编程

April 20, 2014

Part I

C 语言

Part II

javascript

Chapter 8

JavaScript 语言

面向对象, 有两种实现面向对象的方式: 基于类和基于原型的继承.

基于类的继承. 类是模板, 继承的是行为。

基于原型的继承, 所有的都是对象 (当然还有基本类型), 继承的是状态和行为.JavaScript 是基于原型的继承，这里只有对象，没有类的概念。

Mocha 8.1 【JavaScript】

```
1 var a = {};
2 a.b = 10;
3 console.log(a.b); // 10
4 console.log(a["b"]); // 10
5 a.foo = function() {
6   console.log("invoke foo");
7 };
8 a.foo();
```

JavasScript 中对象的定义是: Object 是 Property 的集合，property 是一个值或者对象的引用. JavaScript 对象是一组属性的集合, 这些属性引用的是一个对象或者基本类型.

首先，第一行生成一个对象，并将它赋值给 a。然后第二行，a.b = 10，这个时候，因为 a 没有 b 这个 property，所以给他生成了一个 b property，并将 10 复制给他，第三行，打印 a.b，结果是 10. 第四行，对对象属性的访问有两种方式，. 和 []。不同之处是. 后面是直接跟标识符，而 [] 中是字符串，注意第四行的 b 是有引号的。

在在后面，将我们定义的函数赋值给 foo property，这里又不同于 Java，我们前面说了，JavaScript 中一切皆对象，所以函数也是对象，他是一类比较特殊的对象，可以通过被调用。因为函数是对象，所以他也可以赋值给对象的属性。赋值之后，我们可以调用他。

8.1 JavaScript 的类型

8.1.1 基本类型

JavaScript 只有对象这种说法也不是太准确，JavaScript 有基本类型和对象两类。只是在使用基本类型的时候，如果必要，基本类型会被自动转换为对象类型。

基本类型有 Undefined, Null, Boolean, String 和 Number 类型

类型	说明
Undefined	有且只有一个值，undefined。
Null	有唯一的值 null
Boolean	两个值，true 和 false
String	如”abc”
Number	只有浮点型表示。NaN，Infinity, -Infinity

JavaScript 中的 undefined 就是这个值，只是 undefined 不是关键字，而是一个全局变量。可能被赋其他值。所以可以使用 void 表达式来表示，如 void 0

Mocha 8.2 【JavaScript】

```
1     it('void 0 is undefined', function(){
2       should(void 0).be.exactly(undefined);
3     });
```

8.1.2 对象类型

- 对象就是 property 的集合。
- 给对象 property 赋值，如果存在，就修改值，如果不存在，就创建一个新的 property，并赋值。
- 函数也是对象，可以赋值给 property。

对于 property，我们可以先将他看成有两类 property，一类就是我们上面看到的，实际上，他还要细分为 data property 和 access property。上面和下面所说的也就是 data property。我们平时写代码生成的也是 data property 类型。

再来有一类我们叫做 internal property，他是在我们程序级别是看不到的，而是提供给语言内部实现级别所使用的。例如我们将要讨论的原型，每个对象都有原型属性，但是我们在程序中无法通过. 的方式来访问他，我们在谈论到 internal property 的时候，会使用 [[]] 来表示，例如，原型属性我们会表示成 [[prototype]]。

8.1.3 Prototype

js 是通过原型的方式来实现继承的。原型实际上就是对象的一个 internal property [[prototype]]，每个对象都有 [[prototype]] 属性，他指向一个对象，而原型本身也有 [[prototype]] 属性，这样一直链接到最顶端的一个特殊对象，他的原型是空的（是 null，不是 undefined），所有的对象的原型链的顶端都是这个对象。这就是我们说的原型链。

Mocha 8.3 【JavaScript】原型链的顶端是 null

```
1     it('__proto__ of Object.prototype is null', function(){
2       should(Object.prototype.__proto__).be.exactly(null);
3     });
```

Mocha 8.4 【JavaScript】还有可能是 undefined，这个在花时间看看

```
1
2 var obj = Object.create(null);
```

在浏览器中，我们可以通过属性名__proto__来引用内部属性 [[prototype]]。

然后我们说说原型的作用，他就是帮我们来解析. 和 [] 取得什么样的值的。当我们通过. 的方式来读取一个对象的 property 的时候，会在当前对象中查找看看是否有这个 property，如果有，就返回，如果没有，就会尝试着在他的 [[prototype]] 上来查找，如果找到了就返回，如果没有找到，就继续在这个链往上找，直到找到最顶，如果还没有找到，那就会返回 undefined。

Mocha 8.5 【JavaScript】

```
1 var proto = {bar : 10};
2 var foo = Object.create(proto);
3
4 console.log(foo.bar); // 10
5 console.log(foo.bar2); // undefined
6
7 proto.bar2 = 20;
8 console.log(foo.bar2); // 20
```

这里的 Object.create 函数是用来创建一个对象，他的原型是传入的参数。在第四行，因为在原型对象上定义了 bar，所以可以取到值 10，而 bar2 没有定义，所以得不到值，返回 undefined。然后在第七行，我们给 proto 的 bar2 赋值 20，这样当我们再次调用 foo.bar2 的时候，在原型链上查找，可以找到值 20 了。

这里说到的是取值，实现了继承。接着是赋值，赋值很简单，如果存在就修改值，不存在就新增这个属性，这里所说的存在与不存在是指当前对象，而不是只原型链上存不存在。

Mocha 8.6 【JavaScript】

```
1  var proto = {x: 10};
2
3  var foo = Object.create(proto);
4  var bar = Object.create(proto);
5
6  console.log(foo.x); // 10
7  console.log(bar.x); // 10
8
9  foo.x = 20;
10
11 console.log(foo.x); // 20
12 console.log(bar.x); // 10
13 console.log(proto.x); // 10
```

这里可以看到 foo 和 bar 都是继承 proto，所以 x 都等于 10，而在 foo.x = 20 之后，foo 上创建了 x，并且赋值 20。所以当取 foo.x 是等于 20。而 bar.x 和 proto.x 还是 10。

8.1.4 Property

Data Property 和 Access Property

从逻辑上来讲，Object 是一系列 property 的集合，其中 property 可能是 data property 或者 access property。

- data property 就是一个 key 和一个 ECMAScript 语言类型的值和一系列 boolean 型的 Attribute 组成。
- access property 就是一个 key 和一个或者两个 accessor function 和一系列 boolean 型的 Attribute 组成。

这里的 key，要么是一个 string，要么是一个 symbol(ECMAScript6)。

ECMAScript 的使用 key 来标示 Property，有两中方式来访问 Property。get 和 set。

Attribute

Attribute 则是规范用来定义和解释 Property 的。

Data Property 的 Attribute

属性名	属性值	解释
[[Value]]	任意 ECMAScript 语言类型	属性的值
[[Writable]]	Boolean	如果是 false，则通过 [[Set]] 来设定 [[Value]] 不会成功。
[[Enumerable]]	Boolean	如果是 true，则可以在 for-in 中被迭代。
[[Configurable]]	Boolean	如果是 false，则删除，从 Data 转换成 accessor Property，修改除[[Value]] 之外的值都会失败

Mocha 8.7 【JavaScript】Data Property 中 Attribute 的默认值

```
1  it(' should return correct default descriptor', function(){
2    var foo  = Object.create({}, {
3      bar: {}
4    });
5    var descriptor = Object.getOwnPropertyDescriptor(foo, 'bar');
6    descriptor.should.eql(
7      {
8        value: undefined,
9        writable: false,
10       enumerable: false,
11       configurable: false
12     });
13 });
```

Mocha 8.8 【JavaScript】writable 为 false 时，不能修改 Value

```
1  it(' should not able to change property value when writable is false', function(){
2    var foo = Object.create({}, {
3      bar: {value: 100, writable: false}
4    })
5
6    foo.bar = 200;
7    foo.bar.should.be.exactly(100).and.be.Number;
8  });
```

Mocha 8.9 【JavaScript】enumerable 为 false 时，key 不会再 for in 语句中出现

```
1  it(' should not be able to enumerate in for in statement when enumerable is false', function()
     {
2    var foo = Object.create({}, {
3      bar: {value: 100, enumerable: false}
4    });
5
6    for (var key in foo)
7    {
8      key.should.not.be.exactly('bar');
9    }
10 });
```

Mocha 8.10 【JavaScript】configurable 为 false 时，不能修改除 Value 之外的 Attributes

```
    it(' should not be able to modify attributes of property excluded [[Value]] when configurable
        is false', function(){
      var foo = Object.create({}, {
        bar: {value: 100, writable: true, enumerable: false, configurable: false}
      });

      var descriptor = {value: 200, writable: false, enumerable: true, configurable: true};
      (function(){
        Object.defineProperty(foo, 'bar', descriptor);
      }).should.throw();

      foo.bar = 200;
      foo.bar.should.be.exactly(200).and.be.Number;
    });
```

属性名	属性值	解释
[[Get]]		
[[Set]]		
[[Enumerable]]		
[[Configurable]]	Boolean	

Mocha 8.11 【JavaScript】Accessor Property 的 Attributs 的默认值

```
    it('should get correct default descriptor', function(){
      var foo = Object.create({}, {
        bar : {get: undefined}
      });
      var descriptor = Object.getOwnPropertyDescriptor(foo, 'bar');
      descriptor.should.eql({
        get: undefined,
        set: undefined,
        enumerable: false,
        configurable: false
      });
    });
```

Mocha 8.12 【JavaScript】对于 Accessor Property，接受请求的对象会被当做 this

```
    it('base should be treat as this when assign or get property value', function(){
      var foo = Object.create({}, {
        bar: {get: function(){
          return this._bar;
        },
        set: function(value){
          this._bar = value;
        }}
      });
      foo.bar = 100;
      foo.should.have.ownProperty('_bar').be.exactly(100);
      foo.should.have.ownProperty('bar').be.exactly(100);
    });
```

对 **Property** 进行赋值操作

这里稍微比前面说如果对象上不存在某个被赋值的属性，就创建会要复杂点。对于对象上已经存在这个 Property 的情况，赋值的赋值，调用 set 方法的调用 set 方法。

而对于 Property 在对象上不存在的，可以分三种情况处理：

- 如果这个 Property 在原型链上不存在的话，则会创建一个 Data Property

  Mocha 8.13 【JavaScript】Property 在原型链上也不存在的话

```
    it('should create a own data property when assignning a not exist property', function() {
      foo = Object.create({}, {});
      foo.bar = 100;
      foo.should.have.ownProperty('bar').be.exactly(100);
      var descriptor = Object.getOwnPropertyDescriptor(foo, 'bar');
```

```
6       descriptor.should.eql({
7         value: 100,
8         writable: true,
9         enumerable: true,
10        configurable: true
11      })
12    });
```

- 如果这个 Property 在原型链上存在一个 Accessor Property 的话，直接调用这个 Accessor Property 的[[Set]]方法，reciver 会被当做 this 来传给这个方法。

Mocha 8.14 【JavaScript】原型链上存在 Accessor Property 的情况

```
1     it('should not create own propert when assignning a inherited accessor property',
         function(){
2       var x = 100
3       var parent= {};
4       Object.defineProperty(parent, 'bar', {set: function(value){x = value}});
5
6       foo = Object.create(parent, {});
7
8       foo.should.not.have.ownProperty('bar');
9       foo.bar = 200;
10      foo.should.not.have.ownProperty('bar');
11      x.should.be.exactly(200);
12    });
```

- 如果原型链上存在一个 Data Property 的话，还是直接在当前对象上创建一个 Data Property.

Mocha 8.15 【JavaScript】原型链上存在 Data Property 的情况

```
1     it('should create a own data property when assignning a inherited data property',
         function(){
2       foo = Object.create({bar: 200}, {});
3
4       foo.should.not.have.ownProperty('bar');
5       foo.bar = 100;
6       foo.should.have.ownProperty('bar').be.exactly(100);
7     });
```

8.1.5 == 和 ===

- ToNumber

参数	值
Undefined	NaN
Null	0
Boolean	true 转换为 1，false 转换为 0
String	将 string 转换为数字
Object	先转换为基本类型，然后再按上面的转换

- PrimtiveValue

  根据参数 Hint 来判断怎么执行，如果 Hint 是 number，对象转换为基本类型看 valueOf 是不是返回基本类型，不是再看看 toString 是不是返回基本类型，如果还不是则抛错。

  如果 Hint 是 String，对象转换为基本类型看 toString 是不是返回基本类型，不是再看看 valueOf 啊、是不是返回基本类型，如果还不是则抛错。

  如果不带 Hint，则按 Hint 是 number 来执行。

- ==

  1. 对左右引用类型求值，设左边是 lval, 右边是 rval;
  2. 如果 lval 和 rval 是相同类型的话。
     - 基本类型的话，按正常的相等性进行比较.

  - NaN 不等于其他值.
  - 不是基本类型时，如果不是引用同一个对象，则两者不相等。

3. 当 lval 和 rval 是不同类型的时候
  - undefined 等于 null。
  - undefined 和 null 不等于其他值
  - 如果有 boolean 型，则将其转换为 number，再比较
  - 如果有一方是对象，一方是数字或者字符串，将对象转换为基本类型再比较
  - 如果是数字和字符串比较，转换为数字在比较
  - 其他情况一律不相等.

- === 如果是不同类型，直接返回 false，如果是相同类型，比较结果和 == 一样。

Mocha 8.16 【JavaScript】

```
1    it('test between same type', function(){
2      should(undefined == undefined).be.ok;
3      should(null == null).be.ok;
4      should(NaN == 1).not.be.ok;
5      should(NaN == NaN).not.be.ok;
6      should({} == {}).not.be.ok;
7    });
```

对于相同类型的值得比较，NaN 不和自己本身以及任何数相等。两个不同的对象是不相等的。

Mocha 8.17 【JavaScript】

```
1    it('test between different type', function(){
2      should(undefined == null).be.ok;
3      should(0 == undefined).not.be.ok;
4      should(0 == null).not.be.ok;
5      should(2 == true).not.be.ok;
6      should(1 == true).be.ok;
7      should(0 == false).be.ok;
8
9      should(1 == '1').be.ok;
10
11     var a = {
12       toString: function() {return "a"},
13       valueOf: function(){return 1;}
14     }
15     should(a == 1).be.ok;
16     should(a == "a").not.be.ok;
17
18     should(true == a).be.ok;
19   });
```

Mocha 8.18 【JavaScript】

```
1    it('should throw exception', function(){
2      (function(){
3        var a = {toString: undefined, valueOf: undefined};
4        a == 1;
5      }).should.throw();
6      // 先调用valueOf，再调用toString(一般来说),
7      // 如果都不是函数，  或者返回的都不是基本类型，报错
8      (function(){
9        var a = {toString: function(){ return {}}, valueOf: function(){return {}}};
10       a == 1;
11     }).should.throw();
12   });
```

对于不同的类型，undefined 和 null 是相等的，因为 boolean 的转化造成基本类型是 1 和 0，所以 2 是不会等于 boolean 型的。对于有字符串或者数字参与的比较，需要将对象转变为基本类型。如果和 valueOf 和 toString 都不存在，则会在转换的过程抛出错误。

8.1.6 typeof 和 instanceof

typeof 就是返回后面所跟引用的类型，这个值是固定的。

类型	结果
Undefined	”undefined”
Null	”object”
Boolean	”boolean”
Number	”number”
String	”string”
Host object	Implementation-dependent
Function object	”function”
Any other object	”object”

typeof 的运算过程，为什么不会抛出异常？

instanceof 运算符用来检测 constructor.prototype 是否存在于参数 object 的原型链上.

Mocha 8.19 【JavaScript】

```
1   it('test if Constructor.prototype on the prototype chain', function(){
2     function Foo(){}
3
4     var foo = Object.create(Foo.prototype);
5
6     should(foo instanceof Foo).be.exactly(true);
7
8     var foo2 = Object.create(Foo);
9
10    should(foo2 instanceof Foo).be.exactly(false);
11  });
```

8.1.7 ECMA spec 中和本节相关概念

8.2 关于函数部分的准备知识

8.2.1 Lexical Environment

Lexical Environment 是规范中使用定义变量和函数标识符的关系结构。Lexical Environement 由 Environment Record 和一个指向外部 Lexical Environment 的引用（可空）。

Environment Record 有两种类型，一种是 Declarative Environment Record, 他不提供 this 值, 直接返回返回 undefined。

一种是 Object Environment Record。Declarative Environment Record 用于定义标识符，而 Object Environment 将他的 binding 对象和标示符绑定，标识符会被绑定到 binding 对象的对应 Property 上，如果自身的 provideThis 是 true，绑定的 object 会被当做 this 值来提供，provideThis 的默认值是 false。

Global Environment 是一个唯一的 Lexical Environment，在代码执行之前被创建，他的 Environment Record 是一个 Object Environment Record，这个 Environment Record 的绑定对象是 Global Object，他的 outer lexical Environment 是 null。

Execution Context，进入一段代码，就进入了一个 execution context。execution context 是栈结构，最顶上的 execution context 是 running execution context. 所有的操作都是对 running execution context 的操作。

execution context 有三部分，

- 一个是 ThisBinding，用来保存 this 这个 keyword 引用的对象；
- lexicalEnvironment, 用来解析标识符引用的，这个在执行过程中引用的 lexicalEnvironment 可能是会改变的；
- variableEnvironment，用来保存执行过程中定义的函数和变量，这个在执行过程中是不会改变的。

在创建 execution context 时，lexicalEnvironment 和 variableEnvironment 是引用同一个值。

什么时候 **lexicalEnvironment** 会变化？

with statement with statement 会创建一个新 ObjectEnvironment，绑定 object，outer 为原来的 lexicalEnvironment。同时将 objectEnvironment 的 bingThis 设为 true，表示提供 this，this 就是 object，用于调用 function 的时候提供 this。

Mocha 8.20 【JavaScript】

```
    it('should been add to with variable environment of running execution context', function(){
      var foo = {};

      (function(){
        bar;
      }).should.throw();

      with(foo){
        eval('var bar = 200');
      }
      bar.should.be.exactly(200);
      foo.should.eql({});
    });
```

此时 lexicalEnvironment 的 envRec 添加了一个 object Environment record,但是 variableEnvironment 没有变,所以 bar 任然是添加到 running execution context 的 variableEnvironment 上。等出了 with 的范围,lexicalEnvironment 又还原成和 variableEnvironment 为同一个对象。

Mocha 8.21 【JavaScript】

```
    it('should been add to with variable environment of running execution context', function(){
      var foo = {};
      should(bar).be.exactly(undefined);
      with(foo){
        var bar = 200;
      }
      bar.should.be.exactly(200);
      foo.should.eql({});
    });
```

catch statment catch(e)

会创建一个 DeclarativeEnvironment（不同于 with，with 创建的是 ObjectEnvironment），将他的 outer 指向 running execution context 的 lexicalEnvironment，将传入的对象绑定到 e 上。然后将新的 DeclarativeEnvironment 当做 lexicalEnvironment。

返回执行结果，将 running execution context 的 lexicalEnvironment 回复成原来的那个。

8.2.2 解析标识符

reference Type 是用来在规范中说明算法使用的，reference Type 由三部分组成，base，name,strict。对于标识符，base 是标识符引用的对象真正所在的那个 Environment record；name 就是标识符名字，strict 看看是否是 strict mode。

GetIdentifierReference (lex, name, strict) 算法

1. 如果 lex 是 null，return base: undefined, name: name, strict: strict（unresolveReference）
2. let envRec = lex.environmentRecord
3. 如果 envRec 上有绑定 name，return base: envRec, name: name, strict: strict
4. 否则 let lex = lex.outer, call GetIdentifierReference (lex, name, strict)

8.2.3 建立 Execution Context

global execution context 创建:

- this: global object
- LexicalEnvironment: global environment
- VariableEnvironment: global environment

eval 创建：

如果没有 calling context，步骤同上；否则

- this: this of calling context
- LexicalEnvironment: lexicalEnvironment of calling context
- VariableEnvironment: variableEnvironment of calling context

Mocha 8.22 【JavaScript】

```
1     it('should been add to environment of running execution context', function(){
2       (function(){
3         eval('var bar = 100');
4         bar.should.be.exactly(100);
5       })();
6     });
```

如果是在 strict 模式下（调用代码或者 eval 代码是 strict 模式），生成新的 DeclarativeEnvironment，outer environment 指向上面定义的 lexicalEnvironment。将 lexicalEnvironment 和 variableEnvironment 设定为新生成的 DeclarativeEnvironment。

Mocha 8.23 【JavaScript】

```
1     it('should use new declarative environment in strict mode', function(){
2       (function(){
3         'use strict';
4         var bar = 200;
5         eval('var bar = 100');
6         bar.should.be.exactly(200);
7       })();
8     });
9
10    it('should use new declarative environment in strict mode 2', function(){
11      (function(){
12        var bar = 200;
13        eval('"use strict"; var bar = 100');
14        bar.should.be.exactly(200);
15      })();
16    });
```

function 创建：

调用的代码会传入 this 和参数。对于 strict code，this 就等于传入的 this。

否则，如果传入的 this 是 null 或者 undefined，this 设定为 global object。如果 this 是基本类型，就转换为 object。

新建 DeclarativeEnvironment，将 function 的`[[Scope]]`作为 outer Environment。将 execution context 的 lexicalEnvironment 和 variableEnvironment 设定为新生成的 DeclarativeEnvironment。

之后调用`[[Code]]`的代码。

Mocha 8.24 【JavaScript】this 是 object

```
1     it('this should be object', function(){
2       function foo(){
3         should(typeof this).be.exactly('object');
4       }
5       foo.call(1);
6     });
```

Mocha 8.25 【JavaScript】对于 undefined this 会被指定为 Global object

```
1     it('this should be global object if null or undefined been treat as this', function(){
2       function foo(){
3         this.should.be.exactly(global);
4       }
5
6       foo.call(null);
7       foo.call(undefined);
8     });
```

```
9
10    it('this should be global object if null or undefined been treat as this', function(){
11      function foo(){
12        this.should.be.exactly(global);
13      }
14
15      foo();
16    });
```

当标识符解析时，lexicalEnvironment 不会生成 this, 因为实现是中对于 Lexical Environment Record 的返回的 this 是 undefined。

8.3 函数

函数也是对象，可以使用对象的地方就可以使用函数。

8.3.1 函数申明和函数表达式

通过函数声明和函数表达是来创建 Function Object，

两者不同之处在于函数声明白创建的 function object 的[[Scope]]保存的是 running execution context 的 variableEnvironment，

本来我想写个例子，但是感觉实现不是这么实现的。下面可以访问到处于 lexicalEnvironment 链上的 bar

Mocha 8.26 【JavaScript】

```
1     it('does not work as expect', function(){
2       var foo = {bar: 200};
3       with(foo) {
4         eval('function func(){return bar;}');
5       }
6       func().should.be.exactly(200);
7     });
```

看上去，下面的代码可以说明，但是实际上 function declaration 是在执行代码之前就已经执行了。

Mocha 8.27 【JavaScript】

```
1     it('[[Scope]] should bind with variableEnvironment', function(){
2       // cannot prove
3       var foo = {bar: 100};
4
5       with(foo) {
6         function func() {
7           (function(){
8             bar;
9           }).should.throw();
10        }
11      }
12      func();
13    });
```

匿名函数表达式创建的 function object 的[[Scope]]保存的是 running execution context 的 lexicalEnvironment。

Mocha 8.28 【JavaScript】

```
1     it('[[Scope]] should bind with lexicalEnvironment', function(){
2
3       try {
4         throw {bar: 100};
5       } catch(e) {
6         var foo = function(){
7           return e;
8         }
9       }
10
11      should(foo()).be.eql({bar: 100});
12    });
```

带名字的函数表达式创建的一个 LexicalEnvironment，这个新创建 LexicalEnvironment 的 outer 将指向 running execution context 的 LexicalEnvironment，同时在新创建的 LexicalEnvironment 的 EnviromentRecord 上使用这个函数名字来绑定这个 function object，将这个新创建 LexicalEnvironment 保存在 function object 的`[[Scope]]`上。（我记得此处某个版本的 IE 不是这么实现的）。

Mocha 8.29 【JavaScript】

```
1    it('[[Scope]] should bind with a new DeclarativeEnvironment which outer is lexicalEnvironment
         of running execution context', function(){
2
3      try {
4        throw {bar: 100};
5      } catch(e) {
6        var foo = function e() {
7          return e;
8        }
9        e.should.eql({bar: 100});
10     }
11     should(foo()).be.exactly(foo);
12   });
```

8.3.2 函数对象的创建过程

1. 创建 ECMAScript Object；
2. `[[Class]]`设定为”function”
3. `[[Prototype]]`(可以通过`__proto__`来访问) 设定为 Function.prototype。
4. `[[Scope]]`设定为上面所描述的。
5. 定义 length 属性
6. 定义 prototype 属性，为 object 对象，prototype 上定义 constructor 属性

Mocha 8.30 【JavaScript】此处是函数表达式

```
1    it('[[Class]] is set to function', function(){
2      (typeof function(){}).should.be.exactly('function');
3    });
```

Mocha 8.31 【JavaScript】所以 func instanceof Function 会返回 true

```
1    it('__proto__ is set to Function.prototype', function(){
2      (function(){}).__proto__.should.be.exactly(Function.prototype);
3    });
```

Mocha 8.32 【JavaScript】

```
1    it('length is the number of parameters', function(){
2      (function(a, b, c, d, e, f){}).length.should.be.exactly(6);
3    });
```

Mocha 8.33 【JavaScript】

```
1    it('default prototype should be an object with constructor property', function(){
2      function foo(){}
3      foo.prototype.should.eql({constructor: foo});
4    });
```

8.3.3 函数作为构造函数

使用 new 关键字来调用，对于无参数的构造函数 Foo，new Foo() 和 new Foo 是相同的效果。

1. 创建对象；
2. `[[Class]]`设定为 object；

3. 取得构造函数的 prototype, 如果他不是 object，则取得 Object.prototype, 将他赋值给[[Prototype]]

4. 将创建的对象作为 this 来调用构造函数，如果返回值不是 object，返回创建的对象，如果返回的是 object，返回返回的对象。

Mocha 8.34 【JavaScript】作为构造函数生成对象的过程

```
1   it('steps for new operator', function(){
2     function Foo(){
3       this.bar = 200;
4     }
5
6     var newObj;
7     var foo = {};
8     foo.__proto__ = Foo.prototype;
9     var result = Foo.call(foo);
10    if(result === undefined || result === null || typeof result == 'number' || typeof result =='
          string' || typeof result == 'boolean') {
11      newObj = foo;
12    } else {
13      newObj = result;
14    }
```

Mocha 8.35 【JavaScript】prototype 是基本类型时，会使用 Object.prototype 代替

```
1   it('__proto__ is set to Object.prototyp if constructor.prototype is primitive or null',
        function(){
2     function Foo(){}
3     Foo.prototype = 1;
4
5     var foo = new Foo;
6     foo.__proto__.should.be.exactly(Object.prototype);
7   });
```

8.3.4 调用函数的过程

函数被调用，关注 **this** 的值

这里先会 evaluate 表达式，得到一个 ref,

当 ref 不是 reference type 的时候，this 被设定为 undefined（我的感觉比如匿名函数表达式，返回的就是一个 function 对象。）

当 ref 是 reference type 的时候，解析的 reference type 的的 base 是对象，这个对象就被当做 this。

当 reference type 的 base 是 Environment Record，则看看他是不是提供 this 值，如果提供就传入，否则传入 undefined

当 reference type 的 base 是 undefined 时，this 传入 undefined（在函数执行过程中，undefined 又会被 global object 代替）。

然后按上面提到的方式来创建 execution context。

说起来，this 有这样几种取值：

- 使用 expresson[expression] 或者 identifies.identifies 的格式来调用函数的时候，this 是前面那部分的值；
- 使用变量名的方式来调用的时候，this 就是指向 global object;
- func.call,func.apply 的方式调用的时候，第一个参数指定 this，如果这个时候传入的是 undefined 或者 null，则 this 是 global object，如果传入的是基本类型，则 this 是对应的包装类型
- 在 with block 中，使用变量名的方式来调用，如果这个变量是绑定在 with 传入的对象上时，this 就是 with 传入的对象。因为：此处的 LexicalEnvironment 被替换成 Object Environment Record，绑定了 foo。bar 得到的 reference type 是 base：objEnv(foo), name:bar, strict:false。objEnv 提供的 this 是 foo，所以调用的时候，this 被绑定为 foo

Mocha 8.36 【JavaScript】

```
1   describe("#this", function(){
2     it('dot expression or square bracket', function(){
```

```
    var foo = {
      bar: function(){ return this;}
    }
    foo.bar().should.be.exactly(foo);
    foo['bar']().should.be.exactly(foo);
  });

  it('variable', function(){
    var foo = function(){return this;};
    foo().should.be.exactly(global);
  });

  it('variable in with', function(){
    var foo = {
      bar: function(){return this;}
    }
    with(foo){
      bar().should.be.exactly(foo);
    }
  });

  it('call, apply', function(){
    function foo(){return this;}
    var bar = {};

    foo.call(bar).should.be.exactly(bar);
    foo.apply(bar).should.be.exactly(bar);

    var res = foo.call(1);
    should(res).be.Object;
    (res == 1).should.be.true;

    foo.call(undefined).should.be.exactly(global);
  });
});
```

函数代码执行

下面的步骤是将标示符绑定到当前的 execution context 的 VariableEnvironment 上。

1. 先创建 arguments 对象, 同时在 variableEnvironment 上定义参数列表中的参数，参数列表中如果有重复的变量名，后面的会覆盖前面的值;
2. 绑定函数申明，后声明的覆盖前面申明的函数;
3. 如果没有定义 arguments 的函数，绑定第一步创建的 arguments 对象;
4. 绑定不存在的变量申明，值为 undefined。

Mocha 8.37 【JavaScript】

```
describe('#arguments', function(){
  it('last should override first if the has same name', function(){
    function foo(a, b, c, a){return a;}
    foo(1,2,3,4).should.be.exactly(4);
  });

  it('arguments should be replace by function', function(){
    function foo(){
      arguments.should.be.type('function');
      function arguments(){}
      var arguments = {};
      arguments.should.eql({});
    }
    foo();
  });

  it('arguments should exist', function(){
```

```
18      function foo(){
19        arguments.should.be.arguments;
20        var arguments= {};
21        arguments.should.eql({});
22      }
23      foo();
24    });
25
26    it('should be', function(){
27      function foo(){
28
29        bar.should.be.type('function');
30
31        var bar = 10;
32
33        bar.should.be.exactly(10);
34
35        function bar(){}
36
37        bar.should.be.exactly(10);
38
39      }
40
41      foo();
42    });
43
44    it('should be', function(){
45      function foo() {
46        should(bar).be.exactly(undefined);
47        var bar = 10;
48        bar.should.be.exactly(10);
49      }
50
51      foo();
52    });
53  });
```

变量的赋值

通过 GetIdentifierReference 可以看到，找不到变量定义，reference type 的 base 就是 undefined，这个时候就会在 global object 上定义一个对应的 property。

8.4 ES6

Part II

CSS

Part III

Ruby

Part IV

Java

Part V

C Sharp

Part VI

Function Programming

Part VII

Algorithm

Part VIII

Design Pattern

Part IX

Web

Part X

Linux

Part XI

Protocol