深入理解 JavaScript 系列（2）：揭秘命名函数表达式

2011-12-29 09:02 by 汤姆大叔, 54890 阅读, 64 评论, 收藏, 编辑

前言

网上还没用发现有人对命名函数表达式进去重复深入的讨论，正因为如此，网上出现了各种各样的误解，本文将从原理和实践两个方面来探讨 JavaScript 关于命名函数表达式的优缺点。

简单的说，命名函数表达式只有一个用户，那就是在 Debug 或者 Profiler 分析的时候来描述函数的名称，也可以使用函数名实现递归，但很快你就会发现其实是不切实际的。当然，如果你不关注调试，那就没什么可担心的了，否则，如果你想了解兼容性方面的东西的话，你还是应该继续往下看看。

我们先开始看看，什么叫函数表达式，然后再说一下现代调试器如何处理这些表达式，如果你已经对这方面很熟悉的话，请直接跳过此小节。

函数表达式和函数声明

在 ECMAScript 中，创建函数的最常用的两个方法是函数表达式和函数声明，两者期间的区别是有点晕，因为 ECMA 规范只明确了一点：函数声明必须带有标示符（Identifier）（就是大家常说的函数名称），而函数表达式则可以省略这个标示符：

函数声明:

function 函数名称 (参数：可选){ 函数体 }

函数表达式：

function 函数名称（可选）(参数：可选){ 函数体 }

所以，可以看出，如果不声明函数名称，它肯定是表达式，可如果声明了函数名称的话，如何判断是函数声明还是函数表达式呢？ECMAScript 是通过上下文来区分的，如果 function foo(){}是作为赋值表达式的一部分的话，那它就是一个函数表达式，如果 function foo(){}被包含在一个函数体内，或者位于程序的最顶部的话，那它就是一个函数声明。

```
function foo() {} // 声明，因为它是程序的一部分
var bar = function foo() {}; // 表达式，因为它是赋值表达式的一部分

new function bar() {}; // 表达式，因为它是 new 表达式

(function() {
  function bar() {} // 声明，因为它是函数体的一部分
})();
```

还有一种函数表达式不太常见，就是被括号括住的(function foo(){})，他是表达式的原因是因为括号 ()是一个分组操作符，它的内部只能包含表达式，我们来看几个例子：

```
function foo(){} // 函数声明
(function foo(){}); // 函数表达式：包含在分组操作符内

try {
  (var x = 5); // 分组操作符，只能包含表达式而不能包含语句：这里的var 就是语句
} catch(err) {
  // SyntaxError
}
```

你可以会想到，在使用 eval 对 JSON 进行执行的时候，JSON 字符串通常被包含在一个圆括号里：eval('(' + json + ')')，这样做的原因就是因为分组操作符，也就是这对括号，会让解析器强制将 JSON 的花括号解析成表达式而不是代码块。

```
try {
  { "x": 5 }; // "{" 和 "}" 做解析成代码块
} catch(err) {
  // SyntaxError
}

({ "x": 5 }); // 分组操作符强制将"{" 和 "}"作为对象字面量来解析
```

表达式和声明存在着十分微妙的差别，首先，函数声明会在任何表达式被解析和求值之前先被解析和求值，即使你的声明在代码的最后一行，它也会在同作用域内第一个表达式之前被解析/求值，参考如下例子，函数 fn 是在 alert 之后声明的，但是在 alert 执行的时候，fn 已经有定义了：

```
alert(fn());

function fn() {
  return 'Hello world!';
}
```

另外，还有一点需要提醒一下，函数声明在条件语句内虽然可以用，但是没有被标准化，也就是说不同的环境可能有不同的执行结果，所以这样情况下，最好使用函数表达式：

```
// 千万别这样做！
// 因为有的浏览器会返回 first 的这个 function，而有的浏览器返回的却是第二个

if (true) {
  function foo() {
    return 'first';
  }
```

```
}
else {
  function foo() {
    return 'second';
  }
}
foo();

// 相反，这样情况，我们要用函数表达式
var foo;
if (true) {
  foo = function() {
    return 'first';
  };
}
else {
  foo = function() {
    return 'second';
  };
}
foo();
```

函数声明的实际规则如下：

*函数声明*只能出现在*程序*或*函数体*内。从句法上讲，它们 不能出现在 Block（块）（{ ... }）中，例如不能出现在 if、while 或 for 语句中。因为 Block（块） 中只能包含 Statement 语句， 而不能包含*函数声明*这样的源元素。另一方面，仔细看一看规则也会发现，唯一可能让*表达式*出现在 Block（块）中情形，就是让它作为*表达式语句*的一部分。但是，规范明确规定了*表达式语句*不能以关键字 function 开头。而这实际上就是说，*函数表达式*同样也不能出现在 Statement 语句或 Block（块）中（因为 Block（块）就是由 Statement 语句构成的）。

函数语句

在 ECMAScript 的语法扩展中，有一个是函数语句，目前只有基于 Gecko 的浏览器实现了该扩展，所以对于下面的例子，我们仅是抱着学习的目的来看，一般来说不推荐使用（除非你针对 Gecko 浏览器进行开发）。

1.一般语句能用的地方，函数语句也能用，当然也包括 Block 块中：

```
if (true) {
  function f(){ }
}
else {
```

```
  function f(){ }
}
```

2.函数语句可以像其他语句一样被解析，包含基于条件执行的情形

```
if (true) {
  function foo(){ return 1; }
}
else {
  function foo(){ return 2; }
}
foo(); // 1
// 注：其它客户端会将 foo 解析成函数声明
// 因此，第二个 foo 会覆盖第一个，结果返回 2，而不是 1
```

3.函数语句不是在变量初始化期间声明的，而是在运行时声明的——与函数表达式一样。不过，函数语句的标识符一旦声明能在函数的整个作用域生效了。标识符有效性正是导致函数语句与函数表达式不同的关键所在（下一小节我们将会展示命名函数表达式的具体行为）。

```
// 此刻，foo 还没用声明
typeof foo; // "undefined"
if (true) {
  // 进入这里以后，foo 就被声明在整个作用域内了
  function foo(){ return 1; }
}
else {
  // 从来不会走到这里，所以这里的 foo 也不会被声明
  function foo(){ return 2; }
}
typeof foo; // "function"
```

不过，我们可以使用下面这样的符合标准的代码来模式上面例子中的函数语句：

```
var foo;
if (true) {
  foo = function foo(){ return 1; };
}
else {
  foo = function foo() { return 2; };
}
```

4.函数语句和函数声明（或命名函数表达式）的字符串表示类似，也包括标识符：

```
if (true) {
  function foo(){ return 1; }
```

```
}
String(foo); // function foo() { return 1; }
```

5.另外一个，早期基于 Gecko 的实现（Firefox 3 及以前版本）中存在一个 bug，即函数语句覆盖函数声明的方式不正确。在这些早期的实现中，函数语句不知何故不能覆盖函数声明：

```
// 函数声明
function foo(){ return 1; }
if (true) {
  // 用函数语句重写
  function foo(){ return 2; }
}
foo(); // FF3 以下返回 1，FF3.5 以上返回 2

// 不过，如果前面是函数表达式，则没用问题
var foo = function(){ return 1; };
if (true) {
  function foo(){ return 2; }
}
foo(); // 所有版本都返回 2
```

再次强调一点，上面这些例子只是在某些浏览器支持，所以推荐大家不要使用这些，除非你就在特性的浏览器上做开发。

命名函数表达式

函数表达式在实际应用中还是很常见的，在 web 开发中友个常用的模式是基于对某种特性的测试来伪装函数定义，从而达到性能优化的目的，但由于这种方式都是在同一作用域内，所以基本上一定要用函数表达式：

```
// 该代码来自 Garrett Smith 的 APE Javascript library 库(http://dhtmlkitchen.com/ape/)
var contains = (function() {
  var docEl = document.documentElement;

  if (typeof docEl.compareDocumentPosition != 'undefined') {
    return function(el, b) {
      return (el.compareDocumentPosition(b) & 16) !== 0;
    };
  }
  else if (typeof docEl.contains != 'undefined') {
    return function(el, b) {
      return el !== b && el.contains(b);
    };
  }
```

```
    return function(el, b) {
      if (el === b) return false;
      while (el != b && (b = b.parentNode) != null);
      return el === b;
    };
  })();
```

提到命名函数表达式，理所当然，就是它得有名字，前面的例子 var bar = function foo (){};就是一个有效的命名函数表达式，但有一点需要记住：这个名字只在新定义的函数作用域内有效，因为规范规定了标示符不能在外围的作用域内有效：

```
  var f = function foo(){
    return typeof foo; // foo 是在内部作用域内有效
  };
  // foo 在外部用于是不可见的
  typeof foo; // "undefined"
  f(); // "function"
```

既然，这么要求，那命名函数表达式到底有啥用啊？为啥要取名？

正如我们开头所说：给它一个名字就是可以让调试过程更方便，因为在调试的时候，如果在调用栈中的每个项都有自己的名字来描述，那么调试过程就太爽了，感受不一样嘛。

调试器中的函数名

如果一个函数有名字，那调试器在调试的时候会将它的名字显示在调用的栈上。有些调试器（Firebug）有时候还会为你们函数取名并显示，让他们和那些应用该函数的便利具有相同的角色，可是通常情况下，这些调试器只安装简单的规则来取名，所以说没有太大价格，我们来看一个例子：

```
  function foo(){
    return bar();
  }
  function bar(){
    return baz();
  }
  function baz(){
    debugger;
  }
  foo();

  // 这里我们使用了 3 个带名字的函数声明
  // 所以当调试器走到 debugger 语句的时候，Firebug 的调用栈上看起来非常清晰明了
  // 因为很明白地显示了名称
```

```
baz
bar
foo
expr_test.html()
```

通过查看调用栈的信息，我们可以很明了地知道 foo 调用了 bar，bar 又调用了 baz（而 foo 本身有在 expr_test.html 文档的全局作用域内被调用），不过，还有一个比较爽地方，就是刚才说的 Firebug 为匿名表达式取名的功能：

```
function foo(){
  return bar();
}
var bar = function(){
  return baz();
}
function baz(){
  debugger;
}
foo();

// Call stack
baz
bar() //看到了么？
foo
expr_test.html()
```

然后，当函数表达式稍微复杂一些的时候，调试器就不那么聪明了，我们只能在调用栈中看到问号：

```
function foo(){
  return bar();
}
var bar = (function(){
  if (window.addEventListener) {
    return function(){
      return baz();
    };
  }
  else if (window.attachEvent) {
    return function() {
      return baz();
    };
  }
})();
function baz(){
  debugger;
```

```
}
foo();

// Call stack
baz
(?)() // 这里可是问号哦
foo
expr_test.html()
```

另外，当把函数赋值给多个变量的时候，也会出现令人郁闷的问题：

```
function foo(){
  return baz();
}
var bar = function(){
  debugger;
};
var baz = bar;
bar = function() {
  alert('spoofed');
};
foo();

// Call stack:
bar()
foo
expr_test.html()
```

这时候，调用栈显示的是 foo 调用了 bar，但实际上并非如此，之所以有这种问题，是因为 baz 和另外一个包含 alert('spoofed')的函数做了引用交换所导致的。

归根结底，只有给函数表达式取个名字，才是最委托的办法，也就是使用**命名函数表达式**。我们来使用带名字的表达式来重写上面的例子（注意立即调用的表达式块里返回的 2 个函数的名字都是 bar）：

```
function foo(){
  return bar();
}
var bar = (function(){
  if (window.addEventListener) {
    return function bar(){
      return baz();
    };
  }
  else if (window.attachEvent) {
    return function bar() {
      return baz();
```

```
    };
  }
})();
function baz(){
  debugger;
}
foo();

// 又再次看到了清晰的调用栈信息了耶!
baz
bar
foo
expr_test.html()
```

OK，又学了一招吧？不过在高兴之前，我们再看看不同寻常的 JScript 吧。

JScript 的 Bug

比较恶的是，IE 的 ECMAScript 实现 JScript 严重混淆了命名函数表达式，搞得现很多人都出来反对命名函数表达式，而且即便是最新的一版（IE8 中使用的 5.8 版）仍然存在下列问题。

下面我们就来看看 IE 在实现中究竟犯了那些错误，俗话说知己知彼，才能百战不殆。我们来看看如下几个例子：

例 1：函数表达式的标示符泄露到外部作用域

```
var f = function g(){};
typeof g; // "function"
```

上面我们说过，命名函数表达式的标示符在外部作用域是无效的，但 JScript 明显是违反了这一规范，上面例子中的标示符 g 被解析成函数对象，这就乱了套了，很多难以发现的 bug 都是因为这个原因导致的。

注：IE9 貌似已经修复了这个问题

例 2：将命名函数表达式同时当作函数声明和函数表达式

```
typeof g; // "function"
var f = function g(){};
```

特性环境下，函数声明会优先于任何表达式被解析，上面的例子展示的是 JScript 实际上是把命名函数表达式当成函数声明了，因为它在实际声明之前就解析了 g。

这个例子引出了下一个例子。

例 3：命名函数表达式会创建两个截然不同的函数对象！

```
var f = function g(){};
f === g; // false

f.expando = 'foo';
g.expando; // undefined
```

看到这里，大家会觉得问题严重了，因为修改任何一个对象，另外一个没有什么改变，这太恶了。通过这个例子可以发现，创建 2 个不同的对象，也就是说如果你想修改 f 的属性中保存某个信息，然后想当然地通过引用相同对象的 g 的同名属性来使用，那问题就大了，因为根本就不可能。

再来看一个稍微复杂的例子：

例 4：仅仅顺序解析函数声明而忽略条件语句块

```
var f = function g() {
  return 1;
};
if (false) {
  f = function g(){
    return 2;
  };
}
g(); // 2
```

这个 bug 查找就难多了，但导致 bug 的原因却非常简单。首先，g 被当作函数声明解析，由于 JScript 中的函数声明不受条件代码块约束，所以在这个很恶的 if 分支中，g 被当作另一个函数 function g(){ return 2 }，也就是又被声明了一次。然后，所有“常规的”表达式被求值，而此时 f 被赋予了另一个新创建的对象的引用。由于在对表达式求值的时候，永远不会进入“这个可恶 if 分支，因此 f 就会继续引用第一个函数 function g(){ return 1 }。分析到这里，问题就很清楚了：假如你不够细心，在 f 中调用了 g，那么将会调用一个毫不相干的 g 函数对象。

你可能会文，将不同的对象和 arguments.callee 相比较时，有什么样的区别呢？我们来看看：

```
var f = function g(){
  return [
    arguments.callee == f,
    arguments.callee == g
  ];
};
f(); // [true, false]
g(); // [false, true]
```

可以看到，arguments.callee 的引用一直是被调用的函数，实际上这也是好事，稍后会解释。

还有一个有趣的例子，那就是在不包含声明的赋值语句中使用命名函数表达式：

```
(function(){
  f = function f(){};
})();
```

按照代码的分析，我们原本是想创建一个全局属性 f（注意不要和一般的匿名函数混淆了，里面用的是带名字的生命），JScript 在这里捣乱了一把，首先他把表达式当成函数声明解析了，所以左边的 f 被声明为局部变量了（和一般的匿名函数里的声明一样），然后在函数执行的时候，f 已经是定义过的了，右边的 function f(){}则直接就赋值给局部变量 f 了，所以 f 根本就不是全局属性。

了解了 JScript 这么变态以后，我们就要及时预防这些问题了，首先**防范标识符泄漏带外部作用域**，其次，应该永远**不引用被用作函数名称的标识符**；还记得前面例子中那个讨人厌的标识符 g 吗？——如果我们能够当 g 不存在，可以避免多少不必要的麻烦哪。因此，关键就在于始终要通过 f 或者 arguments.callee 来引用函数。如果你使用了命名函数表达式，那么应该只在调试的时候利用那个名字。最后，还要记住一点，一定要把**命名函数表达式声明期间错误创建的函数清理干净**。

对于，上面最后一点，我们还得再解释一下。

JScript 的内存管理

知道了这些不符合规范的代码解析 bug 以后，我们如果用它的话，就会发现内存方面其实是有问题的，来看一个例子：

```
var f = (function(){
  if (true) {
    return function g(){};
  }
  return function g(){};
})();
```

我们知道，这个匿名函数调用返回的函数（带有标识符 g 的函数），然后赋值给了外部的 f。我们也知道，命名函数表达式会导致产生多余的函数对象，而该对象与返回的函数对象不是一回事。所以这个多余的 g 函数就死在了返回函数的闭包中了，因此内存问题就出现了。这是因为 if 语句内部的函数与 g 是在同一个作用域中被声明的。这种情况下 ，除非我们显式断开对 g 函数的引用，否则它一直占着内存不放。

```
var f = (function(){
  var f, g;
  if (true) {
    f = function g(){};
  }
  else {
    f = function g(){};
```

```
    }
    // 设置 g 为 null 以后它就不会再占内存了
    g = null;
    return f;
  })();
```

通过设置 g 为 null，垃圾回收器就把 g 引用的那个隐式函数给回收掉了，为了验证我们的代码，我们来做一些测试，以确保我们的内存被回收了。

测试

测试很简单，就是命名函数表达式创建 10000 个函数，然后把它们保存在一个数组中。等一会儿以后再看这些函数到底占用了多少内存。然后，再断开这些引用并重复这一过程。下面是测试代码：

```
function createFn(){
  return (function(){
    var f;
    if (true) {
      f = function F(){
        return 'standard';
      };
    }
    else if (false) {
      f = function F(){
        return 'alternative';
      };
    }
    else {
      f = function F(){
        return 'fallback';
      };
    }
    // var F = null;
    return f;
  })();
}

var arr = [ ];
for (var i=0; i<10000; i++) {
  arr[i] = createFn();
}
```

通过运行在 Windows XP SP2 中的任务管理器可以看到如下结果：

```
IE6:
```

```
    without `null`:    7.6K -> 20.3K
    with `null`:       7.6K -> 18K

  IE7:

    without `null`:    14K -> 29.7K
    with `null`:       14K -> 27K
```

如我们所料，显示断开引用可以释放内存，但是释放的内存不是很多，10000 个函数对象才释放大约 3M 的内存，这对一些小型脚本不算什么，但对于大型程序，或者长时间运行在低内存的设备里的时候，这是非常有必要的。

关于在 Safari 2.x 中 JS 的解析也有一些 bug，但介于版本比较低，所以我们在这里就不介绍了，大家如果想看的话，请仔细查看英文资料。

SpiderMonkey 的怪癖

大家都知道，命名函数表达式的标识符只在函数的局部作用域中有效。但包含这个标识符的局部作用域又是什么样子的吗？其实非常简单。在命名函数表达式被求值时，会**创建一个特殊的对象**，该对象的唯一目的就是保存一个属性，而这个属性的名字对应着函数标识符，属性的值对应着那个函数。这个对象会被注入到当前作用域链的前端。然后，被“扩展”的作用域链又被用于初始化函数。

在这里，有一点十分有意思，那就是 ECMA-262 定义这个（保存函数标识符的）“特殊”对象的方式。标准说**“像调用 new Object()表达式那样”**创建这个对象。如果从字面上来理解这句话，那么这个对象就应该是全局 Object 的一个实例。然而，只有一个实现是按照标准字面上的要求这么做的，这个实现就是 SpiderMonkey。因此，在 SpiderMonkey 中，扩展 Object.prototype 有可能会干扰函数的局部作用域：

```
  Object.prototype.x = 'outer';

  (function(){

    var x = 'inner';

    /*
       函数 foo 的作用域链中有一个特殊的对象——用于保存函数的标识符。这个特殊的对象实际上就是{ foo: <function object> }。
       当通过作用域链解析 x 时，首先解析的是 foo 的局部环境。如果没有找到 x，则继续搜索作用域链中的下一个对象。下一个对象
       就是保存函数标识符的那个对象——{ foo: <function object> }，由于该对象继承自 Object.prototype，所以在此可以找到 x。
       而这个 x 的值也就是 Object.prototype.x 的值（outer）。结果，外部函数的作用域（包含 x = 'inner'的作用域）就不会被解析了。
```

```
    */

    (function foo(){

      alert(x); // 提示框中显示：outer

    })();
  })();
```

不过，更高版本的 SpiderMonkey 改变了上述行为，原因可能是认为那是一个安全漏洞。也就是说，“特殊”对象不再继承 Object.prototype 了。不过，如果你使用 Firefox 3 或者更低版本，还可以“重温”这种行为。

另一个把内部对象实现为全局 Object 对象的是黑莓（Blackberry）浏览器。目前，它的*活动对象*（Activation Object）仍然继承 Object.prototype。可是，ECMA-262 并没有说*活动对象*也要“像调用 new Object()表达式那样”来创建（或者说像创建保存 NFE 标识符的对象一样创建）。 人家规范只说了*活动对象*是规范中的一种机制。

那我们就来看看黑莓里都发生了什么：

```
  Object.prototype.x = 'outer';

  (function(){

    var x = 'inner';

    (function(){

      /*
      在沿着作用域链解析 x 的过程中，首先会搜索局部函数的活动对象。当然，在该对象中找不到 x。
      可是，由于活动对象继承自 Object.prototype，因此搜索 x 的下一个目标就是 Object.prototype；而
      Object.prototype 中又确实有 x 的定义。结果，x 的值就被解析为——outer。跟前面的例子差不多，
      包含 x = 'inner'的外部函数的作用域（活动对象）就不会被解析了。
      */

      alert(x); // 显示：outer

    })();
  })();
```

不过神奇的还是，函数中的变量甚至会与已有的 Object.prototype 的成员发生冲突，来看看下面的代码：

```
(function(){

  var constructor = function(){ return 1; };

  (function(){

    constructor(); // 求值结果是{}（即相当于调用了 Object.prototype.constructor()）而不是 1

    constructor === Object.prototype.constructor; // true
    toString === Object.prototype.toString; // true

    // ……

  })();
})();
```

要避免这个问题，要避免使用 Object.prototype 里的属性名称，如 toString, valueOf, hasOwnProperty 等等。

JScript 解决方案

```
var fn = (function(){

  // 声明要引用函数的变量
  var f;

  // 有条件地创建命名函数
  // 并将其引用赋值给 f
  if (true) {
    f = function F(){ }
  }
  else if (false) {
    f = function F(){ }
  }
  else {
    f = function F(){ }
  }

  // 声明一个与函数名（标识符）对应的变量，并赋值为 null
  // 这实际上是给相应标识符引用的函数对象作了一个标记，
  // 以便垃圾回收器知道可以回收它了
  var F = null;
```

```
    // 返回根据条件定义的函数
    return f;
})();
```

最后我们给出一个应用上述技术的应用实例，这是一个跨浏览器的 addEvent 函数代码：

```
// 1) 使用独立的作用域包含声明
var addEvent = (function(){

    var docEl = document.documentElement;

    // 2) 声明要引用函数的变量
    var fn;

    if (docEl.addEventListener) {

        // 3) 有意给函数一个描述性的标识符
        fn = function addEvent(element, eventName, callback) {
            element.addEventListener(eventName, callback, false);
        }
    }
    else if (docEl.attachEvent) {
        fn = function addEvent(element, eventName, callback) {
            element.attachEvent('on' + eventName, callback);
        }
    }
    else {
        fn = function addEvent(element, eventName, callback) {
            element['on' + eventName] = callback;
        }
    }

    // 4) 清除由 JScript 创建的 addEvent 函数
    //    一定要保证在赋值前使用 var 关键字
    //    除非函数顶部已经声明了 addEvent
    var addEvent = null;

    // 5) 最后返回由 fn 引用的函数
    return fn;
})();
```

替代方案

其实，如果我们不想要这个描述性名字的话，我们就可以用最简单的形式来做，也就是在函数内部声明一个函数（而不是函数表达式），然后返回该函数：

```
var hasClassName = (function(){

  // 定义私有变量
  var cache = { };

  // 使用函数声明
  function hasClassName(element, className) {
    var _className = '(?:^|\\s+)' + className + '(?:\\s+|$)';
    var re = cache[_className] || (cache[_className] = new RegExp(_
className));
    return re.test(element.className);
  }

  // 返回函数
  return hasClassName;
})();
```

显然，当存在多个分支函数定义时，这个方案就不行了。不过有种模式貌似可以实现：那就是提前使用函数声明来定义所有函数，并分别为这些函数指定不同的标识符：

```
var addEvent = (function(){

  var docEl = document.documentElement;

  function addEventListener(){
    /* ... */
  }
  function attachEvent(){
    /* ... */
  }
  function addEventAsProperty(){
    /* ... */
  }

  if (typeof docEl.addEventListener != 'undefined') {
    return addEventListener;
  }
  elseif (typeof docEl.attachEvent != 'undefined') {
    return attachEvent;
  }
  return addEventAsProperty;
})();
```

虽然这个方案很优雅，但也不是没有缺点。第一，由于使用不同的标识符，导致丧失了命名的一致性。且不说这样好还是坏，最起码它不够清晰。有人喜欢使用相同的名字，但也有人根本不在乎字眼上的差别。可毕竟，不同的名字会让人联想到所用的不同实现。例

如，在调试器中看到 attachEvent，我们就知 道 addEvent 是基于 attachEvent 的实现。当 然，基于实现来命名的方式也不一定都行得通。假如我们要提供一个 API，并按照这种方式把函数命名为 inner。那么 API 用户的很容易就会被相应实现的 细节搞得晕头转向。

要解决这个问题，当然就得想一套更合理的命名方案了。但关键是不要再额外制造麻烦。我现在能想起来的方案大概有如下几个：

```
  'addEvent', 'altAddEvent', 'fallbackAddEvent'
  // 或者
  'addEvent', 'addEvent2', 'addEvent3'
  // 或者
  'addEvent_addEventListener', 'addEvent_attachEvent', 'addEvent_asPr
operty'
```

另外，这种模式还存在一个小问题，即增加内存占用。提前创建 N 个不同名字的函数，等于有 N-1 的函数是用不到的。具体来讲，如果 document.documentElement 中包含 attachEvent，那么 addEventListener 和 addEventAsProperty 则根本就用不着了。可是，他们都占着内存哪；而且，这些内存将永远都得不到释放，原因跟 JScript 臭哄哄的命名表达式相同——这两个函数都被“截留”在返回的那个函数的闭包中了。

不过，增加内存占用这个问题确实没什么大不了的。如果某个库——例如 Prototype.js——采用了这种模式，无非也就是多创建一两百个函数而已。只要不是（在运行时）重复地创建这些函数，而是只（在加载时）创建一次，那么就没有什么好担心的。

WebKit 的 displayName

WebKit 团队在这个问题采取了有点儿另类的策略。介于匿名和命名函数如此之差的表现力，WebKit 引入了一个“特殊的”displayName 属性（本质上是一个字符串），如果开发人员为函数的这个属性赋值，则该属性的值将在调试器或性能分析器中被显示在函数“名称”的位置上。Francisco Tolmasky 详细地解释了这个策略的原理和实现。

未来考虑

将来的 ECMAScript-262 第 5 版（目前还是草案）会引入所谓的**严格模式（strict mode）**。开启严格模式的实现会禁用语言中的那些不稳定、不可靠和不安全的特性。据说出于安全方面的考虑，arguments.callee 属性将在严格模式下被“封杀”。因此，在处于严格模式时，访问 arguments.callee 会导致 TypeError（参见 ECMA-262 第 5 版的 10.6 节）。而我之所以在此提到严格模式，是因为如果在基于第 5 版标准的实现中无法使用 arguments.callee 来执行递归操作，那么使用命名函数表达式的可能性就会大大增加。从这个意义上来说，理解命名函数表达式的语义及其 bug 也就显得更加重要了。

```
// 此前，你可能会使用 arguments.callee
(function(x) {
  if (x <= 1) return 1;
  return x * arguments.callee(x - 1);
})(10);

// 但在严格模式下，有可能就要使用命名函数表达式
(function factorial(x) {
  if (x <= 1) return 1;
  return x * factorial(x - 1);
})(10);

// 要么就退一步，使用没有那么灵活的函数声明
function factorial(x) {
  if (x <= 1) return 1;
  return x * factorial(x - 1);
}
factorial(10);
```

致谢

理查德·康福德（Richard Cornford），是他率先解释了 JScript 中命名函数表达式所存在的 bug。理查德解释了我在这篇文章中提及的大多数 bug，所以我强烈建议大家去看看他的解释。我还要感谢 ***Yann-Erwan Perio*** 和***道格拉斯·克劳克佛德（Douglas Crockford）***，他们早在 2003 年就在 comp.lang.javascript 论坛中提及并讨论 NFE 问题了。

约翰-戴维·道尔顿（John-David Dalton）对“最终解决方案”提出了很好的建议。

托比·兰吉的点子被我用在了“替代方案”中。

盖瑞特·史密斯（Garrett Smith）和***德米特里·苏斯尼科（Dmitry Soshnikov）***对本文的多方面作出了补充和修正。

英文原文：http://kangax.github.com/nfe/

参考译文：连接访问 (SpiderMonkey 的怪癖之后的章节参考该文)