this: caveat in javaScript's scoping rule

Yiran Sheng

03/09/2016

Lexical vs. Dynamic Scoping

The use of local variables — of variable names with limited scope, that only exist within a specific function — helps avoid the risk of a name collision between two identically named variables. However, there are two very different approaches to answering this question: What does it mean to be "within" a function?

In lexical scoping (or lexical scope; also called static scoping or static scope), if a variable name's scope is a certain function, then its scope is the program text of the function definition: within that text, the variable name exists, and is bound to the variable's value, but outside that text, the variable name does not exist. By contrast, in dynamic scoping (or dynamic scope), if a variable name's scope is a certain function, then its scope is the time-period during which the function is executing: while the function is running, the variable name exists, and is bound to its variable, but after the function returns, the variable name does not exist. This means that if function f invokes a separately defined function g, then under lexical scoping, function g does not have access to f's local variables (assuming the text of g is not inside the text of f), while under dynamic scoping, function g does have access to f's local variables (since g is invoked during the invocation of f).

```
function main() {
  var x = 'apple';

  function inner() {
    var x = 'orange';
    logFrom('inner');
  }

  function logFrom(calledFrom) {
    // variable x used here
    console.log('Callsite: ', calledFrom, ' x is: ', x);
  }
  logFrom('main');
  inner();
}
```

JavaScript, being lexically scoped, outputs the following:

```
Callsite:  main  x is:  apple
Callsite:  inner  x is:  apple
```

For a dynamically scoped language, the output will be:

```
Callsite:  main  x is:  apple
Callsite:  inner  x is:  orange
```

So, what exactly does this program print? It depends on the scoping rules. If the language of this program is one that uses lexical scoping, then g prints and modifies the global variable x (because g is defined outside f), so the program prints 1 and then 2. By contrast, if this language uses dynamic scoping, then g prints and modifies f's local variable x (because g is called from within f), so the program prints 3 and then 1. (As it happens, the language of the program is Bash, which uses dynamic scoping; so the program prints 3 and then 1.)

JavaScript Uses Lexical Scoping ... Mostly

```
function logArgs(...args) {
  console.log('Context: ', this);
  console.log('Arguments: ', args);
}

var obj = {
  log : logArgs
};

// this shows the binding of this can only be determined
// at runtime

if(Math.random() > 0.66) {
  logArgs();  // Context will be window/global object
} else if (Math.random() > 0.33) {
  logArgs.call('This')  // Context will be string 'This'
} else {
  obj.log();  // Context will be obj
}
```

```
var self = this;
this.on('change', function(data) {
  self.method(data);
});
```

```
var that = this;
element.addEventListener('click', function() {
  that.doSomething();
});
```

The trick here is to assign `this` to a variable (common names: `that`, `self`, `context`), eliminate the usage of it inside function bodies, and rely on lexical scoping rule to find the binding to the correct value. Some fun stats using github search:

- 8 million `var self = this;` (Link)
- 61 million `var that = this;` (Link)

```
this.on('change', this.method.bind(this));
```

ES6: Introducing Arrow Functions

Syntax-wise, ES6 introduces two new way of defining a function, both removed the usage of the `function` keyword:

```
class A {
  method() {
    // ...
  }
```

```
}

let f = () => {
   // ...
}
```

```
function oldWay() {
  let newWay = () => {
    console.log(this);  // this may still surprise you
  }
}
```

```
class A {
  func() {
    return this.value;
  }
}
let f = A.prototype.func;
f.call(null);
```