



Lecture 07. Intermediate JavaScript

Modern Web Programming

(<http://my.ss.sysu.edu.cn/wiki/display/WEB/> supported by Deep Focus)

School of Data and Computer Science, Sun Yat-sen University

Secrets of JavaScript

CLOSURE **PROTOTYPE**

TOP SECRET

{JAVASCRIPT}

Part I Theories

LAW & LOGIC

Hoisting

```
> var foo = "bar";
(function () {      IIFE Immediately-Invoked Function Expression
    console.log("Original value was " + foo);

    var foo = "foobar";

    console.log("New value is " + foo);
})();
```



```
2015-11-15 09:26:27.289 Original value was undefined
2015-11-15 09:26:27.289 New value is foobar
```

Hoisting

Variable hoisting

```
> var foo = "bar";
  (function () {
    var foo;

    console.log("Original value was " + foo);
    // Outputs: "Original value was undefined"

    foo = "foobar";

    console.log("New value is " + foo);
    // Outputs: "New value is foobar"
  })();
  console.log('foo: ' + foo);
```

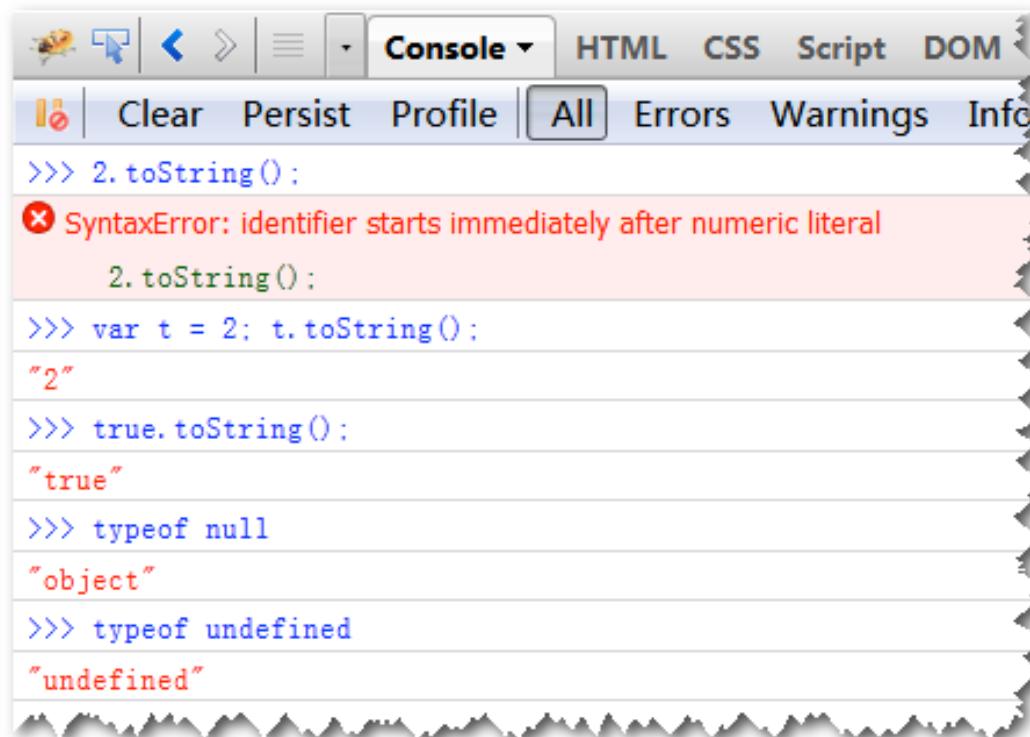
2015-11-15 09:32:11.691 Original value was undefined

2015-11-15 09:32:11.691 New value is foobar

2015-11-15 09:32:11.691 foo: bar

Everything in JS is an Object

- What is an object
- What is OOP



The screenshot shows a browser's developer tools console tab labeled "Console". The tabs above it include "HTML", "CSS", "Script", and "DOM". Below the tabs are buttons for "Clear", "Persist", and "Profile", with "All" selected. The console output shows the following interactions:

```
>>> 2.toString();
✖ SyntaxError: identifier starts immediately after numeric literal
    2.toString();
>>> var t = 2; t.toString();
"2"
>>> true.toString();
"true"
>>> typeof null
"object"
>>> typeof undefined
"undefined"
```

- 只有数字字面表达 (number literal) 和undefined不是对象

Everything in JS is an Object

- JavaScript语言中对象就是一个容器，容器包含了一系列属性（**properties**）。
- 每个属性都是一组名值对，属性名可以是任意字符串，包括空字符串，属性值可以是JavaScript中除undefined任何外合法的值。
- 当属性值为函数时，这个属性也称为对象的方法。
- JavaScript对象的最大特点就是其动态性（动态对象，**dynamic object**），一个对象可以在创建之后，动态地增加和删除属性和方法。

Object literal

源代码 9-1 对象字面表达（Object Literal）示例

```
var member = {  
    name: '张三',  
    age: 23,  
    'goto': 'United States',  
    say: function () {  
        return this.name + '前往' + this['goto'];  
    }  
};  
  
alert(member.age);          // 23  
alert(member.say());        // 张三前往United States
```

Object literal

源代码 9-2 属性读取与更新示例

```
var member = {
    age: 23,
    'goto': 'United States',
};

alert(member.age);          // 23
alert(member['age']);       // 23
alert(member['goto']);      // United States
alert(member.goto);         // 语法错误！（某些JavaScript引擎能够执行）
```

源代码 9-3 安全读取和使用属性示例

```
var member = {
    age: 23,
    'goto': 'United States',
};

var name = member.name;           // undefined
var name = member.name || '未知'; // '未知'
member.address.state            // 抛出TypeError异常
member.address && member.address.state // undefined
```

Use of properties

源代码 9-4 更新和移除属性示例

```
var member = {  
    age: 23,  
    'goto': 'United States',  
};  
  
member.age = 34;  
alert(member.age);           // 34  
member.name = '李四';  
alert(member.name);         // '李四'  
delete member.name;  
alert(member.name);         // undefined
```



JavaScript 动态对象 (dynamic objects): 传统静态类型、强类型语言，如：C++、Java 等等，对象的类型固定，创建之后就不能改变增加或者删除其属性和方法。JavaScript 和很多动态类型、弱类型语言，如：ruby, python 等等，对象的类型不固定，都可以在对象创建后，灵活地增加和删除属性与方法。

Array literal

- Array is a special kind of object

源代码 9-6 数组字面表达示例

```
var a = ['a', 'b', 'c', 0, {name:'张三'}];
alert(typeof a); // object
alert(a.length); // 5
alert(a[4].name); // '张三'
```

JavaScript 的对象在使用中传递的都是引用，没有传值，即复制传递的方式。

源代码 9-5 更新和移除属性示例

```
var peter = {nickname: 'rabbit'};
var littlePeter = peter;
peter.nickname = 'bear';
alert(littlePeter.nickname); // 'bear'
```

Function is also an Object

源代码 9-7 函数对象示例

```
var func = function(otherFunc) {
    alert('func');
    otherFunc();
    return otherFunc;
};

var func2 = function() {
    alert('func2');
};

func.method = function() {
    alert('method of func');
};

var obj = {
    myFunc : func
};

var arr = [func, func2];

func.method();          // method of func
obj.myFunc(func2);    // func func2
arr[0](arr[1])();     // func func2 func2
```

Define a function

```
> function a(){}
<==> var a = function a(){}
var b = function(){}
var c = function d(){}
< undefined
> a.name
< "a"
> b.name
< ""
> c.name
< "d"
> d.name
✖ ► Uncaught ReferenceError: d is not defined(...)
```

VM1142:2

Function hoisting

```
> var a = function(){ // a的scope为global
    b = function c(){};
    function d(){}
}
console.log(typeof a, typeof b, typeof c, typeof d, typeof e);
a();
console.log(typeof a, typeof b, typeof c, typeof d, typeof e);
function e(){}
```

function hoisting

```
2015-11-15 09:57:03.761 function undefined undefined undefined function
2015-11-15 09:57:03.762 function function undefined undefined function
```

Function hoisting

```
> var a = function(){ // a的scope为global
    b = function c(){}; // b的scope为global, c的scope为a
    function d(){}; // d的scope为a
};

console.log(typeof a, typeof b, typeof c, typeof d, typeof e);

a();

console.log(typeof a, typeof b, typeof c, typeof d, typeof e);
var f = function e(){};
```

function hoisting ??

```
2015-11-15 10:03:07.127 function undefined undefined undefined undefined
2015-11-15 10:03:07.128 function function undefined undefined undefined
```

function hoisting is **ONLY** for **declaration**, NOT for **expression**!

Scope

源代码 9-9 JavaScript 嵌套作用域示例

```
function outer() {
    var outerName = 'outer name';
    var inner = function() {
        alert(outerName); // outer name
        var innerName = 'inner name';
    };
    inner();
    alert(innerName); // ReferenceError: innerName is not defined
}

outer();
```

- var 局部；无 var 全局
- 局部为function block scope

Duration

源代码 9-11 JavaScript 词法作用域（lexical scope）示例

```
function outer() {
    var secret = 'secret';
    inner = function() {
        alert(secret);
    };
    inner();
}
outer(); // secret
inner(); // secret
```

词法作用域是 JavaScript 构造闭包（closure）的基础，参考 f)闭包。

arguments, caller, callee

源代码 9-12 arguments 参数示例

```
var sum = function () {
  var i, sum = 0;
  for (i = 0; i < arguments.length
    sum += arguments[i];
  }
  return sum;
};
```

```
> function add(a, b){};
function addThree(a, b, c){};
console.log(add.length);
console.log(addThree.length);
2015-11-15 10:35:54.366 2
2015-11-15 10:35:54.366 3
```

```
> function a(){
  console.log("callee:", arguments.callee);
  console.log("caller:", arguments.callee.caller);
}
function b(){a()};
```

```
< undefined
```

```
> b();
```

```
2015-11-15 14:45:38.539 callee: function a(){
  console.log("callee:", arguments.callee);
  console.log("caller:", arguments.callee.caller);
}
```

```
2015-11-15 14:45:38.539 caller: function b(){a();}
```

4 kinds of function invoking

- 普通函数
- 方法(method)调用
- 构造子
- 应用(apply、call)调用

Constructor

源代码 9-13 函数构造子示例

```
var Foo = function () {
  this.name = 'foo';
}

var result = Foo();          // 普通函数
alert(result);              // undefined
alert(name);                // foo
var result = new Foo();       // 构造子
alert(result);              // Object
alert(result.name);         // foo
```

Constructor

```
> var Foo = function(){ this.name = 'foo' };
  bar = new Foo;
< Foo {name: "foo"}
> bar.constructor === Foo
< true
> bar instanceof Foo
< true
```

call , apply

源代码 9-17 函数应用 (apply、call) 模式调用示例

```
var sayHello = function(message, to) {
    alert(this.name + ' says ' + message + ' to ' + to);
};

var peter = {name: 'peter'};
var name = 'global';
sayHello.apply(this, ['hello', 'Marry']); //global says hello to Marry
sayHello.apply(peter, ['hello', 'Marry']); // peter says hello to Marry
sayHello.call(this, 'hello', 'Marry'); // global says hello to Marry
sayHello.call(peter, 'hello', 'Marry'); // peter says hello to Marry
```

4 kinds of function invoking

表 9-1 4 种调用方式的对比

调用模式	this	无 return 时的返回值
函数模式	顶层对象（在浏览器中执行时为 window）	undefined
方法模式	当前对象（方法从属的对象，即成员操作符 “.” 的左侧）	undefined
构造子模式	正在构造的对象	this（构造好的对象）
应用模式	第一个参数	undefined



Closure

Closure

- 闭包(**closure**)指函数和函数所能访问的函数体**外部局部**变量构成的组合
- 闭包中的函数称为**闭包函数**, 闭包函数能够访问的函数体外部局部变量称为**闭包变量**
- JavaScript 的**scope**和**duration**使得闭包，自然、容易、强大

源代码 9-18 闭包 (closure) 示例

```
var counter = function () { IIFE
    var amount = 0;
    return function () {
        return amount++;
    };
}();

alert(counter()); // 0
alert(counter()); // 1
alert(counter()); // 2
```

A simple closure

```
var f = function(x) {  
    var m = function(y) {  
        return x * y;  
    }  
    return m;  
}  
  
var instance = f(z);
```

Global Scope

A simple closure

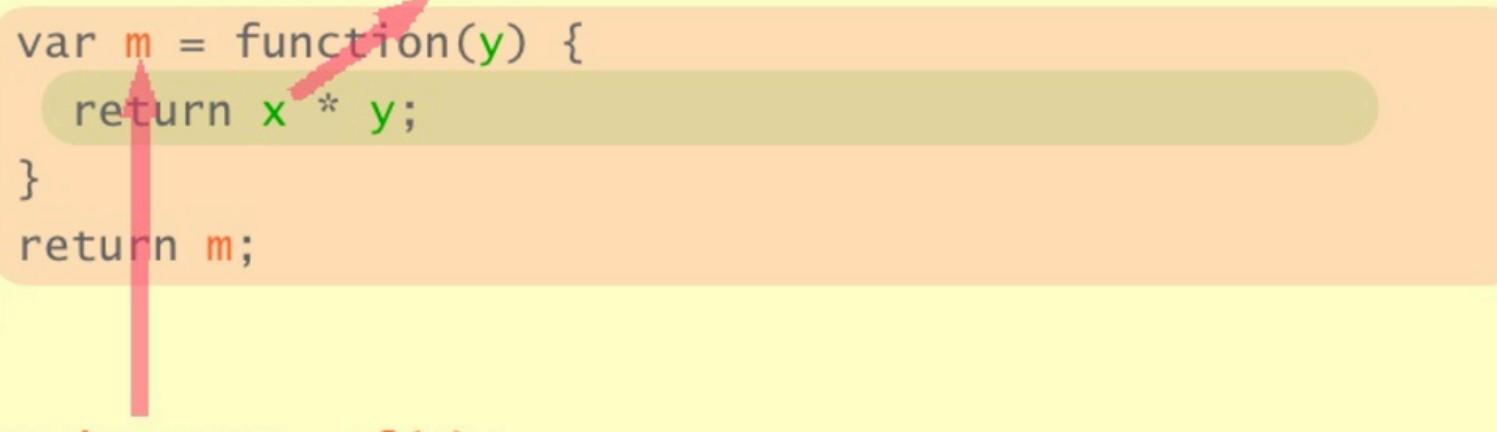
```
var f = function(x) {  
    var m = function(y) {  
        return x * y;  
    }  
    return m;  
}  
  
var instance = f(z);
```

Global Scope

A simple closure

```
var f = function(x) {  
    var m = function(y) {  
        return x * y;  
    }  
    return m;  
}  
  
var instance = f(z);
```

Global Scope



A simple closure

```
var f = function(x) {                                Global Scope
    var m = function(y) {
        return x * y;
    }
    return m;
}
var myDouble = f(2);
var myTreble = f(3);

alert (myDouble(10)) // 20
alert (myTreble(10)) // 30
```

```
.....  
<ol>  
    <li>第一项</li>  
    <li>第二项</li>  
    <li>第三项</li>  
    <li>第四项</li>  
</ol>  
.....
```

```
window.onload = function () {  
    var lis = document.getElementsByTagName ('li');  
    for (var i = 0; i < lis.length; i++) {  
        lis[i].onclick = function () {  
            alert(i);  
        }  
    }  
}
```

this problem

源代码 9-15 this 丢失问题示例

```
var peter = {name: 'peter'};  
var name = 'global';  
var sayHello = function() {  
    var helper = function() {  
        alert(this.name + ' says hello');  
    };  
    helper();  
};  
peter.greeting = sayHello;  
  
peter.greeting(); // global says hello (应该是peter says hello)
```

that for this

源代码 9-16 使用 that 模式修复 this 丢失问题示例

```
var peter = {name: 'peter'};  
var name = 'global';  
var sayHello = function() {  
    var that = this;  
    var helper = function() {  
        alert(that.name + ' says hello');  
    };  
    helper();  
};  
peter.greeting = sayHello;  
  
peter.greeting(); // peter says hello
```



PROTOTYPE

Prototype

```
a = {};
console.log(typeof a.b);
console.log(typeof a.toString);
console.log(typeof a.__proto__);
```

```
2015-11-15 10:26:33.002 undefined
```

```
2015-11-15 10:26:33.002 function
```

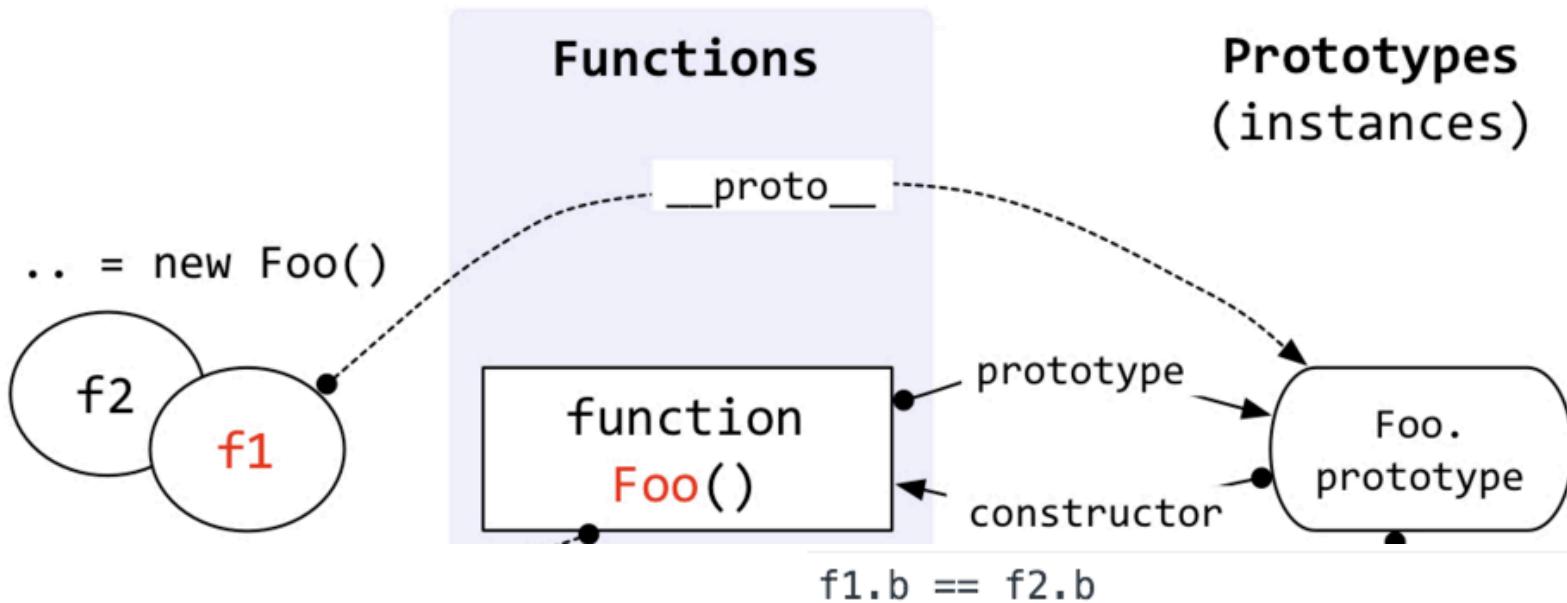
where's it from?

```
2015-11-15 10:26:33.003 object
```

what the hell it is?

Prototype

JavaScript Object Layout [Hrush Jain/mollypages.org]



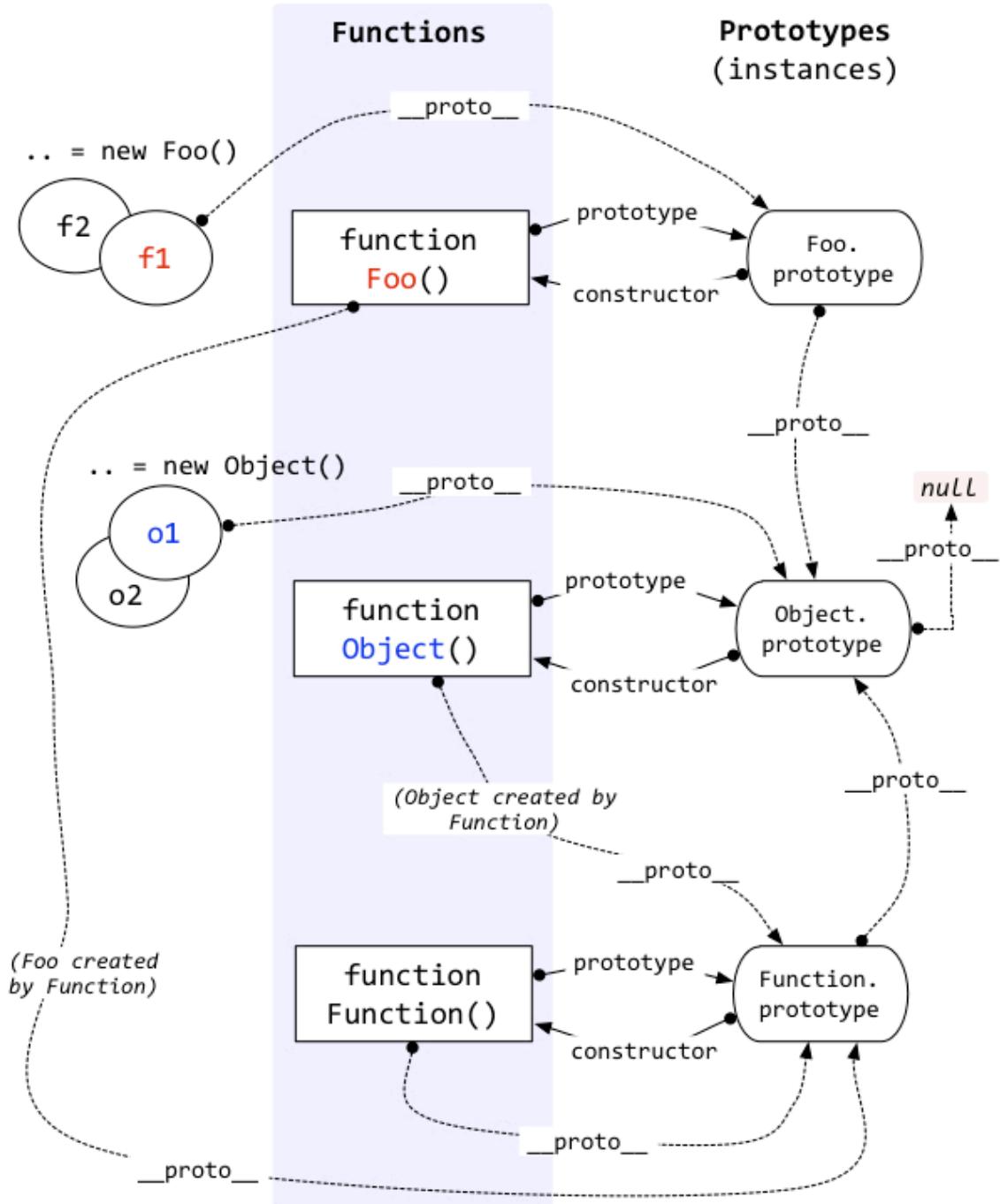
```
> function Foo(name){this.name = name}
  Foo.prototype.b = function(){const f1.__proto__ == f1.constructor.prototype
f1 = new Foo('first');
f2 = new Foo('second');
f1.b();
f2.b();
```

2015-11-15 11:01:52.256 name: f: f1.hasOwnProperty('b')

2015-11-15 11:01:52.256 name: se false

Prototype

- `__proto__`, 缺属性去哪儿找
- `__proto__` 指向构造子的prototype
- 构造子的prototype用`constructor`反指构造子
- 构造子的prototype也是object, 也有自己的构造子
- 形成了`__proto__`链条



Part II Applications

Part II Applications

Information hiding

- private/secret by closure

```
> var count = function(){
  var secretAmount = 0;
  count = function(n){ if (!count.isEnough()) secretAmount += n;
};

count.isEnough = function(){ return secretAmount >= 2;};
count.reset = function(){ return secretAmount = 0;};
return count;
}();
> count.isEnough()
<- false
> count(1)
<- undefined
> count.isEnough()
<- false
> count(1)
<- undefined
> count.isEnough()
<- true
> secretAmount
✖ ▶ Uncaught ReferenceError: secretAmount is not defined...
> count.secretAmount
<- undefined
```

Code reuse

```
> var tom = {name: 'tom', greet: function(){console.log("I am", this.name)}};

var mike = {name: 'mike', greet: tom.greet}

tom.greet();
mike.greet();

2015-11-16 16:25:08.496 I am tom
2015-11-16 16:25:08.497 I am mike
```

```
> [].forEach
< function forEach() { [native code] }

> 'abc'.forEach
< undefined

> Array.prototype.forEach.call('abc', function(c){console.log(c);});

2015-11-15 12:06:48.572 a
2015-11-15 12:06:48.572 b
2015-11-15 12:06:48.572 c
```

Code reuse

```
> function Animal(name){this.name = name;};
Animal.prototype.sleep = function(){console.log(this.name + ' is sleeping.');}

function Dog(name, age){this.age = age; Animal.call(this, name);}
Dog.prototype = new Animal;
Dog.prototype.constructor = Dog;

var ace = new Dog('Ace', 2);
ace.sleep();
console.log("name: ", ace.name, " age: ", ace.age);

var bob = new Dog('Bob', 5);
bob.sleep();
console.log("name: ", bob.name, " age: ", bob.age);
```

2015-11-15 11:57:08.014 Ace is sleeping.

2015-11-15 11:57:08.014 name: Ace age: 2

2015-11-15 11:57:08.014 Bob is sleeping.

2015-11-15 11:57:08.014 name: Bob age: 5

Functional Programming

源代码 9-22 Memoization 示例

```
----- 原始fibonacci -----  
var fibonacci = function(n) {  
    return n < 2 ? n : fibonacci(n - 1) + fibonacci(n - 2);  
};  
  
for (var i = 0; i <= 10; i += 1) {  
    document.writeln('// ' + i + ': ' + fibonacci(i));  
} // fibonacci被调用452次
```

Functional Programming

```
----- Memoization fibonacci -----
var fibonacci = function() {
  var memo = [0, 1];
  var fib = function(n) {
    var result = memo[n];
    if (typeof result !== 'number') {
      result = fib(n - 1) + fib(n - 2);
      memo[n] = result;
    }
    return result;
  };
  return fib;
}();

for (var i = 0; i <= 10; i += 1) {
  document.writeln('// ' + i + ': ' + fibonacci(i));
} // fibonacci被调用29次
```

Functional Programming

源代码 9-23 Memoization 示例

```
----- memoizer -----
var memoizer = function(memo, fundamental) {
  var shell = function(n) {
    var result = memo[n];
    if (typeof result !== 'number') {
      result = fundamental(shell, n);
      memo[n] = result;
    }
    return result;
  };
  return shell;
};
```

```
----- memoizer fibonacci函数 -----
var fibonacci = memoizer([0, 1], function(shell, n) {
  return shell(n - 1) + shell(n - 2);
});
```

```
----- memoizer阶乘函数 -----
var factorial = memoizer([1, 1], function(shell, n) {
  return n * shell(n - 1);
});
```

Thank you!

