函数实例中，有：

```
Class Function {
   JSOptions operations, //函数指令
   Activity staticLink, //静态链接
   ......
}
```

当函数被调用时，实际上执行了如下的“形式代码”：

```
Acrivity p;
p = new Activity();
p.dynamicLink = JSEngine.thread.currentActivity;
p.arg_callee = 被调用的函数;
p.加入参数表和 this 引用;
JSEngine.thread.transfer(p.arg_callee.operations[0]);
```

看见了么？创建活动，压入参数和 this，之后转移线程指针到函数实例的第一个指令。

函数创建的时候呢？在函数指令赋值之后，还要：

```
newFunction.staticLink = JSEngine.thread.currentActivity;
```

现在问题清楚了，我们在函数定义时创建了静态链接，它直接戳向线程的当前 ARI。这样就可以解释几乎所有的简单变量寻址问题了。比如，下面的代码：

```
function test(){
  for(i=0;i<5;i++){
    (function(t){ //这个匿名函数姑且叫做 f
      setTimeout(function(){echo(''+t)},1000) //这里的匿名函数叫做 g
```

```
    })(i)
  }
}
test()
```

这段代码的效果是延迟 1 秒后按照 0 1 2 3 4 的顺序输出。我们着重看 `setTimeout` 作用的那个函数，在它创建时，静态链接指向匿名函数 f，f 的（某个活动的）变量表中含有 i（参数视作局部变量），所以，`setTimeout` 到时时，匿名函数 g 搜索变量 t，它在匿名函数 f 的 ARI 里面找到了。于是，按照创建时的顺序逐个输出 0 1 2 3 4。

公用匿名函数 f 的函数实例的活动一共有 5 个（还记得函数每调用一次，活动创建一次么？），相应的，g 也“创建”了 5 次。在第一个 `setTimeout` 到时之前，堆栈中相当于有下面的记录（我把 g 分开写成 5 个）：

```
+test 的活动  [循环结束时 i=5]
| f 的活动 ; t=0 ←————————g0 的静态链接
| f 的活动 ; t=1 ←————————g1 的静态链接
| f 的活动 ; t=2 ←————————g2 的静态链接
| f 的活动 ; t=3 ←————————g3 的静态链接
| f 的活动 ; t=4 ←————————g4 的静态链接
\------
```

而，g0 调用的时候，“堆栈”是下面的样子：

```
+test 的活动  [循环结束时 i=5]
| f 的活动 ; t=0 ←————————g0 的静态链接
| f 的活动 ; t=1 ←————————g1 的静态链接
| f 的活动 ; t=2 ←————————g2 的静态链接
| f 的活动 ; t=3 ←————————g3 的静态链接
| f 的活动 ; t=4 ←————————g4 的静态链接
\------


+g0 的活动
| 这里要对 t 寻址，于是……t=0
\------
```

g0 的活动可能并不在 f 系列的 ARI 中，可以视作直接放在宿主里面；但寻址所关心的静态链接却仍然戳向各个 f 的活动，自然不会出错咯~因为 `setTimeout` 是顺序压入等待队列的，所以最后按照 0 1 2 3 4 的顺序依次输出。

函数重定义时会修改静态链接吗?

现在看下一个问题：函数定义的时候会建立静态链接，那么，函数重定义的时候会建立另一个静态链接么？先看例子：

```
var x = "x in host";
f = function(){echo(x)};
f();
function big(){
    var x = 'x in big';
    f();
    f = function(){echo (x)};
    f()
}
big()
```

输出：

```
x in host
x in host
x in big
```

这个例子也许还比较好理解，big 运行的时候重定义了宿主中的 f，“新”f 的静态链接指向 big，所以最后一行输出'x in big'。

但是，下面的例子就有趣多了：

```
var x = "x in host";
f = function(){echo(x)};
f();
function big(){
    var x = 'x in big';
    f();
    var f1 = f;
    f1();
    f = f;
    f()
}
big()
```

输出：

```
x in host
x in host
x in host
x in host
```

不是说重定义就会修改静态链接么？但是，这里两个赋值只是赋值，只修改了 f1 和 f 的指针（还记得 JS 的函数是引用类型了么？），f 真正的实例中，静态链接没有改变！所以，四个输出实际上都是宿主中的 x。

函数修改自身会影响静态链接吗?

函数修改自身是一个有趣的话题。下面的代码也许有些不可理喻：

```
var x = 'host'
function f(){
  echo(x);
  modify_me();
  f()
}
function modify_me(){
  var x = 'in modify';
  f = function(){echo(x)}
}
f();
```

f 在调用的时候修改了自身，同时，两次 x 的结果也不同：

```
host
in modify
```

结果类似于上面的讨论，其实，“f”只是一个代号，modify_me 调用时修改了“代号”f 指向的内容，实际上创建了另一个函数实例。

Function 构造器和静态链接

Function 构造器也许是最破坏风景的，它接受字符串作为参数，但不会绑定静态链接到当前 ARI。