rainy.im

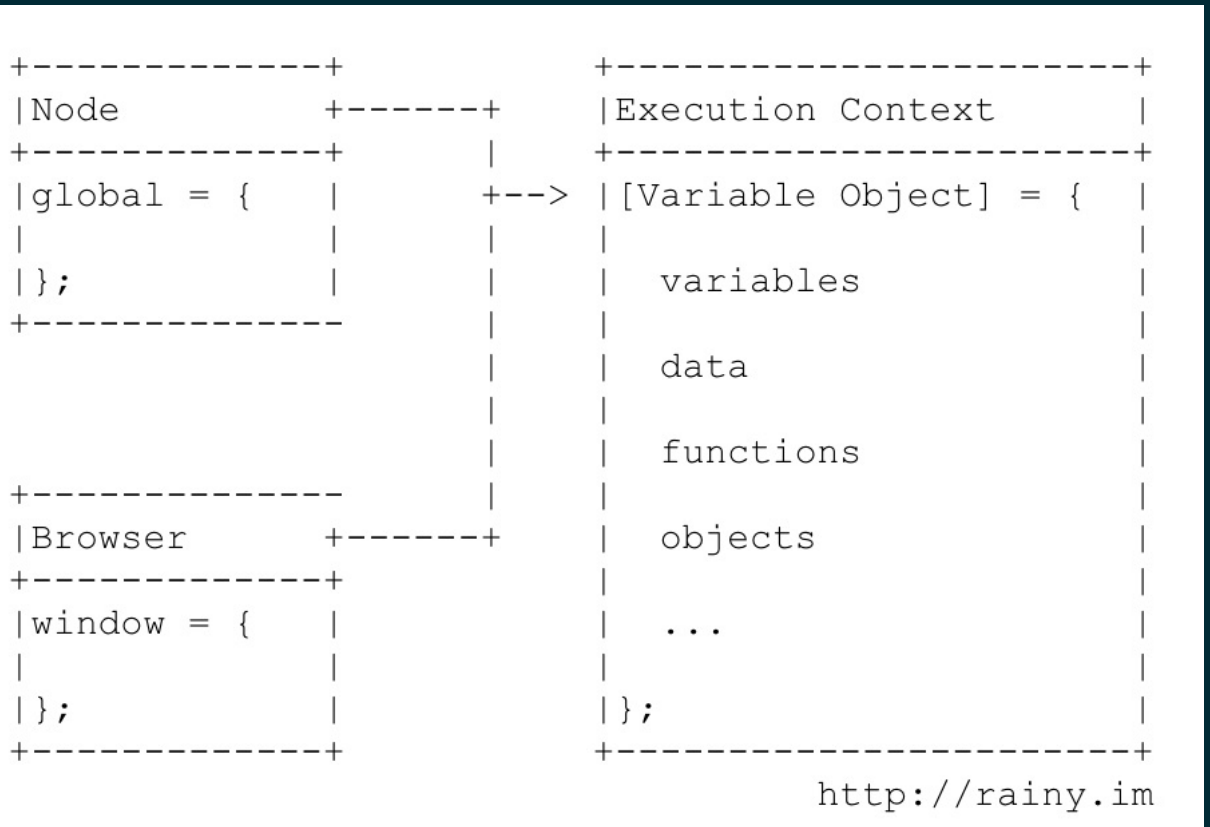
图解Javascript上下文与作用域

rainy • Jul 4th, 2015

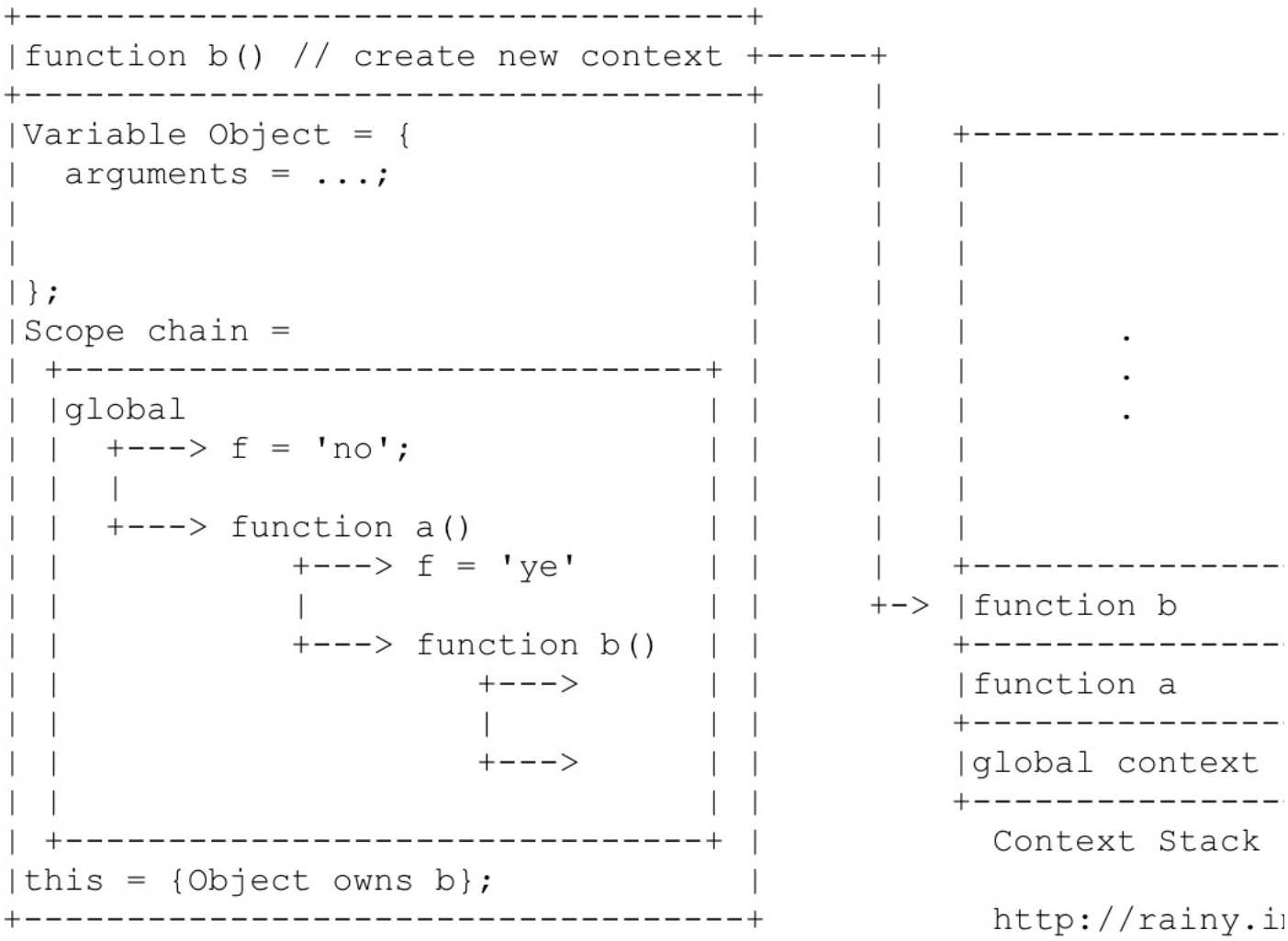
本文尝试阐述Javascript中的上下文与作用域背后的机制，主要涉及到执行上下文（execution context）、作用域链（scope chain）、闭包（closure）、`this`等概念。

Execution context

执行上下文（简称上下文）决定了Js执行过程中可以获取哪些变量、函数、数据，一段程序可能被分割成许多不同的上下文，每一个上下文都会绑定一个变量对象（variable object），它就像一个容器，用来存储当前上下文中所有已定义或可获取的变量、函数等。位于最顶端或最外层的上下文称为全局上下文（global context），全局上下文取决于执行环境，如Node中的`global`和Browser中的`window`：



需要注意的是，上下文与作用域（scope）是不同的概念。Js本身是单线程的，每当有function被执行时，就会产生一个新的上下文，这一上下文会被压入Js的上下文堆栈（context stack）中，function执行结束后则被弹出，因此Js解释器总是在栈顶上下文中执行。在生成新的上下文时，首先会绑定该上下文的变量对象，其中包括`arguments`和该函数中定义的变量；之后会创建属于该上下文的作用域链（scope chain），最后将`this`赋予这一function所属的Object，这一过程可以通过下图表示：



this

上文提到`this`被赋予function所属的Object，具体来说，当function是定义在global对中时，`this`指向global；当function作为Object的方法时，`this`指向该Object：

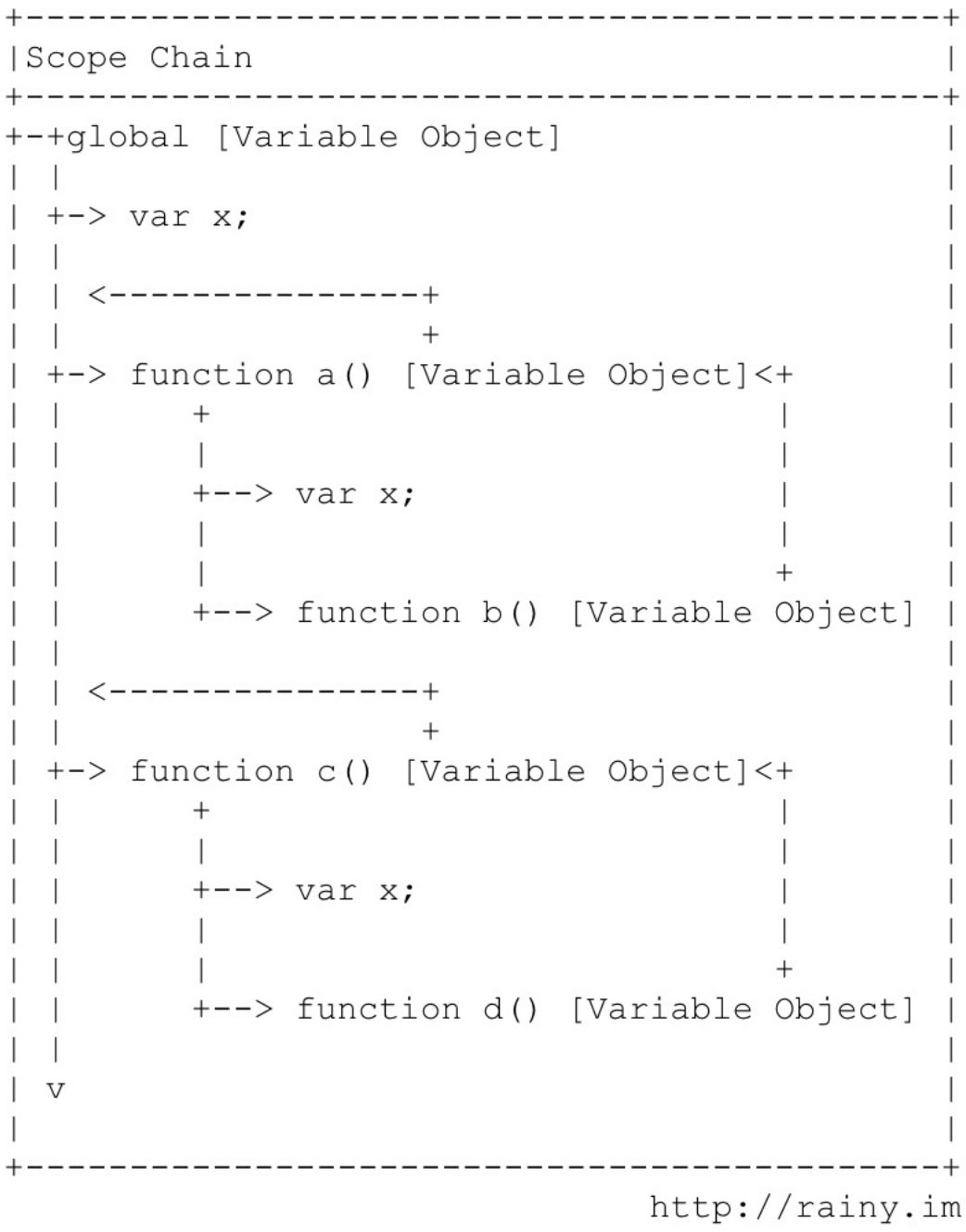
```
var x = 1;
var f = function(){
  console.log(this.x);
}
f();  // -> 1

var ff = function(){
  this.x = 2;
  console.log(this.x);
}
ff(); // -> 2
x     // -> 2

var o = {x: "o's x", f: f};
o.f(); // "o's x"
```

Scope chain

上文提到，在function被执行时生成新的上下文时会先绑定当前上下文的变量对象，再创建作用域链。我们知道function的定义是可以嵌套在其他function所创建的上下文中，也可以并列地定义在同一个上下文中（如global）。作用域链实际上就是自下而上地将所有嵌套定义的上下文所绑定的变量对象串接到一起，使嵌套的function可以“继承”上层上下文的变量，而并列的function之间互不干扰：




```
var x = 'global';
function a(){
  var x = "a's x";
  function b(){
    var y = "b's y";
    console.log(x);
  };
  b();
}
function c(){
  var x = "c's x";
  function d(){
    console.log(y);
  };
  d();
}
a();  // -> "a's x"
c();  // -> ReferenceError: y is not defined
x     // -> "global"
y     // -> ReferenceError: y is not defined
```

Closure

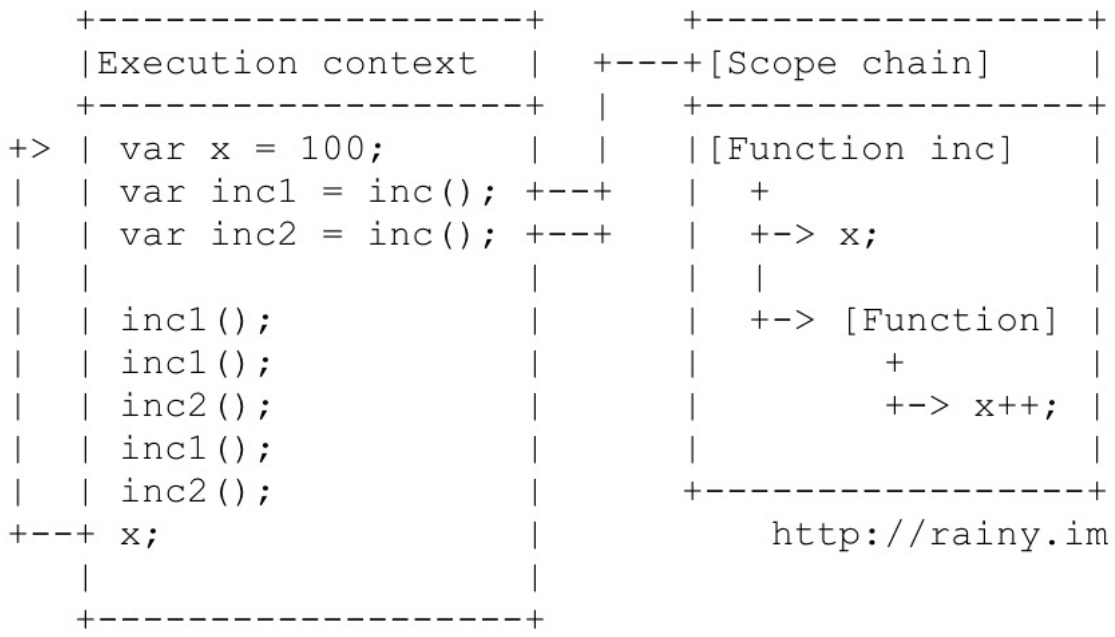
如果理解了上文中提到的上下文与作用域链的机制，再来看闭包的概念就很清楚了。每个function在调用时会创建新的上下文及作用域链，而作用域链就是将外层（上层）上下文所绑定的变量对象逐一串连起来，使当前function可以获取外层上下文的变量、数据等。如果我们在function中定义新的function，同时将内层function作为值返回，那么内层function所包含的作用域链将会一起返回，即使内层function在其他上下文中执行，其内部的作用域链仍然保持着原有的数据，而当前的上下文可能无法获取原先外层function中的数据，使得function内部的作用域链被保护起来，从而形成“闭包”。看下面的例子：

```
var x = 100;
var inc = function(){
  var x = 0;
  return function(){
    console.log(x++);
  };
};

var inc1 = inc();
var inc2 = inc();

inc1();  // -> 0
inc1();  // -> 1
inc2();  // -> 0
inc1();  // -> 2
inc2();  // -> 1
x;       // -> 100
```

执行过程如下图所示，`inc`内部返回的匿名function在创建时生成的作用域链包括了`inc`中的`x`，即使后来赋值给`inc1`和`inc2`之后，直接在`global context`下调用，它们的作用域链仍然是由定义中所处的上下文环境决定，而且由于`x`是在`function inc`中定义的，无法被外层的`global context`所改变，从而实现了闭包的效果：



this in closure

我们已经反复提到执行上下文和作用域实际上是通过function创建、分割的，而function中的`this`与作用域链不同，它是由执行该function时当前所处的Object环境所决定的，这也是`this`最容易被混淆用错的一点。一般情况下的例子如下：

```
var name = "global";
var o = {
  name: "o",
  getName: function(){
    return this.name
  }
};
o.getName();  // -> "o"
```

由于执行`o.getName()`时`getName`所绑定的`this`是调用它的`o`，所以此时`this == o`；更容易搞混的是在closure条件下：

```
var name = "global";
var oo = {
  name: "oo",
  getNameFunc: function(){
    return function(){
      return this.name;
    };
  }
}
oo.getNameFunc()();  // -> "global"
```

此时闭包函数被`return`后调用相当于：

```
getName = oo.getNameFunc();
getName();  // -> "global"
```

侧栏导航

换一个更明显的例子：

```
var ooo = {
  name: "ooo",
  getName: oo.getNameFunc() // 此时闭包函数的this被绑定到新的Object
};
ooo.getName();  // -> "ooo"
```

当然，有时候为了避免闭包中的`this`在执行时被替换，可以采取下面的方法：

```
var name = "global";
var oooo = {
  name: "ox4",
  getNameFunc: function(){
    var self = this;
    return function(){
       return self.name;
    };
  }
};
oooo.getNameFunc()(); // -> "ox4"
```

或者是在调用时强行定义执行的Object：

```
var name = "global";
var oo = {
  name: "oo",
  getNameFunc: function(){
    return function(){
      return this.name;
    };
  }
}
oo.getNameFunc()();  // -> "global"
oo.getNameFunc().bind(oo)(); // -> "oo"
```

总结

Js是一门很有趣的语言，由于它的很多特性是针对HTML中DOM的操作，因而显得随意而略失严谨，但随着前端的不断繁荣发展和Node的兴起，Js已经不再是"toy language"或是jQuery时代的"CSS扩展"，本文提到的这些概念无论是对新手还是从传统Web开发中过度过来的Js开发人员来说，都很容易被混淆或误解，希望本文可以有所帮助。

写这篇总结的原因是我在Github上分享的Learn javascript in one picture，刚开始有人质疑这只能算是一张语法表（syntax cheat sheet），根本不会涉及更深层的闭包、作用域等内容，但是出乎意料的是这个项目竟然获得3000多个star，所以不能虎头蛇尾，以上。

References

1. Understanding Scope and Context in JavaScript
2. this - JavaScript | MDN
3. 闭包 - JavaScript | MDN

Js

- END -

Recommend 分享 按评分高低排序

stonex • 20天前

写的很好！我觉得这个也是个挺有趣的例子，也是说明了this的本质吧：

```
var x = 'global';
function a(){
var x = "a's x";
function b(){
var y = "b's y";
console.log(this.x);
};
b();
}
a(); //global
```

还有一个小问题，有空的话麻烦您指教，谢谢。在闭包的第一个例子中，
图中闭包返回的Scope Chain是没有画完没？按我的理解应该是从global开始的吧。"由于x是在function inc中定义的，无法被外层的global context所改变"，这句话该怎么理解呢？按我的理解是因为function inc内的x在内层的原因吧，global的x才未被改变的吧。做了个小测试。

```
var x = 100;
var y = 99;
```

查看更多

回复 • 分享 ›

rainy Author ➚ stonex • 20天前

你理解的意思是对的。我这里想要表达的意思是闭包中通过Scope Chain“由内而外”地来获取变量的值，因为inc中重新定义了局部变量x，因此在返回的闭包函数中的x读到的是inc中的x，而inc中的x是global中无法改变的。

回复 • 分享 ›

GeekPlux • 1个月前

受益匪浅。p.s.请问一下配图是用什么工具可以画出来，这种图好清晰

回复 • 分享 ›

rainy Author ➚ GeekPlux • 1个月前

http://asciiflow.com/

回复 • 分享 ›