JavaScript 原理调研报告

李博杰 PB10000603

跟我同是 2010 级的一位同学在写一本 JavaScript 原理方面的书（膜拜），他把正在写的一部分内容放在了博客上（http://typeof.net/s/jsmech/）他还基于 JavaScript 开发了一种类似 CoffeeScript 的新语言——moescript，改进了很多语法糖，并将函数式特性发挥得淋漓尽致。

此调研报告就是跟着这些系列博客的路线，结合自己编写 JavaScript 的经历，参考 ECMAScript 5 规范（JavaScript 标准）写成的。

一、JavaScript 中的 this 指针

下面是一段真实系统中的 JavaScript 代码：

```
else if (action == "insert" || action == "replace") {
    var values_str = '(' +
        row.concat(teacher).map(db.escape).join(',') + ')';
```

运行这段代码时，会抛出奇怪的异常，通过调用栈可以发现，异常在 db.escape 内部。db 是一个数据库连接对象，它的 escape 方法是将数据转义，避开 SQL 语句中的特殊字符。这怎么会有问题呢？深入 nodejs 的数据库引擎源码才发现，escape 方法的调用链中需要获取当前时区 this.timezone（虽然这跟 escape 无关），而 escape 内部看到的 this 已经不是 db 对象了！

遇到这个问题时，我只是用简单方法解决了。系列博客的第一篇就解决了我的这个困惑。

在 ECMAScript 规范中，函数调用表达式有两条产生式：

CallExpression -> MemberExpression Arguments

CallExpression -> CallExpression Arguments

这是为了递归调用。调用的“中心语”是 MemberExpression。简要调用过程如下：

1. 计算 MemberExpression 得到 ref
2. 计算 Arguments
3. 如果 ref 不可调用，抛出异常
4. 如果 ref 是对象的方法，则令 this 为这个对象
5. 如果 ref 不是对象的方法，则令 this 为调用者环境中的 this
6. 用 this 和 arguments 调用 ref

在上面的问题代码中，将 db.escape 传给了 map，在 map 的内部实现中调用它的时候，JavaScript 引擎已经不知道它来自 db 对象，也就是上述第 4 条不满足，进入第 5 条，escape 中看到的 this 变成了调用者 map 看到的 this，这显然不是 db。

在正常的 db.escape 调用中，满足上述第 4 条判断，因此被调用者看到的 this 就是正确的 db 对象了。

二、eval 对象为什么要加括号

不借助类库写过 ajax（浏览器异步通信）的 JavaScript 程序员都知道，要想把返回的 json（JavaScript Object Notation）字符串变成一个对象以便使用，必须用

```
obj = eval('(' + str + ')')
```

为什么要加那一对括号呢？Google 一下就知道，JavaScript 规定表达式语句不能用花括号开头，而 json 对象是用花括号括起来的。系列博客的第二篇解释了 JavaScript 这样设计的原因。

看这行代码：

```
{a: 1}
```

它有两种解释：

1. 一个 JavaScript 对象，有一个名为 a、值为 1 的元素
2. 一个 JavaScript 语句块，a 是标号，语句体是 1。由于 JavaScript 在特定情况下允许省略分号，上述语句块是合法的。

这样就带来了语法上的歧义，必须解决。

ECMAScript 规范的办法很简单，就是规定该出现语句块的地方不能出现 JavaScript 对象字面量，也就是语句块不能单由一个对象字面量构成。为了让对象字面量能在 eval 中作为代码执行，只好把它当作表达式，包在圆括号中间了。

这篇博文还分析了 JavaScript 允许这么奇怪的语句块（上述第二种解释）存在的原因。因为 JavaScript 是 90 年代发明的，当时的主流语言都支持悬空语句块，也就是没有任何 for，if 之类前缀的语句块，或者说对应这样的产生式：

```
Block -> { StatementList }
```

事实上，C 语言中这样的语句块能产生新的作用域，还算有点用；但 JavaScript 的语句块不改变作用域，因此这样的悬空语句块除了装饰以外没什么用。可惜，JavaScript 不能免俗，没有割掉“悬空语句块”这块赘肉。

三、语法糖与分号癌

JavaScript 中的自动插入分号是一个“可怕”的特性。在前端 JavaScript 代码中，我曾经遇到了类似这样的奇怪问题：

```
var x = {
  'i': 1,
  'j': 2
}  // No semicolon here.
[normalVersion, ffVersion][isIE]();
```

JavaScript 不会自动补全方法声明后面的分号，而是将对象字面值和方法调用当成了同一条语句。

第三篇文章指出，JavaScript 自动补全分号事实上是在解释换行——一种解释是分号，另一种解释是空格。如果没有歧义，也就是另一种解释会导致不正确的 JavaScript 语法，就会作出唯一的解释；如果两种解释都是合法的 JavaScript，则会解释成空格。这就是前面悲剧代码的起因了。

为什么默认解释为空格呢？因为 JavaScript 那个时代的语言继承了 C 的遗风，允许这样的代码：

```
var a = f
(b+c)
```

它的意思是 var a = f(b+c)。

今天流行的 Python、Ruby 之类编程语言“根据缩进决定程序结构”的风格，还远远没有流行开来。我认为，可能是由于编辑器的变化。在 90 年代，图形界面和网络尚未

普及，程序要印到纸上才能传播，屏幕的字符宽度又只有 80 个，程序经常需要换行，因此程序语言的设计者将换行看成是跟空格一样。但今天我们的程序都通过网络传播，图形界面也赋予了一行显示更多字符的能力，用换行替代空格并不多见。

这篇文章详细说明了，从上述“消除歧义”的解释，是如何一步步导出 ECMAScript 规范中的三条拗口的分号插入规定的。接着提出了一种消除换行歧义的 JavaScript 文法。当然，这样的文法会变得面目全非，因此面对换行歧义，ECMAScript 规范的处理方法跟经典的 if-else 歧义一样，都是用“例外规则”来定义，而非修改文法使其没有歧义。

四、JavaScript 正规式与词法分析中的特殊模式

这个系列目前为止的最后一篇文章提到了 JavaScript 中的转义字符和正则表达式。JavaScript 中的正则表达式是用一种特殊的语法来声明的，这使得语法分析变得复杂。比如

```
a/b/g
```

词法层面上有两种解释：

1. a 除以 b，再除以 g
2. 一个标识符 a，模式 b，模式限定符 g（表示 global）

由于 JavaScript 文法中标识符后不能接模式，上述第 2 种解释是不成立的。不过上述判断只能在语法层面得出，这给基于词法分析的语法高亮器带来了麻烦。

我想到上学期编译原理实验中解析 PHP 的 lex 文法，使用了状态机的办法。PHP 中有类似 bash 和 Perl 的比较奇葩的语法：

```
"$var $arr[0] $arr[$index] ${varname}"
```

上述变量和数组元素的值会被替换进字符串。字符串不再是一个词法元素（终结符），而是一个非终结符，包含了更复杂的语法结构。问题来了：“function”出现在字符串里就是字面值，出现在外面就是关键字，基于正则表达式的词法分析很难分辨出两者的区别。

PHP 引擎对这种情况的处理，就是 lex 在主模式中遇到双引号（字符串的起始标志）时，就进入一种特殊模式（引用模式），这个模式中所用的正则式与外面不同，从而能够正确得到字符串内的词法元素，又不让这些特殊正则式“污染”外面的词法分析。当 lex 在引用模式内匹配上配对的双引号时，再返回主模式。

转义字符也可以用类似的方法处理。我不知道 JavaScript 引擎是如何匹配正规式的，但我认为可以用类似的机制实现。正规式可能比前面提到的 PHP 例子更复杂，因为需要语法分析来消除歧义，也就是需要 yacc 给 lex 提供“下一终结符是否可能是正规式”的信息。lex 知道下一终结符可能是正规式，并匹配上开始标志“/”时，就可以进入状态机，利用括号配对原则，一个一个字符地吃进去，直到遇到配对的结束标志“/”，此时 lex 就能把匹配到的部分作为正规式终结符返回给 yacc。

五、动态作用域与静态作用域

在函数式语言中，闭包是进化的产物，而不是“与生俱来”的。这是从王垠关于 Lisp 的文章（http://www.yinwang.org/blog-cn/2013/03/26/lisp-dead-alive/）里看到的。

在折腾 Emacs 的时候发现它的 Elisp 非常难以理解，其主要原因就是 Lisp 初始版本采用的动态作用域。也就是如果一个函数中存在自由变量，这个自由变量是到它的调用者那里去找。当然 Emacs 最新版本已经改掉了 Elisp 的动态作用域。

Lisp 最初为什么要采用这样“不可理喻”的动态作用域呢？因为实现简单。Lisp 把函数保存成一个 S 表达式，调用的时候要传函数，就把这个 S 表达式传进去了。函数调用是基于堆栈的，S 表达式求值的过程中遇到自由变量，就顺着堆栈往上找，找到哪个就用哪个。这种语义的问题是，自由变量的值会随着调用栈上相隔好几层的毫无关系的变量值改变，这不利于程序的封装。王垠说，Lisp Machine 失败的原因就在此。

因此，现在的函数式语言，如 Haskell、JavaScript，都是采用静态作用域。也就是函数中的自由变量到它的定义处去找。这就需要为每个函数创造一个“闭包”，也就是定义此函数的执行环境。由于函数定义处的环境以后还可能使用，这个环境在函数执行结束后并不能马上释放。因此执行环境不能放在栈里，而要放在堆里。这不仅增加了实现的复杂度，还给垃圾回收增加了麻烦。编译器必须有办法知道闭包的所有引用是否已经全部失效，否则就无法释放闭包占用的内存。

六、与本文主题无关的补充

顺便提一句，前面提到的博客及其友情链接中还有不少关于程序设计语言原理方面的文章。

例如《单子一瞥》（http://typeof.net/m/a-glance-at-monad.html）介绍了接连的异步操作产生的 Monad“楼梯”和 moescript 中的新语法结构，可以通过简单的语法构建 Monad 链。

将异步嵌套结构“平面化”不只是他一个人在做。在 nodejs 中，由于数据库、文件操作都是异步的，也会形成难写、难调、难改的嵌套 contiunation 链。清华大学的 BYVoid 编写的 continuation.js 就是这样的工具，用顺序方式书写异步代码，再由编译器将其翻译成标准 JavaScript 代码。