

函数高级

函数的 prototype 属性

1. * 每个函数都有一个 prototype 属性, 它默认指向一个 Object 空对象(即称为: 原型对象)
 - * 原型对象中有一个属性 constructor, 它指向函数对象
2. 给原型对象添加属性(一般都是方法)
 - * 作用: 函数的所有实例对象自动拥有原型中的属性(方法)

`Fn.prototype.test = function () { // 给原型对象添加方法 }`

显示原型/隐式原型

- 每个函数 function 都有一个 prototype, 即**显式原型**
(就是自己定义的方法)
- 每个实例对象都有一个 `__proto__`, 可称为**隐式原型**
(就是它的构造函数里面的的方法, 他爹的方法)
【比如: Data () 是 Object () new 出来的, Data () 里面获得时分秒的方法叫做自己的显示原型, 但是它也可以用 Object () 里面的 toString()方法】
- 对应构造函数的显式原型的值=对象的隐式原型的值
它爹的显示原型(它爹的方法) = 它的隐式原型(是他爹的方法)

总结:

- * 函数的 prototype 属性: 在定义函数时自动添加的, 默认值是一个空 Object 对象, 就是自己定义的属性
- * 对象的 `__proto__` 属性: 创建对象时自动添加的, 默认值为构造函数的 prototype 属性值, 就是从构造函数 (他爹) 哪里自动继承的属性
- * 程序员能直接操作显式原型, 但不能直接操作隐式原型(ES6 之前)

原型链(图解)

别名: 隐式原型链

作用: 查找对象的属性(方法)

how

- * 访问一个对象的属性时,
 - * 先在自身属性中查找, 找到返回
 - * 如果没有, 再沿着 `__proto__` 这条链向上查找, 找到返回
 - * 如果最终没找到, 返回 `undefined`

就是在自己里面找, 找不到去它爹哪里找, 然后再到它爷爷哪里找

1. 读取对象的属性值时: 会自动到原型链中查找, 如上。
2. 设置对象的属性值时: 不会查找原型链, 如果当前对象中没有此属性, 直接添加此属性并设置其值

eg: `p1.sex = '女' // 给对象添加属性不会查看原型链`

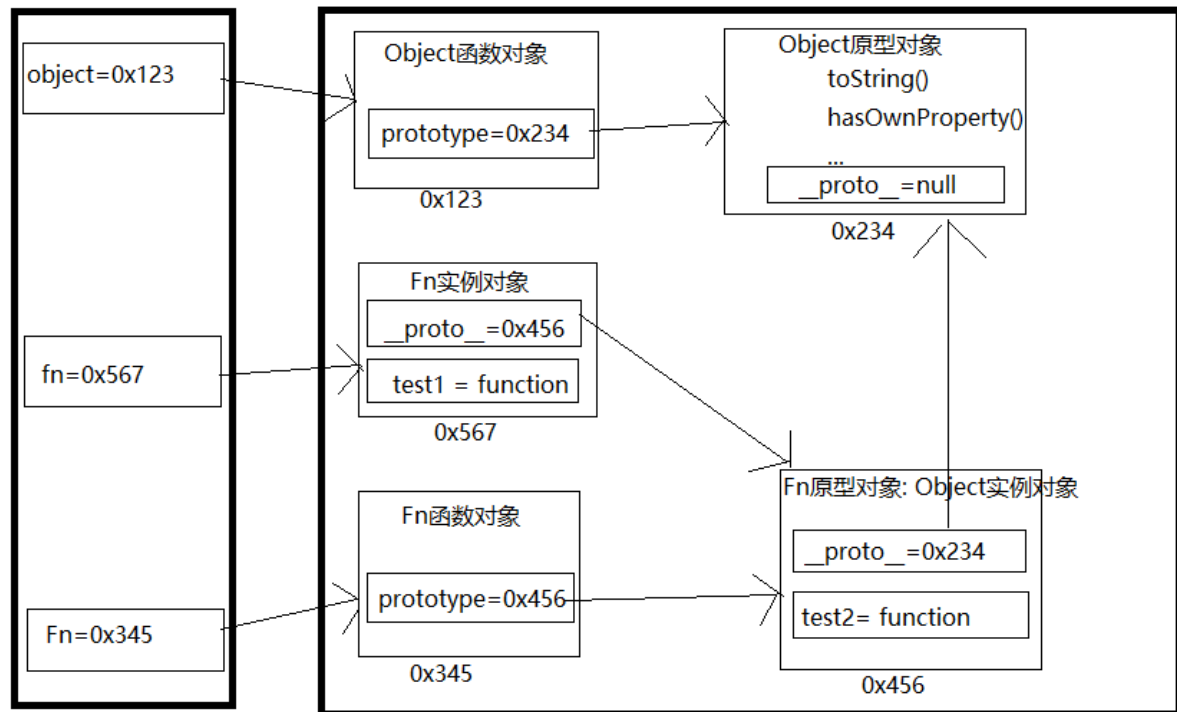
3. 方法一般定义在原型中, 属性一般通过构造函数定义在对象本身上

eg: `var p2 = new Person('Bob', 23)`

```

function Fn() {
  this.test1 = function () {
    console.log('test1()')
  }
}
Fn.prototype.test2 = function () {
  console.log('test2()')
}
var fn = new Fn()
fn.test1()
fn.test2()
console.log(fn.toString())
fn.test3()

```

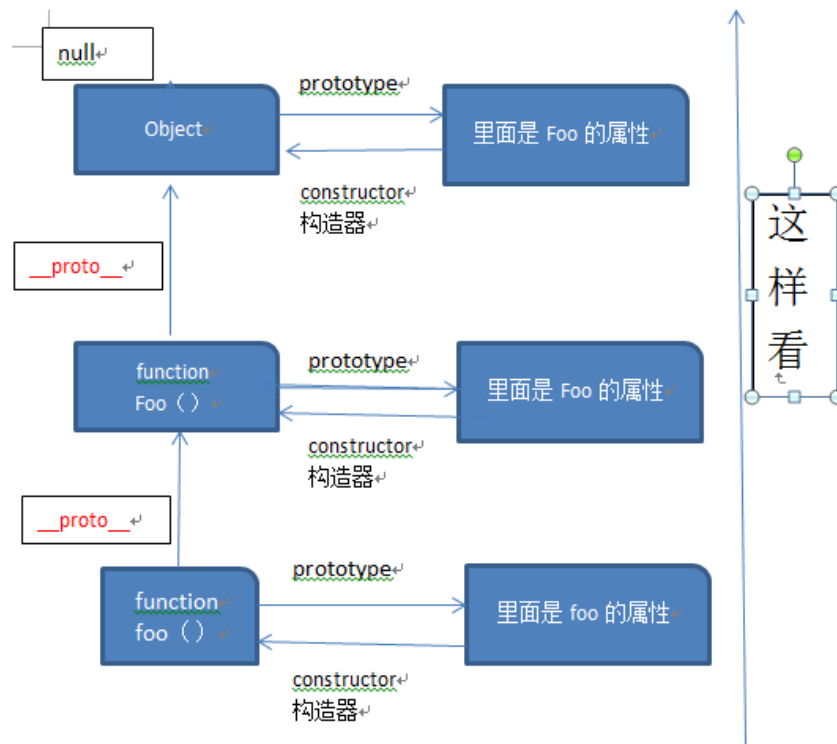


重: Object 的隐式原型是 null, 在原型是最大

任何函数都是 Function 函数的对象

Function 是通过 new 自己产生的实例 var Function = new Function

✱ Function 不是 Object, 但是相等, 用 instanceof 判断是 true



1. 构造器就是通过属性给构造函数添加原型方法 `Person.prototype.setName = function (name) { this.name = name; };`
2. 构造函数的实例对象自动拥有构造函数原型对象的属性(方法)。

探索 instanceof

* 表达式: instanceof (A instanceof B)

* 如果 B 函数的显式原型对象在 A 对象的原型链上, 返回 true, 否则返回 false, 就是看 A 是不是 B 的实例, 是不是他的后代

族谱: Object--->Foo(被 Object,new 出来的)--->foo(Foo, new 出来的)

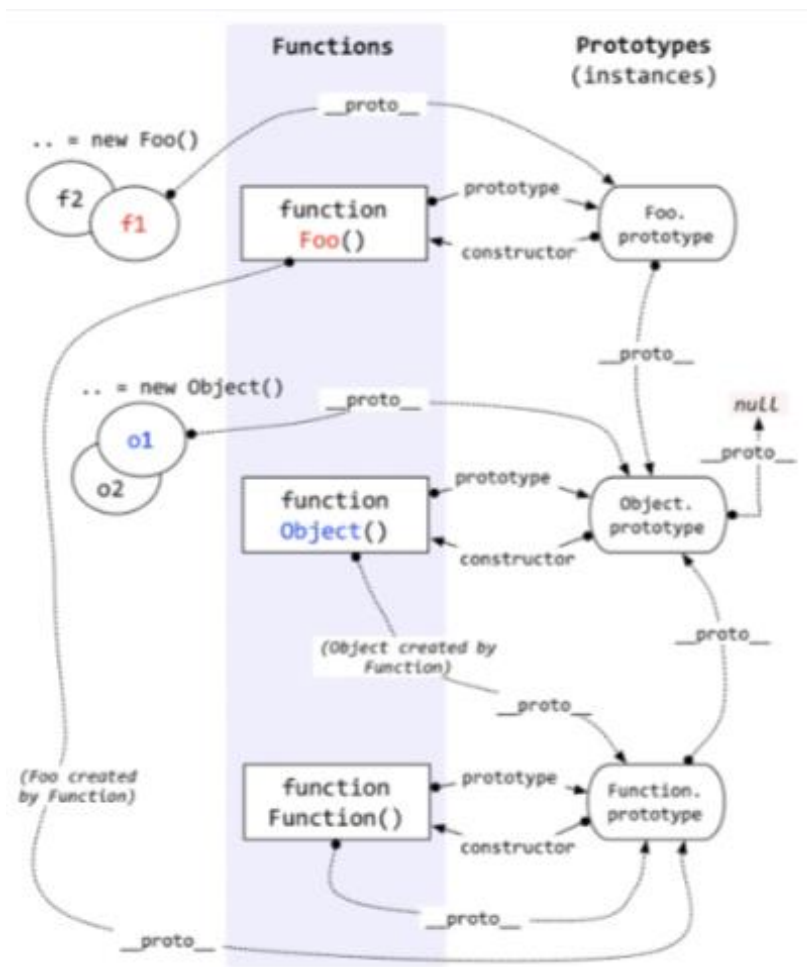
案例 1

```
function Foo() { }  
var f1 = new Foo();  
console.log(f1 instanceof Foo);//true  
console.log(f1 instanceof Object);//true
```

案例 2

- console.log(Object instanceof Function);
//任何函数都是 Function 函数的对象
- console.log(Object instanceof Object);
//true
- console.log(Function instanceof Function);
// Function 是通过 new 自己产生的实例
- console.log(Function instanceof Object);
//true, Function 不是 Object, 但是相等
function Foo() {}
console.log(Object instanceof Foo);

instanceof 图解



试题一：

```
var A = function() {
```

```
}
```

```
A.prototype.n = 1
```

之前在显式原型中是n=1，但是没有m的值
在后面给显式原型添加一个数组，打断之前的，新增
一个区域放后面的值，在把这个地址重新给A

```
var b = new A()
```

```
A.prototype = {
```

```
  n: 2,
```

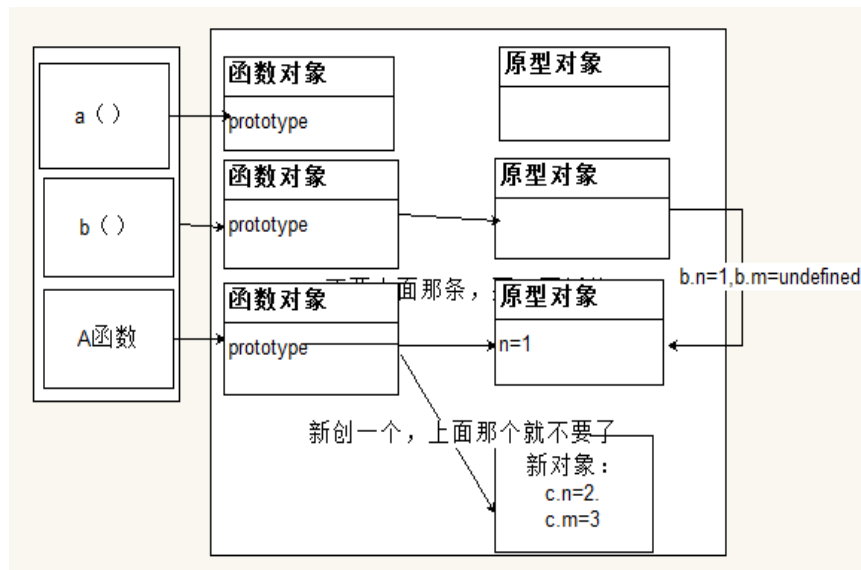
```
  m: 3
```

```
}
```

```
var c = new A()
```

```
console.log(b.n, b.m, c.n, c.m)
```

1, undefined, 2, 3



```
var F = function(){};
```

```
Object.prototype.a = function(){
```

```
  console.log('a()')
```

```
};
```

```
Function.prototype.b = function(){
```

```
  console.log('b()')
```

```
};
```

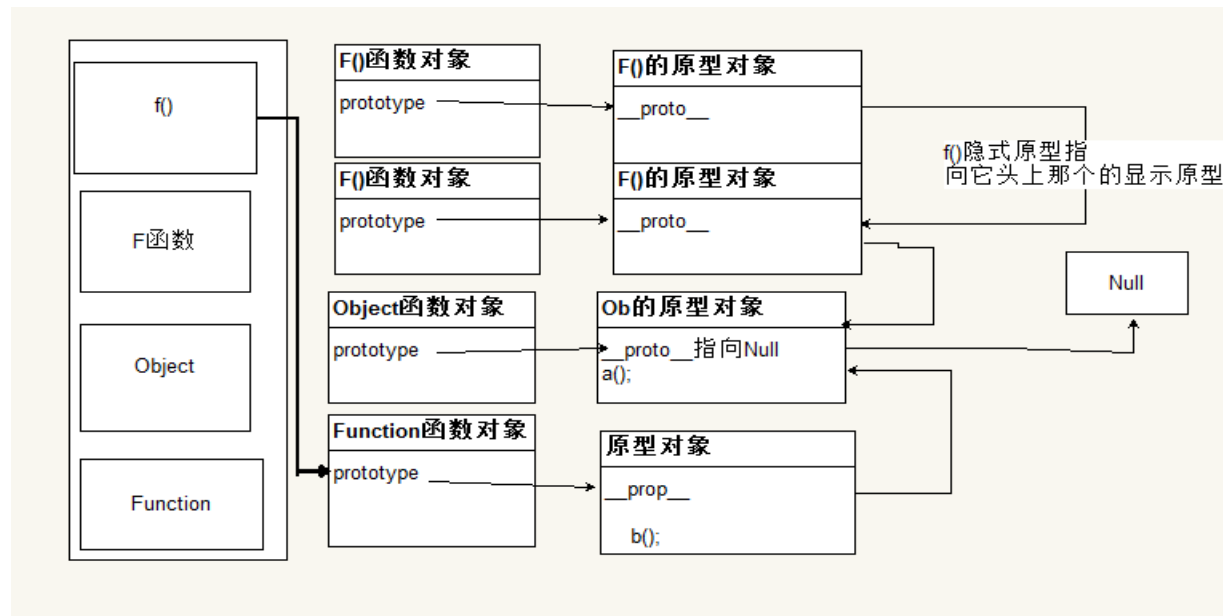
```
var f = new F();
```

```
f.a()
```

```
F.a()
```

```
F.b()
```

```
f.b()
```



f.a()—>f()函数对象--->隐式→F()的显示—>没有—>隐找 obj 的显--->a ();

F.a();--->F()的显示—>没有—>隐找 obj 的显--->a ();

f.b()—>每一个函数都是 Function 出来的-->Function—>prototype—>b());

F.b()→隐找 obj 的显→__proto__--->null 报错

执行上下文（产生次数： $n+1$ ， n 是函数， 1 是全局）

变量函数提升是因为 js 内核里的预定义

变量声明提升

- * 通过 `var` 定义(声明)的变量，在定义语句之前就可以访问到

- * 值: `undefined`

```
var a
```

```
console.log(a)
```

```
var a = 1;
```

输出 `undefined`

注意：每一个变量有没有值都是赋给 `undefined`，后面看是运行先还是赋值先，赋值就可以输出，不赋值就是 `undefined`

函数声明提升

- * 通过 `function` 声明的函数，在之前就可以直接调用

- * 值: 函数定义(对象)

情况 1: `var a`

```
console.log('fn()')
```

```
function fn() {}
```

情况 2:

```
var a
```

```
console.log('fn()')
```

```
var a = function fn() {}
```

分析：函数是整体的提升，全部预处理，预处理之后，

后面无论怎么运行赋值都可以输出但是如果像第二种方式去定义函数就会变成变量提升，就要遵守变量提升的规律

代码分类：

全局代码（在执行全局代码前将 `window` 确定为全局执行上下文）

每次 **var** 一个变量的情况如下：

`var` 定义的全局变量==>`undefined`，添加为 `window` 的属性

- * `function` 声明的全局函数==>赋值(`fun`)，添加为 `window` 的方法

- * `this`==>赋值(`window`)

开始执行全局代码：从 `window` 查找对应属性变量

函数代码 函数执行上下文(不是真实存在的对象)

每次 **function** 一个函数时都会产生如下：

- * 在调用函数，准备执行函数体之前，创建对应的函数执行上下文对象

- * 对局部数据进行预处理

- * 用 `var` 定义的局部变量 ==>`undefined`

- * 使用 `function` 声明的函数 ===>`function`

- * `this` ===> 调用函数的对象，如果没有指定就是 `window`

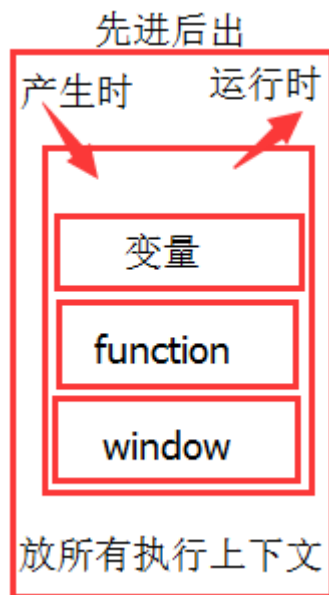
- * 形参变量 ===>对应实参值

- * `arguments` ===>实参列表的伪数组

- * 开始执行函数体代码

执行上下文栈

1. 在全局代码执行前, JS 引擎就会创建一个栈来存储管理所有的执行上下文对象
2. 在函数执行上下文创建后, 将其添加到栈中(压栈)
3. 在全局执行上下文(window)确定后, 将其添加到栈中(压栈)
4. 在当前函数执行完后, 将栈顶的对象移除(出栈)
5. 当所有的代码执行完后, 栈中只剩下 window



先预处理函数, 后预处理变量, 如果已经存在就会被忽略

例 1

```
function a() {}  
var a;  
console.log(typeof a)
```

先放fun a
在放var a 名字一样
var a去掉, 直接忽略
所有a的类型是function

例 2:

```
if (!(b in window)) {  
  var b = 1;  
}  
console.log(b)
```

b上升变成全局, in是只要在原型链上就能找到, 所有找到了, 取反就不是, 所以不进入if中, 直接undefined

例 3:

```
var c = 1  
function c(c) {  
  console.log(c)  
  var c = 3  
}  
c(2)
```

先是函数上升, fun c
然后是var c不要就忽略
执行就在fun c=1, 所以到c(2)
时这个数是1无法调用

作用域:

理解

- * 就是一块"地盘", 一个代码段所在的区域
- * 它是静态的(相对于上下文对象), 在编写代码时就确定了

分类

- * 全局作用域
- * 函数作用域
- * 没有块作用域(ES6 有了) (if 没有作用域)

作用

- * 隔离变量, 不同作用域下同名变量不会有冲突

理解

1. 函数是从里面函数到外面的函数

- * 多个上下级关系的作用域形成的链, 它的方向是从下向上的(从内到外)
- * 查找变量时就是沿着作用域链来查找的

2. 查找一个变量的查找规则

- * 在当前作用域下的执行上下文中查找对应的属性, 如果有直接返回, 否则进入 2
- * 在上一级作用域的执行上下文中查找对应的属性, 如果有直接返回, 否则进入 3
- * 再次执行 2 的相同操作, 直到全局作用域, 如果还找不到就抛出找不到的异常

区别 1

- * 全局作用域之外, 每个函数都会创建自己的作用域, 作用域在函数定义时就已经确定了。而不是在函数调用时
- * 全局执行上下文环境是在全局作用域确定之后, js 代码马上执行之前创建
- * 函数执行上下文环境是在调用函数时, 函数体代码执行之前创建

区别 2

- * 作用域是静态的, 只要函数定义好了就一直存在, 且不会再变化
- * 上下文环境是动态的, 调用函数时创建, 函数调用结束时上下文环境就会被释放

联系

- * 上下文环境(对象)是从属于所在的作用域
- * 全局上下文环境==>全局作用域
- * 函数上下文环境==>对应的函数使用域

例 1

```
var x = 10;
function fn() {
  console.log(x);
}
function show(f) {
  var x = 20;
  f();
}
show(fn);
```

show(f)---show(fn)
var x = 20;
fn()-->console.log(x)
此时x是fn()里面的x--->x=10

例 2

```
var obj = {
  fn2: function () {
    console.log(fn2)
  }
}
obj.fn2()
```

调用fn2, fn2的作用是打印fn2函数, 但是fn2只是属性,
不是函数名, 所以报错说fn2不是函数

闭包

1. 如何产生闭包?

- * 当一个嵌套的内部函数引用了嵌套的外部函数的变量(函数) 时, 就产生了闭包

2. 闭包到底是什么?

- * 使用 chrome 调试查看
- * 理解一: 闭包是嵌套的内部函数(绝大部分人)
- * 理解二: 包含被引用变量(函数)的对象(极少数人)
- * 注意: 闭包存在于嵌套的内部函数中

3. 产生闭包的条件?

- * 函数嵌套
- * 内部函数引用了外部函数的局部变量
- * 执行外部函数

(1) 首先要函数嵌套 fn(){ fn1(){} }

(2) 然后里面的函数要调用外面的变量

```
fn(){ a=1 fn1(){console(a)} }
```

(3) 最外面的函数要调用在产生闭包 fn()

注: 很多人认为在外层函数里面最后要 return 里面函数的值才算闭包(fn(){ a=1 fn1(){b=console(a)} return b }), 这只是闭包的一种用来延续里面(fn1)函数的闭包用法

闭包的作用:

1. 使用函数内部的变量在函数执行完后, 仍然存活在内存中(延长了局部变量的生命周期)

2. 让函数外部可以间接操作(读写)到函数内部的数据(变量/函数)

产生和死亡:

1. 产生: 在嵌套内部函数定义执行完时(创建函数对象)就产生了(不是在调用)

2. 死亡: 在嵌套的内部函数成为垃圾对象时

例子 1:

```
function fn1() {  
  var a = 2  
  function fn2() {  
    a++  
    console.log(a)  
  }  
  return fn2  
}  
var f = fn1()  
f() // 3  
f() // 4
```

因为第一次调用的时候是 3, 调用完之后, fn2 死了, 但是 a 的值还在, 所以再次调用就用 3+1=4, 返回 4

自定义 js 模块

定义 JS 模块

- * 具有特定功能的 js 文件

- * 将所有数据和功能都封装在一个函数内部(私有的)

- * 只向外暴露一个包含 n 个方法的对象或函数

- * 模块的使用者, 只需要通过模块暴露的对象调用方法来实现对应的功能

例如:

```
function myModule() {  
  // 1. 定义数据(变量)  
  var data = 'atguigu'  
  // 2. 定义操作数据的行为(函数)  
  function doSomething() {  
    console.log('doSomething()', data)  
  }  
  function doOtherthing() {  
    console.log('doOtherthing()', data)  
  }  
  // 3. 向外暴露行为(函数/包含多个函数的对象)  
  return {  
    doSomething: doSomething,  
    doOtherthing: doOtherthing  
  }  
}
```

调用:

```
var module = myModule()
```

```
module.doSomething()
```

```
module.doOtherthing()
```

自定义模块 2:

闭包的应用 2: 定义 JS 模块

- * 具有特定功能的 js 文件
- * 将所有数据和功能都封装在一个函数内部(私有的)
- * 只向外暴露一个包信 n 个方法的对象或函数
- * 模块的使用者, 只需要通过模块暴露的对象调用方法来实现对应的功能

能

例如 2

```
(function (window) {  
  // 1. 定义数据(变量)  
  var data = 'atguigu'  
  // 2. 定义操作数据的行为(函数)  
  function doSomething() {  
    console.log('doSomething()', data)  
  }  
  function doOtherthing() {  
    console.log('doOtherthing()', data)  
  } // 3. 向外暴露行为(函数/包含多个函数的对象)  
  window.module = {  
    doSomething: doSomething,  
    doOtherthing: doOtherthing } })(window)
```

调用:

```
module.doSomething()
```

```
module.doOtherthing()
```

闭包的优缺点

1. 缺点

- * 函数执行完后, 函数内的局部变量没有释放, 占用内存时间会变长
- * 容易造成内存泄露

2. 解决

- * 能不用闭包就不用
- * 及时释放

例如:

```
function fn1() {  
  var a = 2  
  function fn2() {  
    a++  
    console.log(a)  
  }  
  return fn2  
}  
var f = fn1()  
f() // 3  
f() // 4  
// 不再需要 a  
f = null // 及时释放
```

面试题 1:

```
var name = "The Window";
var object = {
  name: "My Object",
  getNameFunc: function () {
    return function () {
      return this.name;
    };
  };
};
console.log(object.getNameFunc()); //? The Window
```

调用的是一般调用，就是window，所以是输出window的name

对比

//代码片段二

```
var name2 = "The Window";
var object2 = {
  name2: "My Object",
  getNameFunc: function () {
    var that = this;
    return function () {
      return that.name2;
    };
  };
};
```

用了一个that保存了当前的this，后面调用的就是this，就是当前的调用者，所以返回的是当前函数的name

面试题 2

```
function fun(n, o) {
  console.log(o);
  return {
    fun: function (m) {
      return fun(m, n);
    }
  };
}

var a = fun(0);
a.fun(1);
a.fun(2);
a.fun(3); //undefined, 0, 0, 0

var b = fun(0).fun(1).fun(2).fun(3); //undefined, 0, 1, 2
b.fun(9); // 3

var c = fun(0).fun(1);
c.fun(2);
c.fun(3); //undefined, 0, 1, 1
```

fun(0)->fun(0,0)->colog(0)->0没有值->undefined->n=0->fun内(m, 0)

a.fun(1)->运行里面的函数->fun内(1,0)->fun外(1,0)->colog(0)->0->n=0

a.fun(2)->运行里面的函数->fun内(2,0)->fun外(2,0)->colog(0)->0->n=0

a.fun(3)->运行里面的函数->fun内(3,0)->fun外(3,0)->colog(0)->0->n=0

b.fun(9)->此时同上，无论是什么都是3

返回n=0

返回1

返回2

返回3

这个放面试题 3

不同方法创建对象

方式一: Object 构造函数模式

- * 套路: 先创建空 Object 对象, 再动态添加属性/方法
- * 适用场景: 起始时不确定对象内部数据
- * 问题: 语句太多

-->

```
/*
 一个人: name:"Tom", age: 12
*/
var p = new Object()
p = {}
// 先做一些计算处理, 确定数据
p.name = 'Tom'
p.age = 12
p.setName = function (name) {
  this.name = name
}

// 测试
console.log(p.name, p.age)
p.setName('Jack')
console.log(p.name, p.age)
```

方式二：对象字面量模式

- * 套路：使用{}创建对象，同时指定属性/方法
- * 适用场景：起始时对象内部数据是确定的
- * 问题：如果创建多个对象，有重复代码

-->

```
var p = {  
  name: 'Tom',  
  age: 12,  
  setName: function (name) {  
    this.name = name  
  }  
}
```

```
console.log(p.name, p.age)  
p.setName('Jack')  
console.log(p.name, p.age)
```

```
var p2 = {  
  name: 'Tom2',  
  age: 13,  
  setName: function (name) {  
    this.name = name  
  }  
}
```

方式三：工厂模式

- * 套路：通过工厂函数动态创建对象并返回
- * 适用场景：需要创建多个对象
- * 问题：对象没有一个具体的类型，都是 **Object** 类型

-->

```
function createPerson(name, age) {  
  // 工厂函数：每次调用返回一个新的对象  
  var p = {  
    name: name,  
    age: age,  
    setName: function (name) {  
      this.name = name  
    }  
  }  
  return p  
}  
var p1 = createPerson('Tom', 12)  
var p2 = createPerson('Tom2', 13)  
function createDog(name) {  
  var dog = {  
    name: name  
  }  
  return dog  
}  
var d1 = createDog('旺财')  
var d2 = createDog('来福')  
// d1 与 p1 的类型能区别吗？ 都没有自己的具体类型
```

方式四: 自定义构造函数模式

- * 套路: 自定义构造函数, 通过 `new` 创建对象
- * 适用场景: 需要创建多个类型确定的对象
- * 问题: 每个对象都有相同的数据, 浪费内存

里面的方法会重复出现, 写到原型去就可以只用一遍

```
<script type="text/javascript">
function Person(name, age) {
    this.name = name
    this.age = age
    this.setName = function (name) {
        this.name = name
    }
}
function Dog(name) {
    this.name = name
}

var p1 = new Person('Tom', 12)
var p2 = new Person('Tom2', 13)

var d1 = new Dog('xxx')
var d2 = new Dog('yyy')
```

// d1 与 p1 的类型是可以区别的

```
</script>
```

方式六: 构造函数+原型的组合模式

- * 套路: 自定义构造函数, 属性在函数中初始化, 方法添加到原型上
- * 适用场景: 需要创建多个类型确定的对象

-->

```
<script type="text/javascript">
function Person(name, age) {
    this.name = name
    this.age = age
}
Person.prototype.setName = function (name) {
    this.name = name
}

var p1 = new Person('Tom', 12)
var p2 = new Person('Tom2', 13)
// p1 与 p2 对象本身上没有 setName(), 只有原型上有一个唯一的 setName():
// 省内存
</script>
```

这样就只用执行一次 setName() 方法, 更省内存

Object.create(proto, [propertiesObject])

使用指定的原型对象及其属性去创建一个新的对象

根据现有的对象创建一个新的对象

Object.create(原型对象,可选的对象，新加的或者读取原型的)

-->

```
<script type="text/javascript">
```

```
var obj1 = {  
  name: 'Tom'  
}
```

```
var obj2 = Object.create(obj1)//创建一个新的 2 是 1 哪里来的
```

```
console.log(obj2, obj2.name)
```

//给 obj3 新添加属性

```
var obj3 = Object.create(obj1, {  
  age: { //添加一个 age，可修改属性  
    value: 12,  
    writable: true // 属性值可以修改  
  },
```

```
  sex: { //定义一个 sex 不可修改，无论怎么输都是男  
    value: '男',  
    writable: false // 属性值不能修改  
  }  
})//这样连着
```

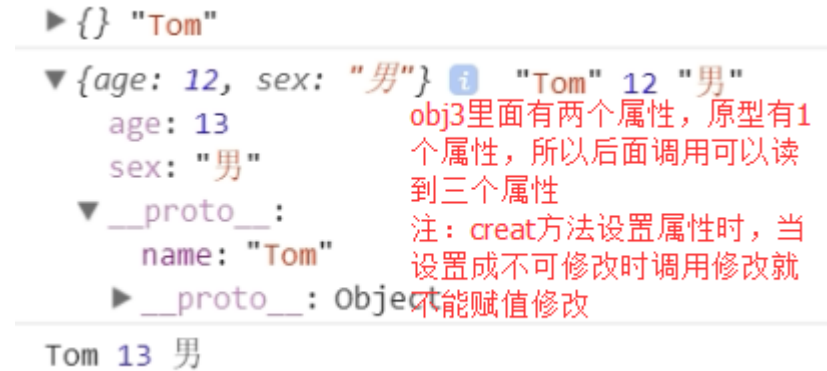
```
console.log(obj3, obj3.name, obj3.age, obj3.sex)
```

```
obj3.age = 13
```

```
obj3.sex = '女'
```

```
console.log(obj3.name, obj3.age, obj3.sex)
```

结果解析:



obj3里面有两个属性，原型有1个属性，所以后面调用可以读到三个属性
注：creat方法设置属性时，当设置成不可修改时调用修改就不能赋值修改

方式 1: 原型链继承

1. 套路

1. 定义父类型构造函数
2. 给父类型的原型添加方法
3. 定义子类型的构造函数
4. 创建父类型的对象赋值给子类型的原型
5. 将子类型原型的构造属性设置为子类型
6. 给子类型原型添加方法
7. 创建子类型的对象: 可以调用父类型的方法

2. 关键

1. 子类型的原型为父类型的一个实例对象

-->

1. 首先定义一个父类型的函数

```
function Parent() {  
    this.pProp = 'The parent'  
}
```

2. 给父类型的函数添加一个方法

```
Parent.prototype.showPProp = function () {  
    console.log('showPProp()', this.pProp)  
}
```

3. 定义一个子函数

```
function Child() {  
    this.cProp = 'The Child'  
}
```

4. 连接父子

- (1) 连接父子, 让子类型的原型指向父类型的实例

```
Child.prototype = new Parent()
```

- (2) 让子类型原型的 constructor 指向子类

```
Child.prototype.constructor = Child//完善代码
```

```
Child.prototype.showCProp = function () {  
    console.log('showCProp()', this.cProp)  
}
```

这样子函数就可以调用父函数的方法

```
var c = new Child ();  
c.showPprop();
```

借用构造函数继承(假的)

1. 套路:

1. 定义父类型构造函数
2. 定义子类型构造函数
3. 在子类型构造函数中调用父类型构造

2. 关键:

1. 在子类型构造函数中通用 `super()`调用父类型构造函数

-->

```
<script type="text/javascript">
```

```
function Person(name, age, sex) {  
    this.name = name  
    this.age = age  
    this.sex = sex  
}
```

```
function Student(name, age, sex, price) { // 身价
```

```
    /*this.name = name
```

```
    this.age = age
```

```
    this.sex = sex 可以写成下面这样*/
```

```
    Person.call(this, name, age, sex)
```

```
// (指定谁来调用这个函数, 后面都是 Person 函数的参数)
```

借用构造函数: 代码简洁

```
    this.price = price
```

```
}
```

```
var s = new Student('Tom', 19, '女', 10000)
```

```
console.log(s.name, s.age, s.sex, s.price)
```

方式 3: 原型链+借用构造函数的组合继承

1. 利用原型链实现对父类型对象的方法继承

2. 利用 `super()`借用父类型构造函数初始化相同属性

```
<script type="text/javascript">
```

```
function Person(name, age, sex) {  
    this.name = name  
    this.age = age  
    this.sex = sex  
}
```

```
Person.prototype.setName = function (name) {
```

```
    this.name = name
```

```
}
```

```
function Student(name, age, sex, price) { // 身价
```

```
    Person.call(this, name, age, sex) // 借用构造函数: 代码简洁
```

为了得到父类型的属性

```
    this.price = price }
```

```
Student.prototype = new Person() // 为了得到父类型的方法
```

```
Student.prototype.constructor = Student
```

```
Student.prototype.setPrice = function (price) {
```

```
    this.price = price
```

```
}
```

```
var s = new Student('Tom', 19, '女', 10000)
```

```
s.setName('Jack')
```

```
s.setPrice(11000)
```

```
console.log(s)
```

进程和线程

进程：

- 例如打开浏览器产生一次进程，再点击打开一次又是一个新的，从软件进去
- 程序的一次执行，它占有一片独立的内存空间

线程：

- 例如打开浏览器以后点击不同的网页，是线程，从进程进入
- 线程线程是一个独立的执行单位，是程序执行的一个完整的流程，是 CPU 的最小的调度单位
- 主线程只能有一个，副线程可以有很多

进程与线程的关系

- 应用程序必须在某个进程中运行
- 一个进程的数据可以供多线程共享（打开一个浏览器就可以用几个网页）
- 多个进程之间的数据是不能直接共享的（浏览器 1 不能和浏览器 2 共享）
- **线程池：**保存多个线程对象的容器，实现线程对象的反复利用

多线程（可用多个网页）优缺点：

优点：提高 CPU 的利用率

缺点：

创建多线程开销

线程之间切换开销（时间片，想想多个网页加载）

死锁与状态同步问题

时间片（这个还没执行完就去别的执行，这个暂定，跳来跳去耗费时间）

单线程：（只可以打开一个）

优点：顺序编程简单易懂

缺点：效率低

js 是单线程的，但是在 H5 中可使用 **Web Workers** 可以多线程运行
浏览器：**firefox** 一直是单进程，**ie** 一直是多进程

浏览器内核：不同浏览器可能不一样

主线程运行模块

js 引擎模块：负责 js 程序的编译与运行

html, css 文档解析模块：负责页面文本解析

DOM/CSS 模块：负责 DOM/CSS 在内存中的相关处理

布局和渲染模块：负责页面的布局和效果的绘制（内存中的对象）

分线程运行模块：

异步效果，都有回调函数，通过分线程来执行异步

定时器：负责定时器的管理,alter 会暂停计时和暂定主线程

定时器真是定时执行的吗？

- * 定时器并不能保证真正定时执行

- * 一般会延迟一丁点(可以接受),也有可能延迟很长时间(不能接受)

定时器回调函数是在分线程执行的吗？

- * 在主线程执行的,js 是单线程的

定时器是如何实现的？

- * 事件循环模型(后面讲)

DOM 事件响应模块：负责事件的管理

网络请求模块：负责 ajax 请求

页面先主线程执行，把页面渲染完了之后才看他的动作

JS 单线程

1. 如何证明 js 执行是单线程的？

- * setTimeout()的回调函数是在主线程执行的

- * 定时器回调函数只有在运行栈中的代码全部执行完后才有可能执行

2. 为什么 js 要用单线程模式，而不用多线程模式？

- * JavaScript 的单线程，与它的用途有关。

- * 作为浏览器脚本语言，JavaScript 的主要用途是与用户互动，以及操作 DOM。

- * 这决定了它只能是单线程，否则会带来很复杂的同步问题

3. 代码的分类：

- * 初始化代码：初始化执行的代码

- * 回调代码：回调函数中的代码

4. js 引擎执行代码的基本流程

- * 先执行初始化代码：包含一些特别的代码

- * 设置定时器

- * 绑定监听

- * 发送 ajax 请求

- * 后面在某个时刻才会执行回调代码

事件：

1. 所有代码分类

- * 初始化执行代码(同步代码)：包含绑定 dom 事件监听，设置定时器，发送 ajax 请求的代码

- * 回调执行代码(异步代码)：处理回调逻辑

2. js 引擎执行代码的基本流程：

- * 初始化代码==>回调代码

3. 模型的 2 个重要组成部分：

- * 事件管理模块

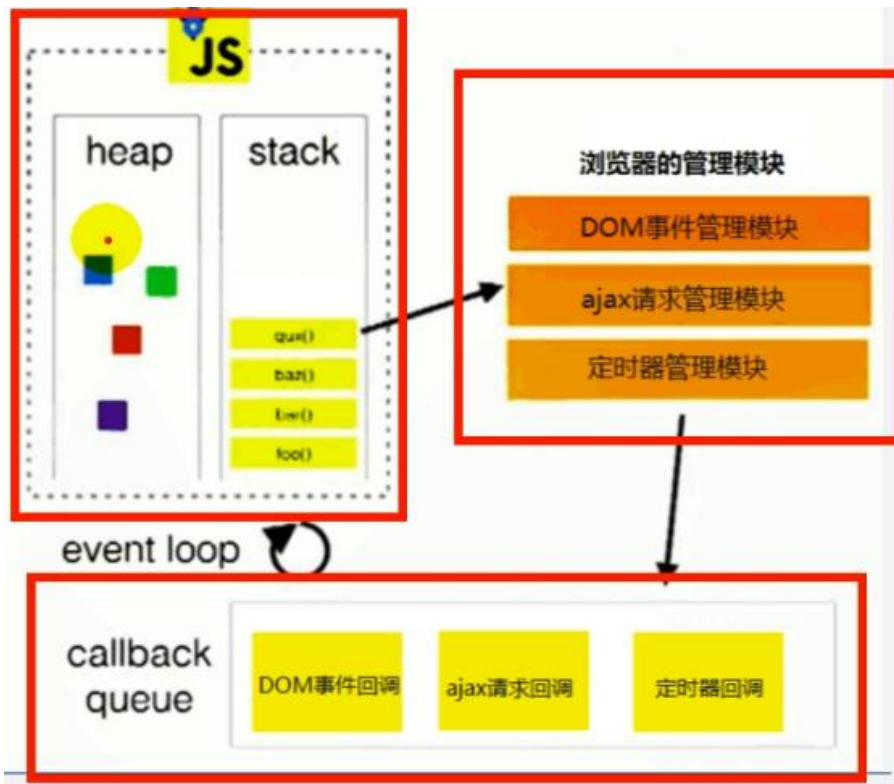
- * 回调队列

4. 模型的运转流程

- * 执行初始化代码，将事件回调函数交给对应模块管理

- * 当事件发生时，管理模块会将回调函数及其数据添加到回调队列中

* 只有当初始化代码执行完后(可能要一定时间), 才会遍历读取回调队列中的回调函数执行



1. 内存溢出

- * 一种程序运行出现的错误
- * 当程序运行需要的内存超过了剩余的内存时, 就抛出内存溢出的错误

2. 内存泄露

- * 占用的内存没有及时释放
- * 内存泄露积累多了就容易导致内存溢出
- * 常见的内存泄露:
 - * 意外的全局变量
 - * 没有及时清理的计时器或回调函数
 - * 闭包没有及时释放

1. 内存溢出

```
/*var obj = {}  
for (var i = 0; i < 100000; i++) {  
    obj[i] = new Array(10000000)  
}*/
```

2. 内存泄露

```
// 意外的全局变量  
function fn () {  
    a = [] //不小心没有 var 定义  
}  
fn()  
// 没有及时清理的计时器  
setInterval(function () {  
    console.log('----')  
}, 1000)
```

