详解javascript立即执行函数表达式（IIFE）

2015/06/15 · JavaScript · IIFE, Javascript

5 原文出处： 韩子迟

写在前面

这是一篇译文，原文：Immediately-Invoked Function Expression (IIFE)

原文是一篇很经典的讲解IIFE的文章，很适合收藏。本文虽然是译文，但是直译的很少，而且添加了不少自己的理解。

ps：下文中提到的“立即执行函数”其实就是“立即执行函数表达式”

我们要说的到底是什么?

在javascript中，每一个函数在被调用的时候都会创建一个执行上下文，在该函数内部定义的变量和函数只能在该函数内部被使用，而正是因为这个上下文，使得我们在调用函数的时候能创建一些私有变量。

JavaScript

```
// makeCounter函数返回的是一个新的函数，该函数对makeCounter里的局部变量i享有使用权
function makeCounter() {
  // i只是makeCounter函数内的局部变量
  var i = 0;

  return function() {
    console.log( ++i );
  };
}

// 注意counter和counter2是不同的实例，它们分别拥有自己范围里的i变量

var counter = makeCounter();
```

```
var counter = makeCounter();
counter(); // 1
counter(); // 2

var counter2 = makeCounter();
counter2(); // 1
counter2(); // 2

i; // 报错，i没有定义，它只是makeCounter内部的局部变量
```

很多情况下我们并不需要像以上代码一样初始化很多实例，甚至有时候并不需要返回值。

- 问题的核心

现在我们定义了一个函数（function foo(){}或者var foo = function(){}），函数名后加上一对小括号即可完成对该函数的调用，比如下面的代码：

```JavaScript
var foo = function(){ /* code */ };
foo();
```

接着我们来看下面的代码：

```JavaScript
function(){ /* code */ }(); // SyntaxError: Unexpected token (
```

报错了，这是为何？这是因为在javascript代码解释时，当遇到function关键字时，会默认把它当做是一个函数声明，而不是函数表达式，如果没有把它显视地表达成函数表达式，就报错了，因为函数声明

错，因为函数声明是应该有个函数名的。）

- 一波未平一波又起

有意思的是，如果我们给它函数名，然后加上()立即调用，同样也会报错，而这次报错原因却不相同：

```JavaScript
function foo(){ /* code */ }(); // SyntaxError: Unexpected token )
```

为什么会这样？在一个表达式后面加上括号，表示该表达式立即执行；而如果是在一个语句后面加上括号，该括号完全和之前的语句不搭嘎，而只是一个分组操作符，用来控制运算中的优先级（小括号里的先运算）。所以以上代码等价于：

```JavaScript
function foo(){ /* code */ }
(); // SyntaxError: Unexpected token )
```

相当于先声明了一个叫foo的函数，之后进行()内的表达式运算，但是()（分组操作符）内的表达式不能为空，所以报错。（以上代码，也就是执行到右括号时，发现表达式为空，所以报错）。

如果想要了解更多，可以参考ECMA-262-3 in detail. Chapter 5. Functions.

立即执行函数（IIFE）

看到这里，相信你一定迫不及待地想知道究竟如何做了吧，其实很简单，只需要用括号全部括起来即可，比如下面这样：

```JavaScript
(function(){ /* code */ }());
```

为什么这样就能立即执行并且不报错呢？因为在javascript里，括号内部不能包含语句，当解析器对代码进行解释的时候，先碰到了()，然后碰到function关键字就会自动将()里面的代码识别为函数表达式而不是函数声明。

而立即执行函数并非只有上面的一种写法，写法真是五花八门：

```JavaScript
// 最常用的两种写法
(function(){ /* code */ }()); // 老道推荐写法
(function(){ /* code */ })(); // 当然这种也可以

// 括号和JS的一些操作符（如 = && || ,等）可以在函数表达式和函数声明上消除歧义
// 如下代码中，解析器已经知道一个是表达式了，于是也会把另一个默认为表达式
// 但是两者交换则会报错
var i = function(){ return 10; }();
true && function(){ /* code */ }();
0, function(){ /* code */ }();

// 如果你不怕代码晦涩难读，也可以选择一元运算符
!function(){ /* code */ }();
~function(){ /* code */ }();
-function(){ /* code */ }();
+function(){ /* code */ }();

// 你也可以这样
new function(){ /* code */ }
new function(){ /* code */ }() // 带参数
```

- 无论何时，给立即执行函数加上括号是个好习惯

通过以上的介绍，我们大概了解通过()可以使得一个函数表达式立即执行。

有的时候，我们实际上不需要使用()使之变成一个函数表达式，啥意思？比如下面这行代码，其实不加上()也不会保错：

```JavaScript
var i = function(){ return 10; }();
```

但是我们依然推荐加上()：

```JavaScript
var i = (function(){ return 10; }());
```

为什么？因为我们在阅读代码的时候，如果function内部代码量庞大，我们不得不滚动到最后去查看function(){}后是否带有()来确定i值是个function还是function内部的返回值。所以为了代码的可读性，请尽量加上()无论是否已经是表达式。

- 立即执行函数与闭包的暧昧关系

立即执行函数能配合闭包保存状态。

像普通的函数传参一样，立即执行函数也能传参数。如果在函数内部再定义一个函数，而里面的那个函数能引用外部的变量和参数（闭包），利用这一点，我们能使用立即执行函数锁住变量保存状态。

```JavaScript
// 并不会像你想象那样的执行，因为i的值没有被锁住
// 当我们点击链接的时候，其实for循环已经执行完了
// 于是在点击的时候i的值其实已经是elems.length了
var elems = document.getElementsByTagName( 'a' );

for ( var i = 0; i < elems.length; i++ ) {

  elems[ i ].addEventListener( 'click', function(e){
```

```
    e.preventDefault();
    alert( 'I am link #' + i );
  }, 'false' );

}

// 这次我们得到了想要的结果
// 因为在立即执行函数内部，i的值传给了lockedIndex，并且被锁在内存中
// 尽管for循环结束后i的值已经改变，但是立即执行函数内部lockedIndex的值并不会改变
var elems = document.getElementsByTagName( 'a' );

for ( var i = 0; i < elems.length; i++ ) {

  (function( lockedInIndex ){

    elems[ i ].addEventListener( 'click', function(e){
      e.preventDefault();
      alert( 'I am link #' + lockedInIndex );
    }, 'false' );

  })( i );

}

// 你也可以这样，但是毫无疑问上面的代码更具有可读性
var elems = document.getElementsByTagName( 'a' );

for ( var i = 0; i < elems.length; i++ ) {

  elems[ i ].addEventListener( 'click', (function( lockedInIndex ){
    return function(e){
      e.preventDefault();
      alert( 'I am link #' + lockedInIndex );
    };
  })( i ), 'false' );

}
```

其实上面代码的lockedIndex也可以换成i，因为两个i是在不同的作用域里，所以不会互相干扰，但是写成不同的名字更好解释。以上便是立即执行函数+闭包的作用。

- 我为什么更愿意称它是“立即执行函数”而不是“自执行函数”

IIFE的称谓在现在似乎已经得到了广泛推广（不知道是不是原文作者的功劳？），而原文写于10年，似乎当时流行的称呼是自执行函数（Self-executing anonymous function），接下去作者开始为了说明立即执行函数的称呼好于自执行函数的称呼开始据理力争，有点咬文嚼字，不过也蛮有意思的，我们来看看作者说了些什么。

JavaScript

```
// 这是一个自执行函数，函数内部执行的是自己，递归调用
function foo() { foo(); }

// 这是一个自执行匿名函数，因为它没有函数名
// 所以如果要递归调用自己的话必须用arguments.callee
var foo = function() { arguments.callee(); };

// 这可能也算是个自执行匿名函数，但仅仅是foo标志引用它自身
// 如果你将foo改变成其它的，你将得到一个used-to-self-execute匿名函数
var foo = function() { foo(); };

// 有些人叫它自执行匿名函数，尽管它没有执行自己，只是立即执行而已
(function(){ /* code */ }());

// 给函数表达式添加了标志名称，可以方便debug
// 但是一旦添加了标志名称，这个函数就不再是匿名的了
(function foo(){ /* code */ }());

// 立即执行函数也可以自执行，不过不常用罢了
(function(){ arguments.callee(); }());
```

```
21 (function foo(){ foo(); }());
```

我的理解是作者认为自执行函数是函数内部调用自己（递归调用），而立即执行函数就如字面意思，该函数立即执行即可。其实现在也不用去管它了，就叫IIFE好了。

- 最后的旁白：模块模式

立即执行函数在模块化中也大有用处。用立即执行函数处理模块化可以减少全局变量造成的空间污染，构造更多的私有变量。

JavaScript

```
// 创建一个立即执行的匿名函数
// 该函数返回一个对象，包含你要暴露的属性
// 如下代码如果不使用立即执行函数，就会多一个属性i
// 如果有了属性i，我们就能调用counter.i改变i的值
// 对我们来说这种不确定的因素越少越好

var counter = (function(){
  var i = 0;

  return {
    get: function(){
      return i;
    },
    set: function( val ){
      i = val;
    },
    increment: function() {
      return ++i;
    }
  };
}());

// counter其实是一个对象

counter.get(); // 0
counter.set( 3 );
counter.increment(); // 4
counter.increment(); // 5

counter.i; // undefined i并不是counter的属性
i; // ReferenceError: i is not defined (函数内部的是局部变量)
```

扩展阅读

如果你愿意了解更多内容，特别是关于函数和模块模式的内容，建议阅读下列文章。

1. ECMA-262-3 in detail. Chapter 5. Functions. – Dmitry A. Soshnikov
2. Functions and function scope – Mozilla Developer Network
3. Named function expressions – Juriy “kangax” Zaytsev
4. JavaScript Module Pattern: In-Depth – Ben Cherry
5. Closures explained with JavaScript – Nick Morgan

如果有人让你推荐前端技术书，请让他看这个列表 -> 《经典前端技术书籍》