

# Table of Contents

---

1. JavaScript 高级
2. 面向对象进阶
  - i. 面向对象基础复习
  - ii. 原型的基本概念
  - iii. 继承的概念
  - iv. Object 对象
  - v. Function 对象
  - vi. 作用域链
  - vii. 闭包的基本概念
  - viii. getter 与 setter
3. 面向对象高级
  - i. 原型链
  - ii. 函数的四中调用模式
  - iii. 对象的创建模式
  - iv. 继承的方式
  - v. 闭包案例
  - vi. 正则表达式对象
  - vii. XMLHttpRequest 对象

# JavaScript 高级

---

JavaScript 高级教程讲义, 本文主要作为 JavaScript 高级课程讲义.

# 面向对象进阶

---

介绍创建方法, 和使用技巧, 以及 继承

- 原型的基本概念
- 继承的概念
- Object 对象
- Fountion 对象
- 作用域链
- 闭包的基本概念
- getter 与 setter

# 面向对象基础复习

---

主要内容:

- JavaScript 的构成以及 ECMAScript 与 JavaScript 的关系
- JavaScript 中的数据类型以及转换
- JavaScript 的语句
- JavaScript 中面向对象的基本概念
- 构造函数, 属性和方法的概念
- 常见内置对象
- 调试器的使用

# 原型的基本概念

---

主要内容:

- 原型对象的概念
- 获得原型对象的方法
- 使用原型对象
- `__proto__` 和 `Object.prototype`

在 JavaScript 中对象是一个由两个对象组成的概念. 这句话比较晦涩, 下面通过分析来详细讨论对象和原型对象的概念.

## 原型对象的概念

---

### 什么是原型

JavaScript 中原型的概念来源于 Self 语言. Self 中将精简作为设计原则. 设计之初就舍去了类的概念, 只有对象的概念. 因此在 JavaScript 中出现了原型继承之说, 即 JavaScript 的继承所依附的是原型对象.

在 JavaScript 中, 对象 `{}` 中包含 `toString`, `valueOf`, 和 `constructor` 等方法, 实际上就是由原型继承来实现的. 对象继承自原型对象

`Object.prototype`.

### 为什么需要原型

解释为何使用原型之前, 首先看下面的案例.

```
// 创建一个构造方法
function Person() {
  this.sayHello = function() {
    console.log("你好, 我是蒋坤, 很高兴认识你!");
  };
}
```

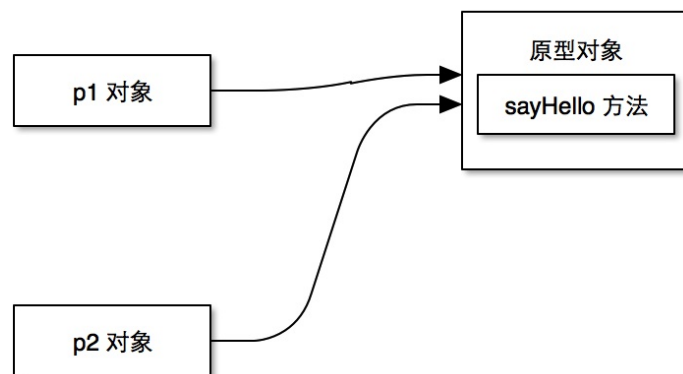
```
};  
}  
  
// 创建两个对象  
var p1 = new Person(),  
    p2 = new Person();  
  
// 调用和比较  
p1.sayHello();  
p2.sayHello();  
console.log("p1.sayHello === p2.sayHello 的结果是: " +  
    (p1.sayHello === p2.sayHello));
```

表达式 `p1.sayHello === p2.sayHello` 的运行结果是 `false`. 可见两个对象的方法是两个独立的方法.

在 JavaScript 中函数是对象, 也就是说 `sayHello` 方法也是对象的成员. 与 `num = 123`, `name = "蒋坤"` 一样, `sayHello` 也是一个变量. 在内存逻辑中:



如图, 在内存中, 对象 `p1` 和对象 `p2` 同时具有独立的两个成员 `sayHello`, 都占有内存. 而 `sayHello` 作为方法是相同的一段执行逻辑. 无论是什么对象, 执行逻辑都是一样的. 因此应该将该逻辑共享. 内存逻辑应该为:



因此, 在 JavaScript 中有了原型的概念, 原型用来保存那些共享的数据和方法.

## 原型也是对象

在 JavaScript 中, 一个对象是由两部分构成的, 一个是当前对象, 一个是原型对象. 如下面代码:

```
function Person() {  
    this.name = "蒋坤";  
    this.sayHello = function() {  
        console.log("你好, 我是" + this.name);  
    };  
}  
var p = new Person();
```

这段代码中, 定义了一个 `Person` 构造函数, 创建了一个 `Person` 类型的对象 `p`. 它就是当前对象. 构造方法是如何设计的, 那么它就具有什么成员.

但是对象 `p` 还包含其他方法, 例如 `toString`. 这个方法就是由原型对象提供.

```
console.log(p.toString()); // => [object Object]
```

也就是说一个对象由构造方法创建的部分, 和原型部分一起构成.

## 原型属性

每一个对象都有一个原型, 这个原型常常称为"这个对象的原型属性", 或"这个对象的原型".

## 原型对象

对象的原型实际上是构造函数的一个属性. 在定义构造函数的时候就定义了其原型. 如果需要引用原型对象使用语法:

```
<构造方法>.prototype
```

在原型对象中定义的所有成员, 都将被其构造方法创建出的对象所继承. 也就是说原型对象中有什么, 那么由构造方法创建出来的对象就有什么.

## 设置原型对象

由于原型中所有的成员, 对象中都会有, 因此可以将之前的案例修改成:

```
// 创建一个构造方法
function Person() {
}

// 给其原型添加方法
Person.prototype.sayHello = function() {
    console.log("你好, 我是蒋坤, 很高兴认识你!");
};

// 创建两个对象
var p1 = new Person(),
    p2 = new Person();

// 调用和比较
p1.sayHello();
p2.sayHello();
```



```
console.log("p1.sayHello === p2.sayHello 的结果是: " +  
(p1.sayHello === p2.sayHello));
```

此时运行, 表达式 `p1.sayHello === p2.sayHello` 的结果便是 `true`. 也就是说这两个对象是共享同一个方法的执行代码.

因此, 在编写构造函数创建对象的时候, 应该提供两部分: 一个是构造方法, 一个是原型对象. 将属性部分写在构造方法中, 将方法添加到原型对象中.

### \_\_proto\_\_

想要获得原型对象, 必须使用构造方法, 即 `<构造方法>.prototype`. 但是在代码上下文中, 获得指定对象的原型对象就不那么方便了.

Mozilla 实现的 JavaScript 中引入了一个 `__proto__` 属性接口. 利用该属性可以直接访问对象的原型对象.

```
function Person() {  
}  
var p = new Person();  
console.log(p.__proto__ === Person.prototype); // => true
```

但是, 不建议在开发中直接使用该属性, 虽然 FireFox, Chrome 和 Safari 实现了该属性, 但并不是所有的浏览器.

### 技巧

为了兼容不支持 `__proto__` 属性的浏览器, 可以提供一个通用方法来解决

```
// 使用 __jkproto 来表示对象的原型对象  
if (!({}).__proto__) {  
    Object.prototype.__jkproto = function () {  
        return this.constructor.prototype;  
    }  
} else {
```

```
Object.prototype.__jkproto = function() {  
    return this.__proto__;  
}  
}
```

## 小结

- 对象由两部分构成, 一个是原型对象, 一个是构造方法描述的对象.
- 获得原型引用使用 `<构造方法>.prototype` 或者 `<对象>.__proto__`.
- 设计构造方法, 将属性写在构造方法中, 将方法添加到原型对象中.

# 继承的概念

---

主要内容:

- 继承的概念
- 原型继承的概念
- `Object.prototype`
- 一种继承的实现

JavaScript 是基于面向对象的编程语言, 继承是其中比较重要的概念. 接下来详细讨论继承的基本内容.

## 继承的概念

---

在开发过程中, 面向对象是一种处理代码的思考方式. 在面向对象中继承就是其中一个非常重要的概念. 接下来本节详细讨论继承的概念.

## 为何需要使用继承

在开发中, 经常会发现多个不同类型的对象都有共同的方法. 例如代码:

```
var o1 = new Number(144);
console.log("打印 Number 对象 o1 = " + o1);
var o2 = new Date();
console.log("打印 Date 对象 o2 = " + o2);
var o3 = new Array(1, 4, 4);
console.log("打印 Array 对象 o3 = " + o3);
var o4 = new Error("a test");
console.log("打印 Error 对象 o4 = " + o4);
function MyConstructor() {
}
var o5 = new MyConstructor();
console.log("打印自定义对象 MyConstructor o5 = " + o5);
```

在本例中, 打印对象都有一个共同的操作, 即 "和字符串相连接". 该操作在执行的时候, 会自动的调用一个方法, 即 `toString()` 方法. 因此, 执行代码

```
console.log("打印自定义对象 MyConstructor o5 = " + o5);
```

等价于执行代码

```
console.log("打印自定义对象 MyConstructor o5 = " + o5.toString());
```

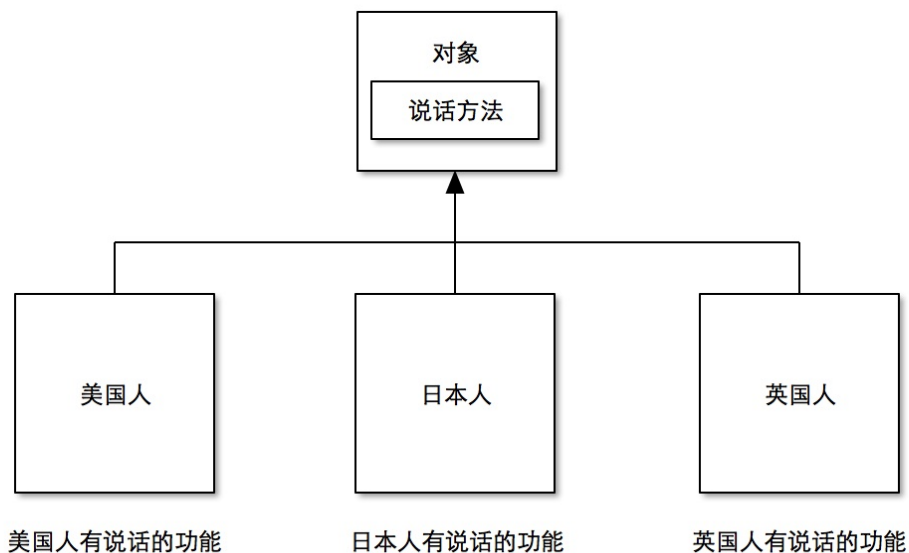
那么问题来了, 这个类型中没有定义 `toString()` 方法呢? 很显然是没有的. 那么如何实现?

其实很简单, 就是为了 **"复用"**.

在开发中常常会有重复的操作, 例如页面上很多东西都可以点击; 一个游戏中, 常常会有角色可以走动等等. 因此将重复执行的代码提取出来, 不用再编写代码的时候每次都要将其再写一遍. 那么这种拿来主义就是继承.

在继承中, 一个对象继承自另一个对象. 继承的对象中包含被继承对象的所有成员.

例如人会说话, 那么将说话的功能提取出来作为一个对象. 继承自该对象的美国人, 日本人, 或是英国人就都具有说话的方法了.



因此一句话总结继承, 就是 **"为了偷懒, 就拿来主义"**.

## 继承的实现方式

继承的实现方式有很多种, 如今主流的继承有类继承和原型继承. 类继承的语言有: C++, Java, C# 等, 而原型继承有: Self, Io, JavaScript 等.

### C# 中基于类的继承

在基于类继承的编程语言中, 有一个对象的模板, 称之为类. 需要对象则首先设计一个类, 由类来创建对象. 而继承是指类之间的继承.

例如写一个父类

```
class BaseClass {
    public void SayHello() {
        System.Console.WriteLine("你好");
    }
}
```

然后提供一个子类

```
class SubClass : BaseClass {
```

```
}
```

最后直接创建子类独享, 即可调用方法

```
public static void Main(string[] args) {  
    SubClass o = new SubClass();  
    o.SayHello(); // => 你好  
}
```

这里的继承实际上是利用模板来实现的. 在模板 `BaseClass` 中定义了一个方法 `SayHello`, 然后设计一个子类 `SubClass` 继承自 `BaseClass`, 在子类中没有规定任何东西. 但是由子类创建出来的对象具有 `SayHello` 方法, 并且可以调用.

这个就是利用类的继承. 类继承了, 那么由该类创建出来的对象就具有被继承的成员.

## 原型继承

与类继承不同, 在 `JavaScript` 中, 使用原型来实现继承. 在原型中定义的所有成员都会被其构造方法创建的对象所继承. 在 `JavaScript` 中不存在类的概念, 因此实现继承的方式也不再唯一和统一.

还是说话的例子, 使用 `JavaScript` 来实现

```
// 定义一个对象, 将来作为原型对象  
var proto = {  
    sayHello : function() {  
        console.log("你好!!!");  
    }  
};  
  
// 定义一个构造函数  
function Person() {  
}  
  
// 设置 Person 的原型
```

```
Person.prototype = proto;

// 创建对象, 具有 sayHello 方法
var p = new Person();
p.sayHello();
```

在本例中没有类的概念, 继承也不是模板之间的继承. 而是给构造方法设置一个原型对象, 由该构造函数创建出来的对象就具有该原型对象中的所有内容. 我们称这个对象就继承自原型对象.

注意: 前面曾经介绍过 `__proto__` 的概念, 因此实现继承的方法也就不统一了, 比较随意.

值得说明的是, 所有由该类创建出来的对象, 都具有了原型中定义的属性 (方法). 与定义和设置的顺序无关. 但是如果重新设置属性就不正确了.

例如, 下面的代码可以正常执行.

```
function Person() {
}

var p1 = new Person();

Person.prototype.sayHello = function() {
    console.log("hello, 你好 JavaScript!");
};

var p2 = new Person();

p1.sayHello(); // => hello, 你好 JavaScript!
p2.sayHello(); // => hello, 你好 JavaScript!
```

上面的代码, 给 `Person` 的原型添加了一个 `sayHello` 方法, 因此两个 `Person` 对象都可以调用该方法.

如果是直接重新设置构造函数 `Person` 的原型对象, 那么就会报一个 `TypeError` 异常.

```
function Person() {  
}  
  
var p1 = new Person();  
  
Person.prototype = { sayHello : function() {  
    console.log("hello, 你好 JavaScript!");  
}};  
  
var p2 = new Person();  
  
p1.sayHello(); // => 异常  
p2.sayHello();
```

原因很简单, 这个原型赋值修改了构造函数 `Person` 的原型对象类型.

```
function Person() { }  
  
var p1 = new Person();  
  
Person.prototype = { sayHello : function() {  
    console.log("hello, 你好 JavaScript!");  
}};  
  
var p2 = new Person();  
  
console.log(p1.constructor); // => function Person() { }  
console.log(p2.constructor); // => function Object() { [native code]}
```

可见修改后, 原型不再是 `Person` 类型的了, 而是 `Object` 类型.

## Object.prototype

在 JavaScript 中, 每一个对象都有原型. 而且每一个原型都直接, 或间接的继承自 `Object.prototype` 对象.



```
function Person() {}
```

定义了一个构造函数, 那么他就有一个 `Object` 类型的原型对象被设置给了 `Person.prototype` .

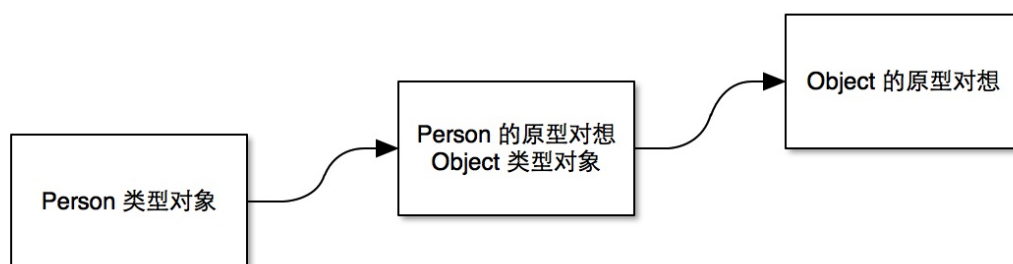
```
console.log(Person.prototype);  
console.log(Person.prototype instanceof Object); // => true  
// note: 可见 Person.prototype 是 Object 类型的
```

如此, 由 `Person` 创建出来的对象就继承自 `Object` 类型的对象. 而 `Person.prototype` 也是一个对象. 自然也有一个原型对象.

在 `Chrome` 中可以使用 `__proto__` 引用. 因此可以获得 `Person.prototype` 的原型对象

```
console.log(Person.prototype.__proto__); // => Object {}
```

那么从逻辑上就有下面的继承模型:



需要注意的是, `Object` 类型的原型对象的原型对象为 `null` .

```
console.log(Person.prototype.__proto__.__proto__ === null);  
// => true
```

## 一种继承的实现

有了上面的分析, 实现继承就可以分为三个步骤:

1. 获得构造函数 `F`
2. 设置 `F` 的原型为被继承的对象
3. 用 `F` 创建继承对象

简单的实现为

```
// 被继承对象
var pareObj = {
  sayHello: function() {
    console.log("Hello, jk");
  }
};

// 创建构造函数
function F() {
}

// 设置原型对象
F.prototype = pareObj;

// 创建继承对象
var obj = new F();
```

但是这么写太过于繁琐, 因此大师 Douglas Crockford 在他的《JavaScript: The Good Parts》中给出了下面的做法:

```
if (!Object.create) { // ECMAScript 5 中已经支持该方法
  Object.prototype.create = function (pare) {
    function F() {}
    F.prototype = pare;
    return new F();
  }
}
```

## 小结

---

- 所谓继承就是拿来主义, 将重复的东西进行复用
- JavaScript 中继承就是给构造函数的 `prototype` 设置对象
- 每一个对象都有一个原型对象
- 一个经典的继承方法

# Object 对象

---

主要内容:

- `Object` 对象和其他对象的关系
- `Object` 对象中已的方法
- 字面量的类型
- `new` 的含义

与其他面向对象的编程语言 ( Java, C# ) 类似, 可以说, JavaScript 中所有的对象都来自于 `Object` . 但是又不完全是这样. JavaScript 是利用原型来实现继承的. 为了弄清楚 JavaScript 的继承的本质, 接下来首先讨论一下 `Object` 对象的一些细节.

## Object 对象和其他对象的关系

---

要讨论 `Object` 对象与其他对象之间的关系, 首先来看对象的类型.

### `typeof` 运算符和对象的类型

在 JavaScript 中, 数据类型分为基本类型和复合类型. 基本类型都有自己的类型名, 但是复合类型都被描述为 `object`. 可以使用 `typeof` 运算符加一验证.

```
console.log(typeof 123);           // => number
console.log(typeof 1.23);          // => number
console.log(typeof true);           // => boolean
console.log(typeof "jk");           // => string
console.log(typeof []);             // => object
console.log(typeof new Date());     // => object
console.log(typeof {});             // => object
console.log(typeof /.)/);           // => object
```

可见 JavaScript 中将对象都当做 `object` 来对待. 其实这也是有原因的. 在

JavaScript 中所有的对象都包含 `toString()` , `valueOf()` 等方法, 实际上这些方法都来源于 `Object` , 严格上讲是 `Object` 的原型 `Object.prototype` .

```
// 有这个方法所有为 true
console.log(![].toString); // => true
console.log(!(new Date).valueOf); // => true
function MyCtr() {}
console.log(!(new MyCtr).hasOwnProperty); // => true
// 不包含才返回 false
console.log(!{}.jkMethod); // => false
```

原因很简单, 即使任何对象都直接或间接的继承自 `Object.prototype` 对象.

## Object 的原型

了解到所有对象都直接或间接继承自 `Object.prototype` 后, 我们来理一理 `Object` 和 `Object.prototype` 以及其他对象之间的关系.

首先来看 `Object` 对象与 `Object.prototype` 的关系.

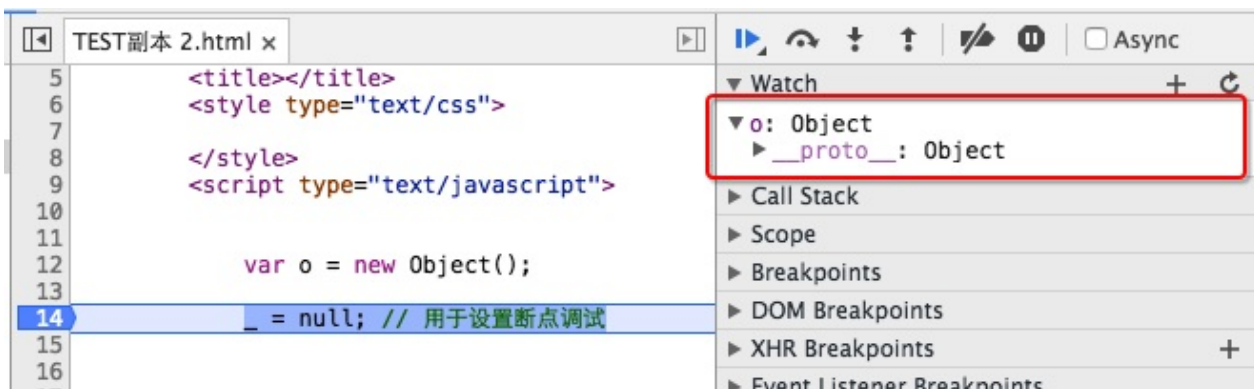
## Object 对象与它的原型

先来看一段代码:

```
var o = new Object();

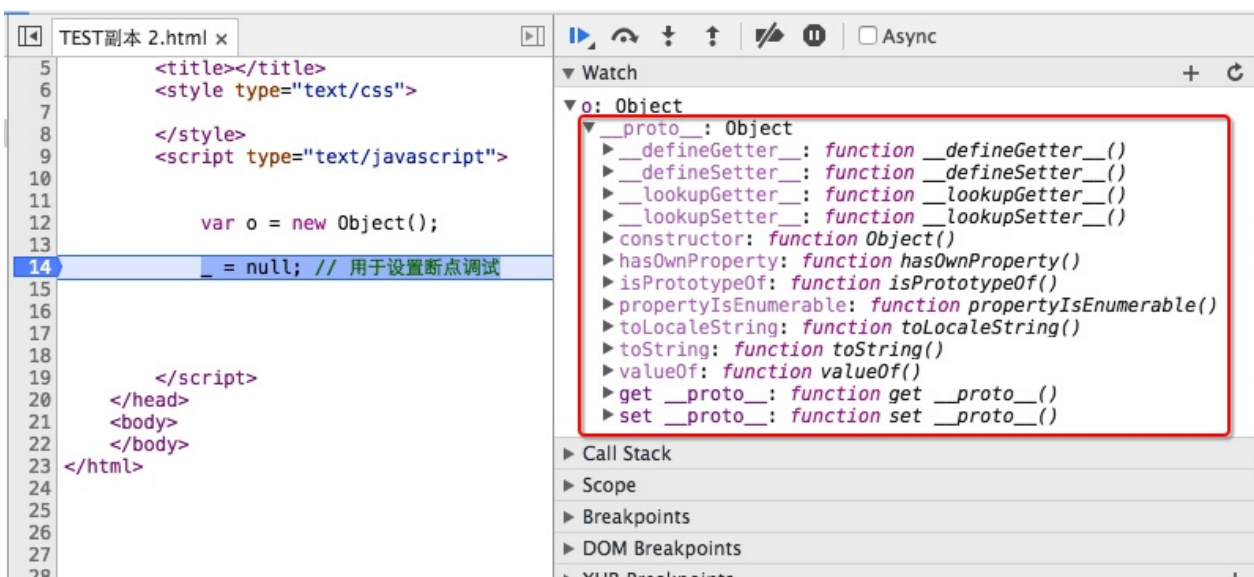
_ = null; // 用于设置断点调试
```

设置断点运行后:

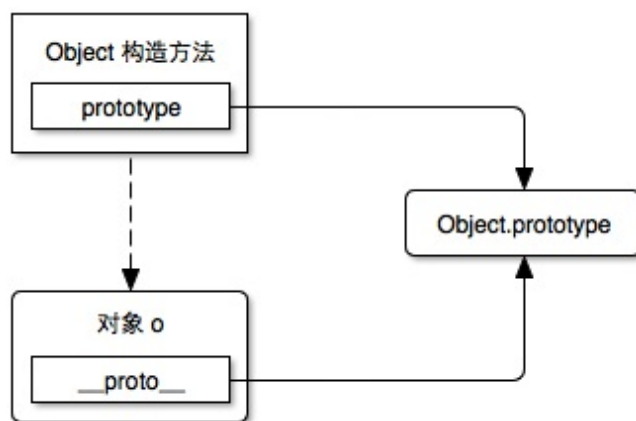


可见由 `Object` 构造方法创建出来的对象没有任何成员。

那些对象都共有的方法, 实际上都来源于 `__proto__`。



暂时不管这些方法的含义, 这里 `o.__proto__` 就是 `Object.prototype`。因此他们之间的关系就很清楚了。



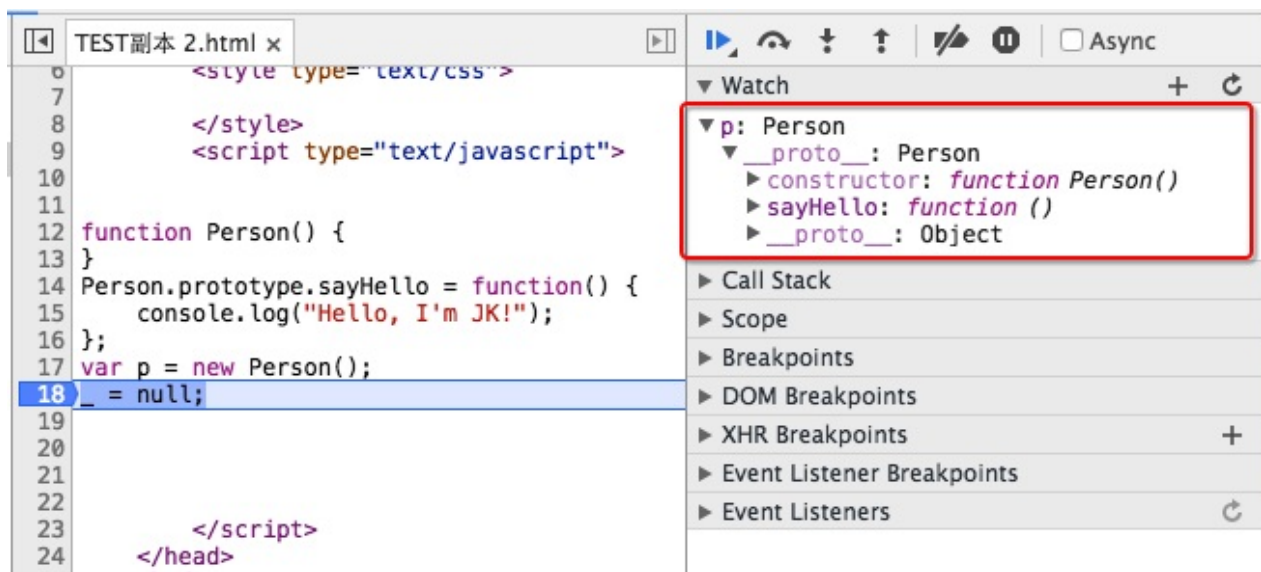
注意: `__proto__` 并非标准属性。

## 其他对象的原型和 `Object.prototype`

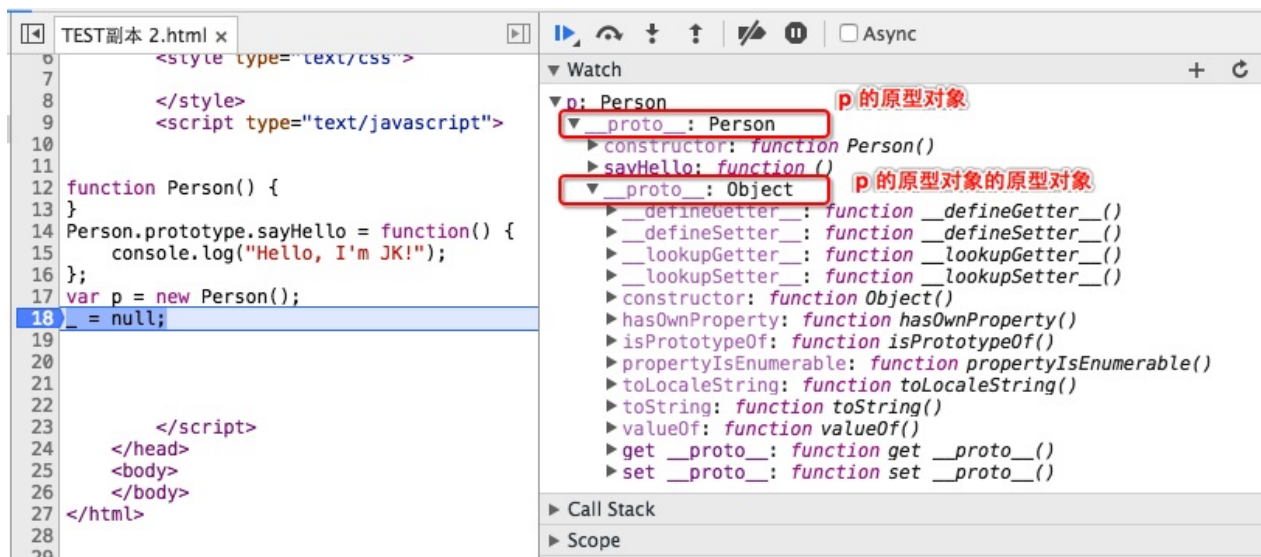
了解了 `Object` 对象与其原型的关系后, 下面来看看一般对象与原型的关系, 以及一般对象与 `Object.prototype` 的关系.

```
function Person() {  
}  
Person.prototype.sayHello = function() {  
    console.log("Hello, I'm JK!");  
};  
var p = new Person();  
_ = null;
```

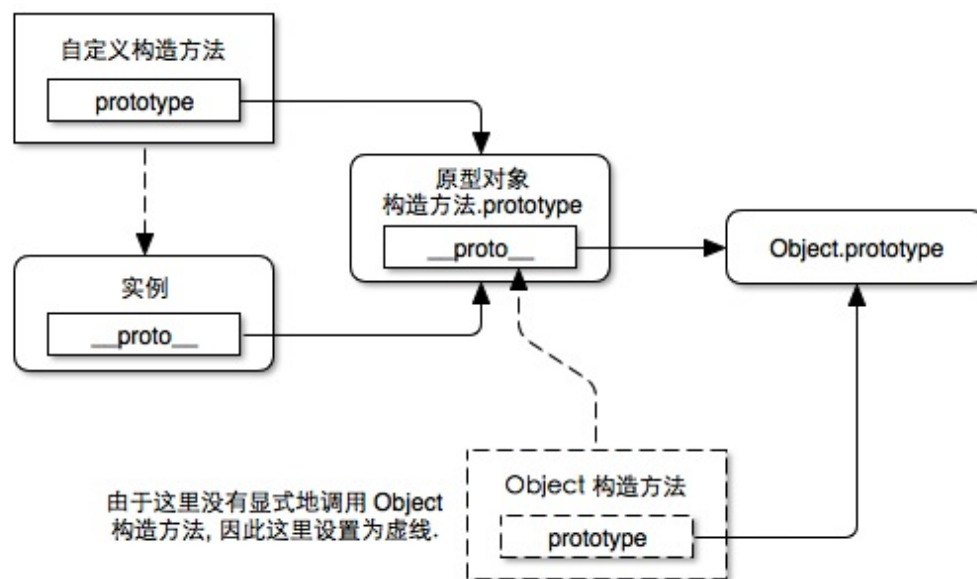
运行设置断点后:



可见其结构与 `Object` 对象和 `Object.prototype` 的结构一样. 然而原型对象也是有原型的, 虽然无法使用构造方法, 但是可以使用 `__proto__` 来查看.



从中可以发现自定义构造方法创建出来的对象 ( `new <构造方法>` ) 有一个原型对象, 即 `<构造方法>.prototype` . 而该原型对象也有原型对象. 根据图中可以看到原型对象的原型对象是 `Object` 类型的. 可见, 自定义构造方法创建对象的原型对象就是 `Object` 构造方法的实例.



从图中可以看出自定义对象和 `Object` 对象, 以及之间原型对象的关系.

## 内置对象

前面讨论了自定义对象, 接下来看看内置对象是否也复合这个关系.

首先看看数组:



```
var arr = [];
_ = null;
```

设上断点后运行

Uncaught TypeError: Cannot read property 'length' of null

Watch

arr: Array[0]  
length: 0

\_\_proto\_\_: Array[0] (数组对象的原型对象)

concat: function concat()  
constructor: function Array()  
copyWithin: function copyWithin()  
entries: function entries()  
every: function every()  
fill: function fill()  
filter: function filter()  
find: function find()  
findIndex: function findIndex()  
forEach: function forEach()  
includes: function includes()  
indexOf: function indexOf()  
join: function join()  
keys: function keys()  
lastIndexOf: function lastIndexOf()  
length: 0  
map: function map()  
pop: function pop()  
push: function push()  
reduce: function reduce()  
reduceRight: function reduceRight()  
reverse: function reverse()  
shift: function shift()  
slice: function slice()  
some: function some()  
sort: function sort()  
splice: function splice()  
toLocaleString: function toLocaleString()  
toString: function toString()  
unshift: function unshift()  
Symbol(Symbol.iterator): function values()  
Symbol(Symbol.unscopables): Object  
\_\_proto\_\_: Object

Call Stack

数组对象的原型对象的原型对象

可以看到, 与自定义对象一致.

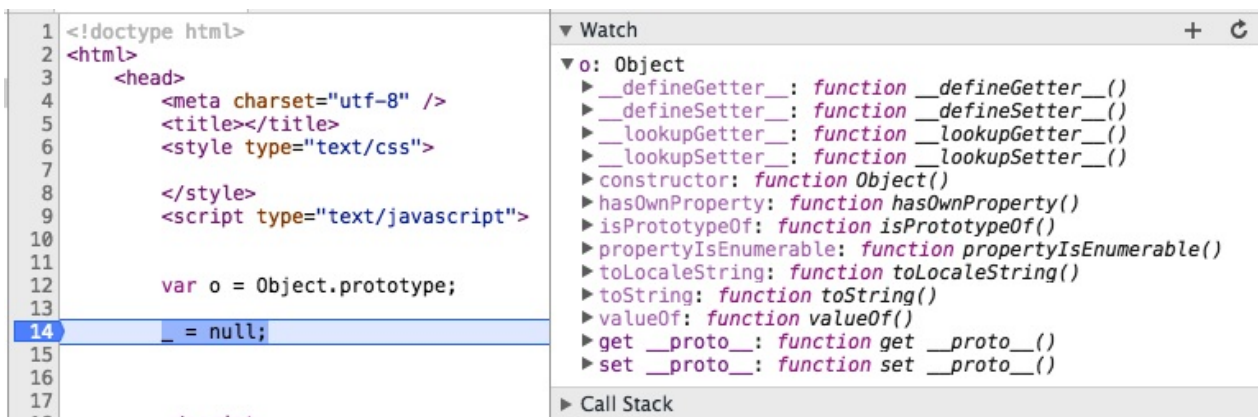
同理验证 Date 对象, Error 对象都是如此. 但是 Math 对象不同. Math 本身是一个 Object 类型的对象.

## Object 原型中定义的方法

接下来讨论 Object 原型对象中定义的方法.

```
var o = Object.prototype;
_ = null;
```

## 设定断点, 调试查看



可见原型对象中包含的都是方法:

- `__defineGetter__` 方法
- `__defineSetter__` 方法
- `__lookupGetter__` 方法
- `__lookupSetter__` 方法
- `constructor` 方法
- `hasOwnProperty` 方法
- `isPrototypeOf` 方法
- `propertyIsEnumerable` 方法
- `toLocaleString` 方法
- `toString` 方法
- `valueOf` 方法
- `get __proto__` 读写器
- `set __proto__` 读写器

在这些方法中, 带有两个下划线开头和两个下划线结尾的方法是标准方法. 因此在部分浏览器中可能没有实现.

**constructor 方法.** 该方法即构造方法. 调用构造方法, 可以创建对象. 那么这个属性就是该对象构造方法的引用.

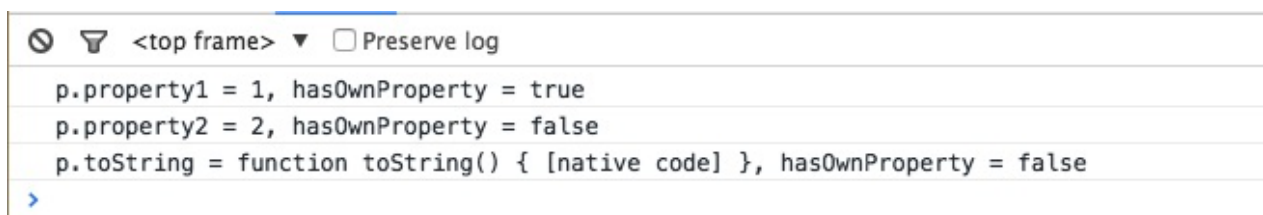
```
function Person() {}  
var p = new Person();
```

```
console.log(typeof p.constructor); // => function
console.log(p.constructor === Person); // => true
```

**hasOwnProperty** 方法. 该方法需要一个字符串参数, 用来判断该字符串表示名字的属性是否为非继承属性. 如果是非继承属性, 就返回 `true`, 否则返回 `false`.

```
function Person() {
    this.property1 = 1;
}
Person.prototype.property2 = 2;
var p = new Person();
console.log("p.property1 = " +
    p.property1 +
    ", hasOwnProperty = " +
    p.hasOwnProperty("property1"));
console.log("p.property2 = " +
    p.property2 +
    ", hasOwnProperty = " +
    p.hasOwnProperty("property2"));
console.log("p.toString = " +
    p.toString +
    ", hasOwnProperty = " +
    p.hasOwnProperty("toString"));
```

执行结果为



```
p.property1 = 1, hasOwnProperty = true
p.property2 = 2, hasOwnProperty = false
p.toString = function toString() { [native code] }, hasOwnProperty = false
```

**isPrototypeOf** 方法. 如果目标对象是参数对象的原型对象, 那么就返回 `true`, 否则返回 `false`.

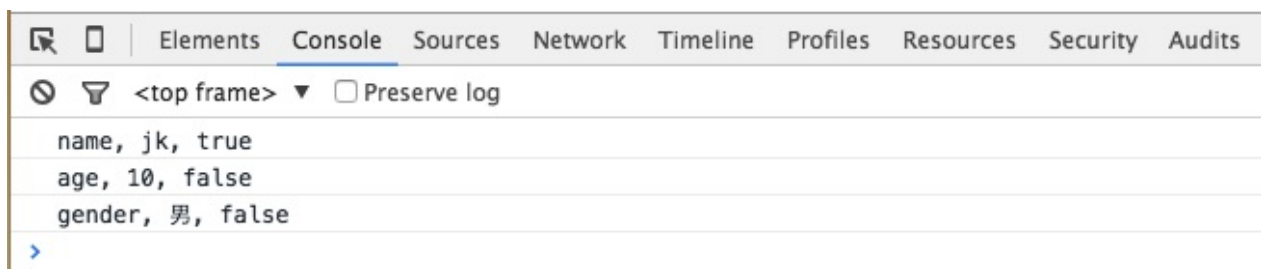
```
function Person() {}
var p = new Person();
```

```
console.log(p.isPrototypeOf(p)); // => false
console.log(Person.prototype.isPrototypeOf(p)); // => true
console.log({}.isPrototypeOf(p)); // => false
```

**propertyIsEnumerable** 方法. 该方法需要一个字符串参数, 如果该字符串表示的名字是对象的自有属性, 同时该属性可枚举, 那么方法就会返回 `true`, 否则返回 `false`.

```
function Person() {
    this.name = "jk";
}
Person.prototype.age = 10;
Object.prototype.gender = "男";
var p = new Person();
for (var k in p) {
    console.log(k + ", " + p[k] + ", " + p.propertyIsEnumerable(k));
}
```

执行结果为



实际上该方法与 `hasOwnProperty` 方法类似. 在 JavaScript 中只有一些内置的系统方法无法枚举. 因此该方法几乎总是返回 `true`. 但是在 ECMAScript 5 中给出了设置对象特性的方法, 可以设置某个属性是否可以枚举.

**toString** 方法和 **toLocaleString** 方法. 这两个方法都是将对象转换成字符串. 不同的是 `toLocaleString` 是转换为本地字符串. 一般对象与 `toString` 一样. 但是时间类型不同.

```
var arr = [1,2,3];
console.log(arr.toString());
console.log(arr.toLocaleString());
var now = new Date();
console.log(now.toString());
console.log(now.toLocaleString());
```

执行结果为



**valueOf** 方法. 当对象需要转换为数字类型的时候就会调用该方法. 该方法与 **toString** 类似.

```
var a1 = [1,2];
var a2 = [1,2];
a1.valueOf = function() {
    console.log("调用了 valueOf");
    return 3;
};
a1.toString = function() {
    console.log("调用了, toString");
    return "---";
};
console.log(Number(a1));
console.log(Boolean(a1));
console.log(String(a1));
console.log(Number(a2));
```

执行结果为



至于 `get` 和 `set` 以及带有两个下划线的方法, 后面再详细说明.

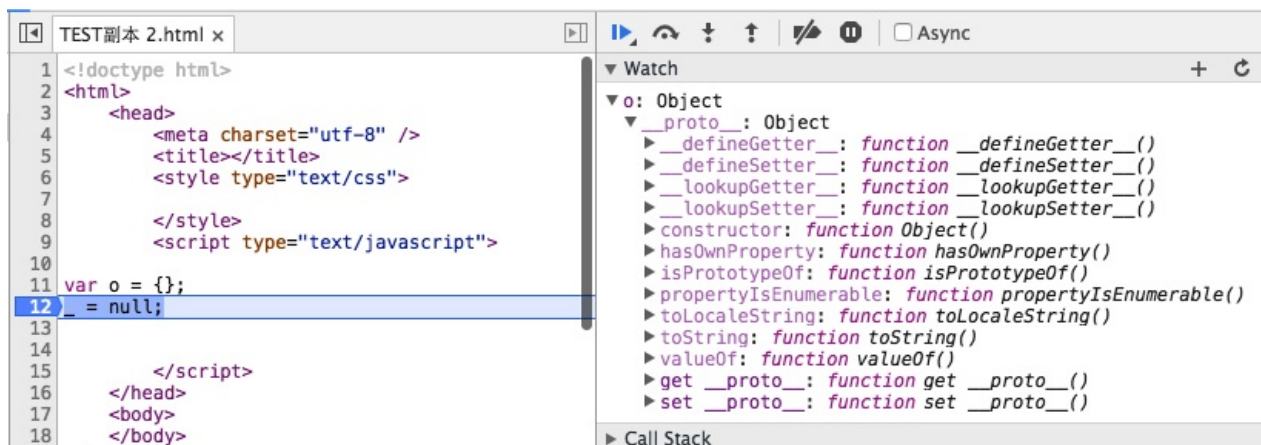
## 字面量的类型

在实际开发中, 由于字面量不需要定义构造方法, 也不需要 `new` 等操作. 因此字面量的效率会比较高, 因此尽量使用对象的字面量. 但是, 没有构造方法, 也就表明无法实现复用. 因此选择需要权衡.

那么字面量的类型是什么呢? 还是利用调试来进行查看

```
var o = {};  
_ = null;
```

设断点后:



可见对象的字面量是 `Object` 类型的. 它的原型就是 `Object.prototype`.

## new 以及构造函数的执行过程

除了字面量以外, 对象也可以使用 `new` 构造函数的形式被创建出来. 使用构造函数与使用对象字面量的最大区别在于, 使用构造方法可以复用, 利用构造方法可以创建出多个对象; 但是使用对象的字面量, 只会有一个对象.

那么构造函数具体做了什么, 下面来进行说明.

```
function Person(name, age, gender) {  
    this.name = name;  
    this.age = age;  
    tis.gender = gender;  
}  
var p = new Person("jk", 19, "m");
```

这里定义了构造函数 `Person`, 然后调用构造方法, 创建了一个对象 `p`.

代码 `var p = new Person("jk", 19, "m");` 可以分成三个步骤:

- 执行 `new` 关键字, 开辟内存控件, 闭关获得内存引用
- 调用构造方法 `Person`, 传递参数: 内存引用, 字符串 `"jk"`, 数字 `19`, 和字符串 `"m"` (`m` 表示男, `f` 表示女)
- 执行完构造方法, 将地址引用返回给左边的变量 `p`

## 小结

---

- 所有的对象都继承自 `Object.prototype`
- `Object.prototype` 中常见的方法有:
  - `constructor` 方法
  - `hasOwnProperty` 方法
  - `isPrototypeOf` 方法
  - `propertyIsEnumerable` 方法
  - `toLocaleString` 方法
  - `toString` 方法
  - `valueOf` 方法
- 对象字面量是 `Object` 类型的对象

- `new` 关键字开辟内存空间
- 构造方法调用隐含参数 `this`



# Function 对象

---

主要内容:

- 函数也是对象
- 函数的匿名表达式
- 函数的参数
- 函数名提升
- eval 与 Function

JavaScript 中, 函数是一等公民. 实际上所有的东西都需要依附于函数. 前面已经学习过函数的基本用法, 接下来详细讨论一下函数的更深层次的内容.

## 函数也是一个对象

---

在 JavaScript 中函数也是一种数据类型.

```
function foo() {}  
console.log(typeof foo); // => function
```

也就是说, 函数的地位与对象的地位是一样的, 与数字 `123`, 字符串 `"abc"` 一样, 是有类型的一种数据. 因此也就有了函数表达式的概念 (后面介绍).

## Function 函数

在 JavaScript 中, 有一个 `Function` 函数, 它是所有函数的类型, 每一个函数都是 `Function` 的实例. 定义一个函数, 除了使用 `function` 关键字定义, 同样可以使用 `Function` 来表示, 两种表示方法等价.

定义 `Function` 的语法:

```
var func = new Function(参数1, 参数2, ..., 方法体);
```

在这个语法结构中, 构造函数 `Function` 的参数至少有一个, 参数都是字符串类型. 多个参数分别表示定义函数时的参数与方法体.

例如, 定义一个加法的函数:

```
function func (num1, num2) {  
    return num1 + num2;  
}
```

这个函数有两个参数, 分别是 `num1`, 和 `num2`. 方法体是 `return num1 + num2;`. 因此将其改写为 `Function` 的形式:

```
var func = new Function("num1", "num2", "return num1 + num2;");
```

调用函数直接使用 `func` 即可:

```
console.log(func(1, 2)); // => 3
```

可见使用方法与直接定义函数一样.

练习 改写下面函数为 `Function` 形式

```
// 求最大值  
function max(num1, num2, num3) {  
    var max;  
    if (num1 >= num2) {  
        if (num1 >= num3) {  
            max = num1;  
        } else {  
            max = num3;  
        }  
    }  
}
```

```
    }  
  } else {  
    if (num2 >= num3) {  
      max = num2;  
    } else {  
      max = num3;  
    }  
  }  
  return max;  
}
```

## 自定义函数是 `Function` 的对象

可见 `new Function(...)` 后得到的是一个函数。实际上, 自定义的每一个函数都是一个对象, 是 `Function` 的实例。

在 JavaScript 中要判断一个对象是否为某个构造方法创建出来的, 可以使用 `instanceof` 运算符, 其语法为:

实例 `instanceof` 构造函数

如果实例是被构造方法创建出来的, 该表达式返回 `true`; 如果实例是继承自构造函数创建出来的对象的, 表达式返回 `true`。否则返回 `false`。

```
function foo1() {}  
var o1 = new foo1();  
function foo2() {}  
function foo3() {};  
foo3.prototype = o1; // 设置原型  
var o2 = new foo3(); // o2 继承自 o1  
  
console.log(o2 instanceof foo1); // => true  
console.log(o2 instanceof foo2); // => false  
console.log(o2 instanceof foo3); // => true
```

使用 `instanceof` 运算符, 可以验证自定义函数是否为 `Function` 的实例:

```
function foo1() {}  
function foo2() {}  
console.log(foo1 instanceof Function); // => true  
console.log(foo1 instanceof Array); // => false  
console.log(foo1 instanceof foo2); // => false
```

## 函数的字面量

函数在使用的时候也可以动态的赋值, 使用的便是函数的字面量. 其语法为:

```
function (参数, ...) {  
    函数体;  
}
```

函数的字面量, 和数字字面量, 字符串字面量, 或对象字面量一样, 表示的就是函数本身. 使用的使用可以使用一个变量来接收, 再使用该变量来调用:

```
var foo = function() {  
    console.log("函数表达式");  
}; // 赋值表达式, 需要使用分号结束  
  
foo(); // 调用
```

或者使用自调用函数:

```
(function () {  
    console.log("自调用函数");  
})();
```

注意: 使用函数字面量的时候, 使用的是赋值语句, 因此需要使用分号结尾.

## Function 的原型

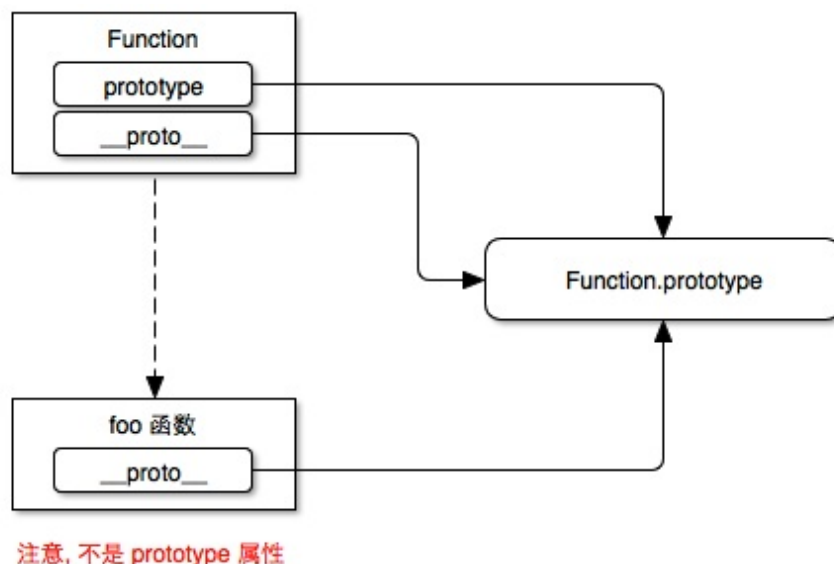
类似于对象和 `Object.prototype` 的关系一样. 自定义函数是 `Function` 的实例. 因此自定义函数的非正式属性 `__proto__` 与 `Function.prototype` 的引用相同:

```
function foo1() {}  
var foo2 = function() {};  
console.log(foo1.__proto__ === Function.prototype); // => true  
console.log(foo2.__proto__ === Function.prototype); // => true
```

因此 `Function.prototype` 就是任意函数的原型. 奇怪的是 `Function` 也是函数, 因此 `Function.__proto__` 与 `Function.prototype` 引用也相等.

```
console.log(Function.__proto__ === Function.prototype);
```

也就是说任意函数都继承自 `Function.prototype` 对象.



因此如果需要给任意一个函数都添加方法, 可以使用下面语法:

```
Function.prototype.showName = function() {  
    console.log(/function\s(.+?)\s/.exec(this.toString())[1]);  
}
```

```
};

function jkmethod() {}
jkmethod.showName();
```

给 `Function.prototype` 添加的任何成员, 都会被继承到任意函数中. 因此在 Douglas 的 *JavaScript: The Good Parts* 中有下面的经典代码 (后面解释):

```
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
} // 任意方法都有 method 方法, 给当前方法的原型添加新方法
```

## 函数参数

函数在调用的时候, 为了更好的复用, 需要给其传递参数. 在本节中主要讨论三个话题: 形参与实参的概念, 没有函数重载, 以及 `arguments` 对象.

## 形参与实参

在函数的术语中有两个概念, 一个是形参一个是实参. 那么怎么理解呢? 首先看看形式计算的概念.

所谓的形式计算, 就是符号计算, 不涉及具体的数据. 取而代之的是符号的规律与逻辑运算符. 例如中学阶段学习的代数就是典型的形式计算. 例如

$$((a + b)^2 = a^2 + 2ab + b^2 \int_a^b f(x) \, dx = \left. F(x) \right|_a^b$$

\$\$

那么将运算可以从具体的数据转到符号逻辑推导中. 将繁复的计算过程简化成几个简单的公式, 然后带入数据完成计算. 这便是形式计算.

那么形参是什么呢? 定义一个函数, 例如打印一个数字

```
function showNum(num) {  
    console.log(num);  
}
```

函数体就像数学符号表达式, 整个逻辑就是基于参数这个符号进行处理的一段逻辑. 打印的是参数 `num`. 也就是说整个函数体都是围绕参数这个符号来执行的逻辑, 不涉及到具体的参与数据 (函数体中定义的常量数据不算), 因此就是一个形式逻辑结构. 所以将函数定义中, 定义的参数称为**形式参数**.

实际参数, 顾名思义就是实际的数据, 即具体参数运算的数据. 将函数定义中的参数称为形参, 将函数调用时传入的参数称为实际参数.

```
function max(num1, num2) { // 参与函数逻辑的符号, num1, num2 是形式参数  
    return num1 > num2 ? num1 : num2;  
}  
var res1 = max(12, 23); // 传入参与运算, 数字 12, 23 是实际参数  
var n1 = 34, n2 = 45;  
var res2 = max(n1, n2); // 传入参与运算, 变量 n1, n2 是实际参数
```

## 没有函数重载

在 C++ 中有函数重载的概念, 所谓的函数重载, 是指允许定义多个函数, 函数名相同, 但是要求参数不同. 那么在 JavaScript 中是不支持的.

```
function foo() {  
    console.log("无参数函数 foo");  
}  
function foo(n) {  
    console.log("一个参数函数, 参数是 " + n);  
}  
function foo(n, m) {  
    console.log("两个参数函数, 参数是 " + n + " 和 " + m);  
}
```

```
foo();  
foo(1);  
foo(1,2);
```

从词法分析的顺序看, 后面的函数会覆盖前面的函数.

### arguments 对象

函数没有重载, 但是很显然, 前面介绍的 `Function` 在调用的时候可以有多个参数. 那么是如何实现的呢? 实际上每一个函数内部都有一个 `arguments` 对象. 它是一个像数组一样的对象, 描述了参数的信息与函数的信息.

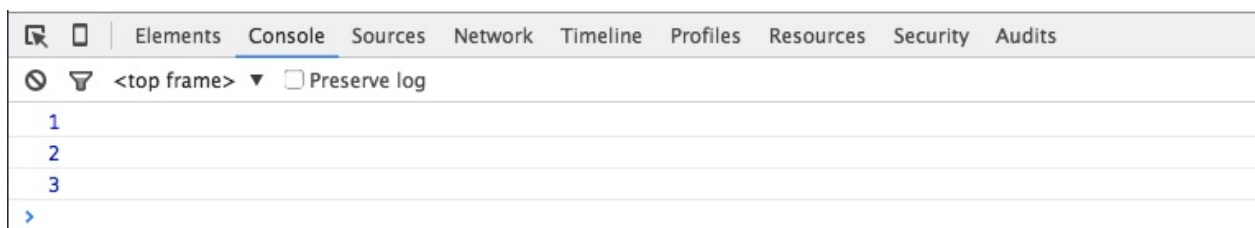
作为对象 `arguments` 有下面属性

- `length` + 索引
- `callee`

`arguments` 是一个像数组的对象, 因此有 `length` 属性, 表示他里面有多少数据. 同时可以使用索引来访问每一个数据:

```
function foo() {  
    for (var i = 0; i < arguments.length; i++) {  
        console.log(arguments[i]);  
    }  
}  
foo(1,2,3);
```

执行结果为



可见, `arguments` 中存储的就是传入的参数. 可以将其看成参数数组 (注意, 是



看成数组).

利用 `arguments` 对象, 可以判断传入的参数是什么, 以及传入的参数个数. 那么根据个数可以做出不同的判断, 完成不同的事情.

将前面的 `foo` 函数进行改写, 既有

```
function foo() {
    var len = arguments.length;
    switch (len) {
        case 0: console.log("无参数函数 foo"); break;
        case 1: console.log("一个参数函数, 参数是 " + arguments[0]); br
        case 2:
            console.log("两个参数函数, 参数是 " +
                arguments[0] +
                " 和 " +
                arguments[1]);
            break;
        default:
            console.log("参数依次是: " + [].join.call(arguments, ", "));
            break;
    }
}

foo();
foo(1);
foo(1, 2);
foo(1, 2, 3, 4, 5, 6);
```

例: 求多个数字的和

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

例: 简单模拟 jQuery 的 `onload` 与 `id` 选择

```
function J() {
    var len = arguments.length,
        arg = arguments[0];
    if (len !== 1) return;
    if (typeof (arg) === "function") {
        window.onload = arg;
    }
    if (typeof (arg) === "string") {
        return document.getElementById(arg);
    }
}
```

练习: 求多个数字的最大值与最小值

`callee` 是当前函数引用. 即在函数 `foo` 内部, `arguments.callee` 就是函数 `foo` 本身.

```
function foo() {
    console.log(foo === arguments.callee);
}
foo(); // => true
```

一般来说用不着该属性, 而且在严格模式中, `callee` 是未定义. 但是使用它, 可以支持匿名函数的递归.

## 函数名提升

函数有定义语法, 也有字面量语法. 虽然几乎是一样的, 但是还是有一点点的异同. 先来看一个案例:

```
var num = 123;
```

```
function foo() {  
    console.log(num);  
    var num = 456;  
    console.log(num);  
}  
foo();
```

原因是, 在 javascript 中, js 引擎在解释代码的时候, 会将所有的变量声明提升到最前面. 因此在函数 `foo` 里面, 代码就会变成:

```
var num = 123;  
function foo() {  
    var num;  
    console.log(num);  
    num = 456;  
    console.log(num);  
}  
foo();
```

打印未初始化的变量, 当然是 `undefined`. 这样虽然变量名定义在下面, 但是感觉好像将变量定义在前面一样. 这个的结论我们成为 "变量名提升". 同样我们说过: 函数也是一中数据类型, 和其他类型一样. 因此存储变量名提升, 也存在函数名提升.

```
var foo = function() {  
    console.log("foo");  
}  
  
var func = function() {  
    foo();  
    var foo = function() {  
        console.log("func.foo");  
    }  
}  
  
func();
```

运行结果是抛出异常. 原因很简单, 和变量名提升一样, 函数名也会提升, 修改一下代码:

```
var foo = function() {
    console.log("foo");
}

var func = function() {
    var foo;
    foo();
    foo = function() {
        console.log("func.foo");
    }
}

func();
```

很明显, `foo` 是 `undefined`, 自然也就不能当做函数来调用. 所以出现错误.

那么如果使用函数的声明语法呢? 那就会正常执行. 原因是在 `js` 引擎解析代码的时候. 首先会将函数的声明语法全部加载一遍, 因此在执行赋值语句的时候, 内存中早已有函数的声明了. 因此调用就不会出错啦.

```
function foo() {
    console.log("foo");
}

function func() {
    foo();
    function foo() {
        console.log("func.foo");
    }
}

func();
```

由于有了这些不同, 因此在实际开发的时候, 推荐将变量都写在开始的地方,

也就是在函数的开头将变量就定义好, 类似于 C 语言的规定一样. 这个在 js 库中也是这么完成的, 如 jQuery 等.

问题: 下面代码执行结果是什么

```
// 1.
var a = 123;
function a() {
    console.log("a function");
}
console.log(a);

// 2.
b();
function b() {
    console.log("b function");
}

// 3.
c();
var c = function () {
    console.log("c function");
};
```

## eval 函数与 Function 函数

在 javascript 有一个函数 `eval` 非常强大. 它可以动态的执行脚本. 将字符串作为参数传入, 然后函数执行的时候就会将这个字符串解析成脚本执行. 例如:

```
console.log(num); // => 报错
```

如果加上 `eval` 代码:

```
eval("var num = 123;");
console.log(num); // => 123;
```

可见 `eval` 的功能非常强大. 当然也有人会认为这个写法完全是浪费表情. 当然实际开发中不会这么使用. 这里只是为了说明它的基本方法而已. 那么在实际开发中可以利用它实现一些较为灵活的功能. 例如绘制函数曲线. 可以根据用户的输入决定函数是什么, 然后再进行绘制. 再比如利用 `ajax` 获得数据. 那么实际开发中一般得到的是 `json` 格式的字符串, 那么完全可以使用该函数将字符串变成对象.

```
var txt = '{name: "蒋坤", course: "JavaScript 面向对象高级"}';  
var o = eval("(" + txt + ")");
```

当然远远这么些, 凡是动态处理的东西都可以利用它来实现.

但是 `eval` 函数也有它的弊端. 因为它会不经意的污染全局变量. 因此一般开发中推荐使用 `Function` 来代替 `eval`.

```
var txt = '{name: "蒋坤", course: "JavaScript 面向对象高级"}';  
var o = (new Function("return " + txt))();
```

## 小结

- 函数也是对象, 是 `Function` 的实例
- `Function` 既是函数, 又是对象
- `Function.prototype` 是函数对象的原型
- `arguments` 对象
- 函数名提升
- 使用 `eval` 和 `Function` 动态的执行脚本

# 作用域链

---

主要内容:

- 词法作用域与块级作用域
- 函数限定作用域
- 作用域链
- 变量名提升与函数名提升

JavaScript 作用域的问题是面试题中常考的一个技巧, 也是开发中常常出现问题的地方. 本节逐步分析作用域的问题, 同时给出一个判断作用域的通用方法.

## 块级作用域的概念

---

在 C 系的编程语言中, 基本上都有块级作用域. 所谓块级作用域, 就是利用代码块来表示一个指定的范围. 在该范围内定义的变量, 只允许在该范围内使用, 该范围外就找不到变量. 而这个范围使用花括号来限定. 例如:

```
{  
    int num = 123;  
}  
printf("num = %d\n", num);
```

这段代码就会报错. 因为 `num` 定义在花括号中, 只允许在该花括号中被访问. 但是在 JavaScript 中就不是这样.

## JavaScript 没有块级作用域

---

在 JavaScript 中, 代码

```
{
```

```
    int num = 123;
  }
  console.log(num);
```

可以正常运行. 那么在 JavaScript 中作用域是如何限定的呢?

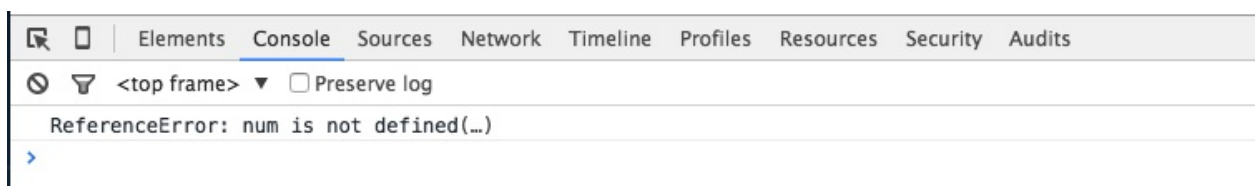
## 函数限定变量作用域

在 JavaScript 中, 只有函数可以限定一个变量的作用范围. 什么意思呢? 就是说, 在 JavaScript 中, 在函数里面定义的变量, 可以在函数里面被访问, 但是在函数外无法访问. 看如下代码:

```
var func = function() {
  var num = 123;
}

try {
  console.log(num);
} catch(e) {
  console.log(e);
}
```

执行结果为:



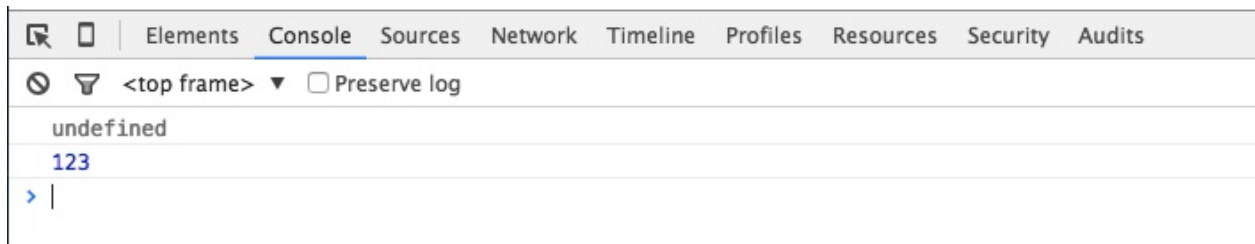
可见与之前块级作用域效果一样报错. 再来看代码:

```
var func = function() {
  console.log(num);
  var num = 123;
  console.log(num);
}
```



```
try {  
    func();  
} catch(e) {  
    console.log(e);  
}
```

运行结果为:



因此, 定义在函数中的变量, 只允许在函数中被访问.

问题: 下面代码会报错吗? 如果不会应该有什么结果? 为什么?

```
function foo() {  
    function func() {  
        console.log(num);  
    }  
    func();  
    var num = 123;  
}  
foo();
```

有了函数作用域的存在, 可以利用自调用函数来模拟块级作用域.

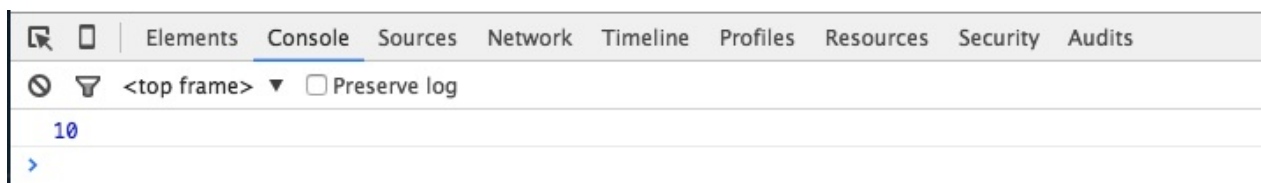
## 子域访问父域

前面介绍了, 函数可以限定变量的作用域, 那么在函数中再定义一个函数, 就是在一个作用域内又有一个作用域, 称为该作用域的子域.

有一个结论: 子域中的代码可以访问到父域中的变量.

```
var func = function() {  
    var num = 10;  
    var sub_func = function() {  
        console.log(num);  
    };  
    sub_func();  
};  
func();
```

看执行结果:



可见 `sub_func` 中可以直接访问 `func` 中的变量. 规律和明显, 函数限定作用域, 只要是在该函数中就可以被访问. 但是子域访问也是有一定条件的, 看下面代码:

```
var func = function() {  
    var num = 10;  
    var sub_func = function() {  
        var num = 20;  
        console.log(num);  
    };  
    sub_func();  
};  
func();
```

运行的结果为 20. 因为在 `sub_func` 的代码中有 `var`. 这个是声明变量, 因此在子域中声明了一个变量 `num`, 那么父域中的变量就会被覆盖掉. 因此打印的结果是 20.

由此可见访问有一定规则可言. 在 JavaScript 中使用变量, JavaScript 解释器首先在当前作用域中搜索是否有该变量的定义. 如果有, 就是用这个变量; 如

果没有就到父域中寻找该变量. 以此类推, 直到最顶级作用域, 仍然没有找到就抛出异常"变量未定义". 看下面代码:

```
(function () {  
    var num = 123;  
    (function () {  
        var num = 456;  
        (function () {  
            console.log(num);  
        })();  
    })();  
})();
```

这里执行, 在当前函数中没有找到 `num`, 因此向上查找, 打印的是 456, 如果将 `var num = 456;` 注释掉, 则继续向上找, 打印 123, 如果将 `var num = 123;` 注释掉, 那么还是向上找, 找到全局作用域, 如果全局作用域未定义变量 `num` 那么就会抛出异常.

## 作用域链

---

有了 JavaScript 的作用域的划分, 那么可以将 JavaScript 的访问作用域连成一个链式树状结构. JavaScript 的作用域链一旦能清晰的了解, 那么对于 JavaScript 的变量与闭包就是非常清晰的了.

下面采用绘图的方法, 绘制出作用域链.

### 绘制规则

1. 作用域链就是对象 ( 变量 ) 的数组
2. 全部的整个 `script` 是 0 级链, 每一个对象占一个位置
3. 凡是看到一个函数, 就延展一个链, 一级级展开
4. 访问首先看当前函数, 如果没有定义往上一级链看
5. 如此往复, 直到 0 级链

### 举例

看下面的代码:

```
var num = 10;
var func1 = function() {
  var num = 20;
  var func2 = function() {
    var num = 30;
    console.log(num);
  };
  func2();
};
var func2 = function() {
  var num = 20;
  var func3 = function() {
    console.log(num);
  };
  func3();
};
func1();
func2();
```

分析一下这段代码:

- 首先整段代码是一个全局作用域, 可以标记为 0 级别链, 那么就有一个数组:

```
var link_0 = [num, func1, func2];
```

- 这里 `func1` 和 `func2` 是两个函数, 因此引出两条 1 级链, 分别为:

```
var link_1 = { func1: [num, func2] };
var link_1 = { func2: [num, func3] };
```

- 第一条 1 级链, 引出 2 级链:

```
var link_2 = { func2: [num] };
```

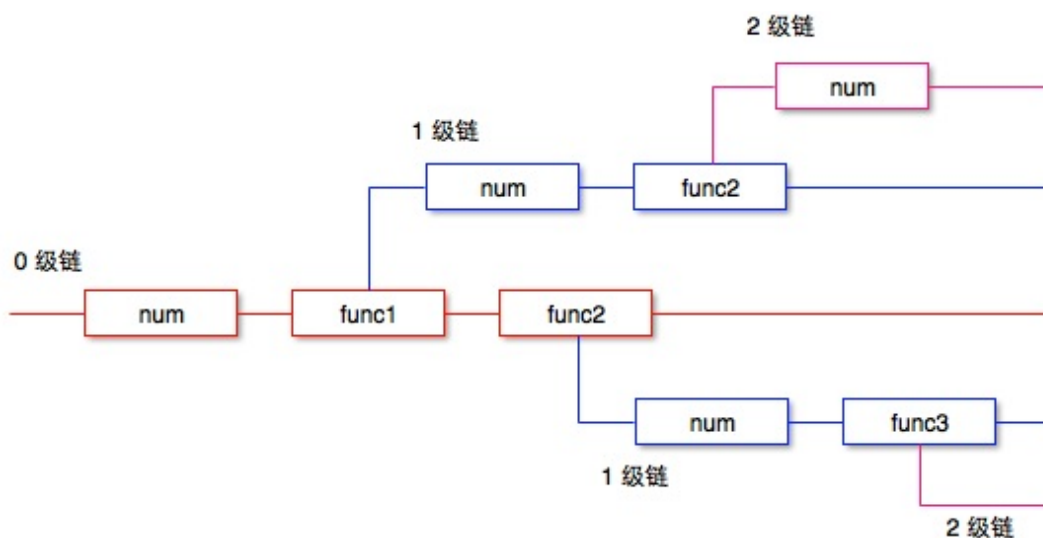
- 第二条 1 级链, 也引出一条 2 级链, 该函数中未定义变量, 因此为空数组:

```
var link_2 = { func3: [] };
```

- 将伪代码整合一下:

```
var link = [      // 0 级链
  num,
  { func1: [      // 1 级链
    num,
    { func2: [num] } // 2 级链
  ] },
  { func2: [      // 1 级链
    num,
    { func3: [] }   // 2 级链
  ] }
];
```

绘成图为:



有了这个作用域链的图, 那么就可以非常清晰的了解访问变量是如何进行的:

在需要使用变量时, 首先在当前的链上寻找变量, 如果找到就直接使用, 不会向上再找; 如果没有找到, 那么就向上一级作用域链寻找, 直到0级作用域链.

如果能非常清晰的确定变量所属的作用域链的级别, 那么在分析 JavaScript 代码与使用闭包等高级 JavaScript 特性的时候就会非常容易.

思考: 下面代码执行结果是什么?

```
if (! "a" in window) {  
    var a = "定义变量";  
}  
console.log(a);
```

## 小结

---

- 块级作用域
- 词法作用域
- 函数定义作用域
- 如何绘制作用域链分析代码

# 闭包的基本概念

---

主要内容:

- 闭包的基本概念
- 闭包的基本形式
- 闭包与内存

一般人认为闭包是一个神秘而又晦涩的内容. 实际上并非如此. 接下来详细说明闭包的概念和闭包的一般的形式, 同时讨论闭包的有点和缺点.

## 闭包的基本概念

---

所谓的闭包, 简单说就是允许低级子链访问高级子链的数据的结构.

在 "作用域链" 一节中介绍过, 高级子链可以直接访问低级子链的数据, 但是反过来无法访问:

```
var num1 = 123;
function foo() {
    var num2 = 456;
    console.log("1 级子链允许访问 0 级子链 " + num1);
}
foo();

try {
    console.log(num2);
} catch (e) {
    console.log("0 级子链无法访问 1 级子链的数据");
}
```

但是如果允许访问这个数据, 那就构成闭包结构.

## 闭包的通俗含义

闭包, 从字面意义上来解释, 就是包裹起来的, 封闭起来的结构. 在 JavaScript 中, 函数就是一个封闭的结构. 在函数中定义的数据, 在外面是访问不到的.

```
(function () {  
    var num = 123,  
        obj = {},  
        arr = [],  
        isTrue = false,  
        str = "abc",  
        foo = function() {};  
})();
```

在函数外面是绝对无法访问里面定义的数据的. 因此它就构成一个封闭的结构. 但是没有任何作用, 因此在 JavaScript 中需要将其对外开放一些接口, 以便使用这些数据. 这就是要求函数返回一个数据, 好么函数要么对象.

## 闭包的基本结构

要在函数外访问函数内的数据, 最简单的就是返回一个函数:

```
function foo() {  
    var num = 123;  
    return function() {  
        return num;  
    }  
}
```

当函数返回一个函数的时候, 返回的函数允许返回一个数据. 那么在函数外就可以访问函数内的数据了. 但是仅仅是访问这个数据, 有时需要对其进行修改. 因此既要求获得数据, 也要求可以修改数据. 因此就需要两个函数, 那可以返回一个对象:



```
function foo() {  
    var num = 123;  
    return {  
        get_Num: function() {  
            return num;  
        },  
        set_Num: function(value) {  
            num = value;  
        }  
    }  
}  
var o = foo();  
console.log(o.get_Num()); // => 123  
o.set_Num(456);  
console.log(o.get_Num()); // => 456
```

小结一下闭包的这个结构:

- 首先有一个函数, 在函数中定义数据
- 同时返回一个函数, 允许访问函数内部的数据
- 调用函数, 返回允许访问内部数据的函数
- 利用返回的函数, 允许在外界访问函数内定义的数据

## 闭包的基本案例

---

有了闭包, 可以做到事情就变得非常多了. 下面通过几个案例来说明闭包的使用

## 使用闭包模拟私有变量

```
// v3 使用方法, v5 可以直接使用 getter  
var person = function(name, age, gender) {  
    return {  
        get name() {  
            return name;  
        },  
    },  
};
```

```
        get age() {
            return age;
        },
        get gender() {
            return gender;
        }
    }
};

var obj = person("jk", 19, "male");

console.log(obj.name);
console.log(obj.age);
console.log(obj.gender);
```

## 使用闭包模拟块级作用域

```
var sum = 0;
(function () {
    for (var i = 0; i <= 100; i++) {
        sum += i;
    }
})();
console.log(sum);
```

## 使用闭包创建唯一标识符

```
var uniqueInteger = (function () {
    var count = 0;
    return function() {
        return count++;
    };
})();
```

## 闭包与内存

使用闭包可以非常方便的组织代码, 将不需要对外公开的代码都封装起来, 保护起来. 使得代码更加紧凑, 维护更加方便. 但是使用闭包, 保存了私有数据, 容易造成内存的泄露.

为了更加方便的解释这个问题, 首先讨论一下 JavaScript 中内存的管理.

## 内存管理

JavaScript 具有垃圾收集的功能. 也就是说当执行环境会负责管理代码的执行过程使用的内存, 包括内存的分配与释放. 在传统的程序设计中, 例如 C, C++, 需要手动的管理内存, 申请, 释放都需要手动的处理. 因此容易出现问题. 在 JavaScript 中不需要这么麻烦, 所有的事情都由 JavaScript 引擎去完成.

常见的内存管理方式有两种: 标记式, 引用计数式.

使用标记式内存管理, 运行时在内存中会标记所有的不再使用的数据, 进而加以回收. 如今大部分浏览器所采用的是标记式内存管理. 只是进行内存回收的时间不同而已.

使用引用计数式内存管理, 运行时会给每一个对象加上编号, 凡是有变量指向该对象, 引用计数就 +1, 凡是减少一个引用, 计数就 -1. 那么当引用计数为 0 的时候, 回收内存. 但是一旦出现循环引用, 就容易造成内存泄露.

## 内存管理

在 JavaScript 中不需要自己考虑内存维护的问题, 所有的内存维护都可以由 JavaScript 引擎去处理. 但是为了更好的提升性能, 建议在不使用数据的时候, 将其设置为 `null`.

## 小结

---

- 闭包的基本概念
- 闭包的实现方式
- 闭包案例

- 内存简介

## getter 与 setter

---

主要内容:

- 对象特性
- getter 与 setter

在 ES5 中加入了对对象的特性, 本节就读写器特性加一说明.

## 读写方法

---

我们知道, 对象就是键值对 (函数就是数据). 但是访问对象数据的时候有时需要通过计算得到. 例如, 利用闭包模拟私有数据的时候:

```
var createPerson = function(name) {  
    return {  
        get_Name: function() {  
            return name;  
        },  
        set_Name: function(value) {  
            name = value;  
        }  
    };  
};  
var p = createPerson("jk");  
console.log(p.get_Name());  
p.set_Name("蒋坤");  
console.log(p.get_Name());
```

在这段代码中是早期的一些实现方式, 可以看到主要是为了访问对象的 `name` 并对其做修改. 但是在使用的时候并不是直接当做属性在使用, 而是作为方法进行调用.

程序员很强大, 希望对于这个操作可以像直接操作变量一样, 访问数据, 不再

使用 `p.get_Name()`，而是直接使用 `p.name`；修改数据不再使用 `p.set_Name("蒋坤")`，而是直接使用 `p.name = "蒋坤"`。有朋友就说了。那直接定义成属性不就完了。

so, 问题来了。如果是只允许读, 而不允许修改呢? 再或者有一个 `age` 属性, 要求不允许输入负数怎么办呢? ...

这样的问题很多, 很明显如果有一些限制, 那么就无法使用属性, 必须使用函数。因此在 JavaScript 中引入了一个语法糖, `getter` 与 `setter` 读写器。

## 基本语法

首先要实现该功能的基本语法为:

```
var obj = {  
    变量:值,  
    set 属性名 (参数) {  
        setter 方法体  
    },  
    get 属性名 () {  
        getter 方法体  
    }  
};
```

注意, 这里使用的语法与之前的语法不太一样。这里使用 `get` 和 `set` 引导。没有使用 `function` 关键字, 但是创建的依旧是函数 (方法)。这里没有冒号, 直接在名字后面跟上圆括号和方法体。这里名字可以随意命名, 但是一般与需要约束的变量同名。 `set` 方法参数名是随意的, 在赋值的时候, 就会调用 `setter` 方法。并将被赋值作为参数而是用。如果在读取数据的时候, 则会调用 `getter` 方法, 从而返回数据。

将前面的案例进行改写:

```
var createPerson = function(name) {
```

```
    return {
      get name () {
        return name;
      },
      set name (value) {
        name = value;
      }
    };
  };
  var p = createPerson("jk");
  console.log(p.name);
  p.name = "蒋坤";
  console.log(p.name);
```

运行结果为:



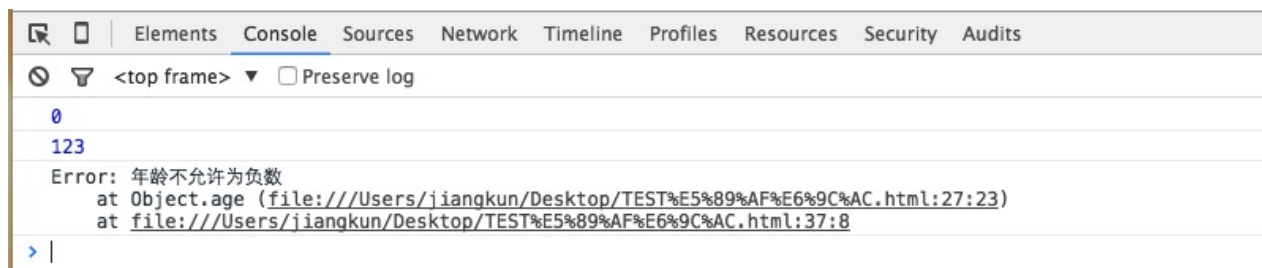
看起来书不是很简单. 这样其实和调用方法是一样的, 只是使用起来更加舒服与方便.

同理, 我们假设给年龄设置数值, 但是不允许赋值为负数, 如果为负数就抛出一个错误:

```
var createPerson = function(age) {
  return {
    get age () {
      return age;
    },
    set age (value) {
      if (value > 0) {
        age = value;
      } else {
        throw new Error("年龄不允许为负数");
      }
    }
  }
}
```

```
    };  
};  
var p = createPerson(0);  
console.log(p.age);  
p.age = 123;  
console.log(p.age);  
try {  
    p.age = -1;  
    console.log(p.age);  
} catch (e) {  
    console.log(e);  
}
```

执行结果为：



## 小结

- 对象读写属性的基本语法



# 面向对象高级

---

介绍原型链 与 基本模式 和 一些使用技巧

- 原型链
- 对象的创建模式
- 继承的方式
- 函数的四中调用模式
- 闭包案例
- 正则表达式对象
- XMLHttpRequest 对象

# 原型链

---

主要内容:

- 对象都有原型
- `Object` 对象的原型
- 函数对象的原型
- 原型链
- 属性搜索原则

前面已经介绍过原型与一般对象的关系, 函数与 `Function` 等的关系了. 可以发现一般的对象, 构造函数, `Function`, 和 `Object` 等都是有一定内在联系的. 接下来我们详细讨论它们之间的联系, 给出一个完整的描述.

**注意, 本节内容会比较绕. 但是非常重要.**

## 复习前面的内容

---

前面介绍的内容简单来说就是:

- 自定义对象继承自构造函数的原型
- 构造函数继承自 `Function.prototype`
- `Function.prototype` 作为对象继承自 `Object.prototype`
- `Function` 和 `Object` 作为函数继承自 `Function.prototype`

接下来我们将其局部的逻辑进行展开.

## 自定义对象继承自构造函数的原型

根据前面的知识, 分析下面代码:

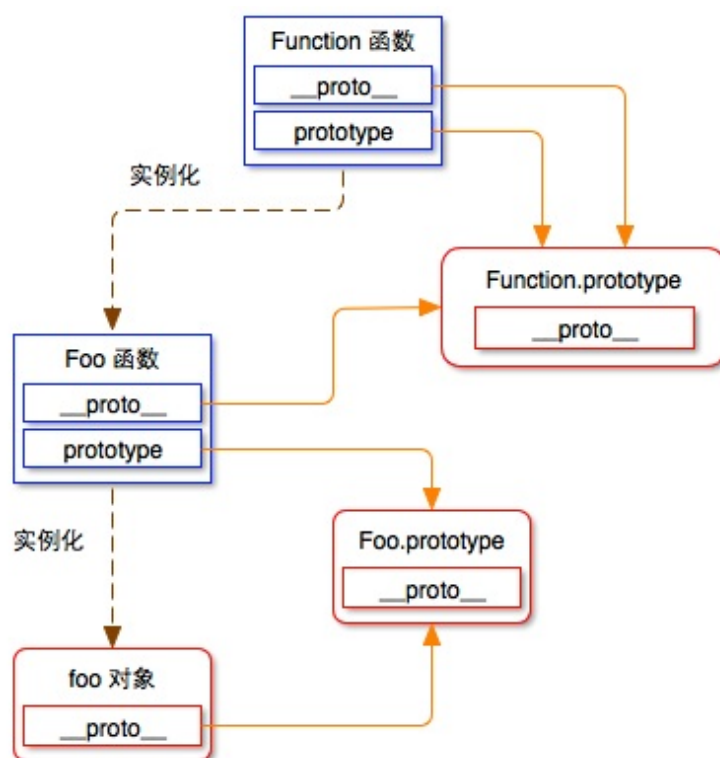
```
function Foo() {}
```

```
var foo = new Foo();
```

简单分析一下:

- 首先 `Foo` 是构造函数. 那么存在对象 `Foo.prototype` .
- 同时 `Foo` 继承自 `Function.prototype` , 是函数 `Function` 的实例.
- 然后利用 `Foo` 创建对象 `foo` , `foo` 继承自 `Foo.prototype` .

利用图绘制出逻辑:



## Object 的实例继承自 Object.prototype

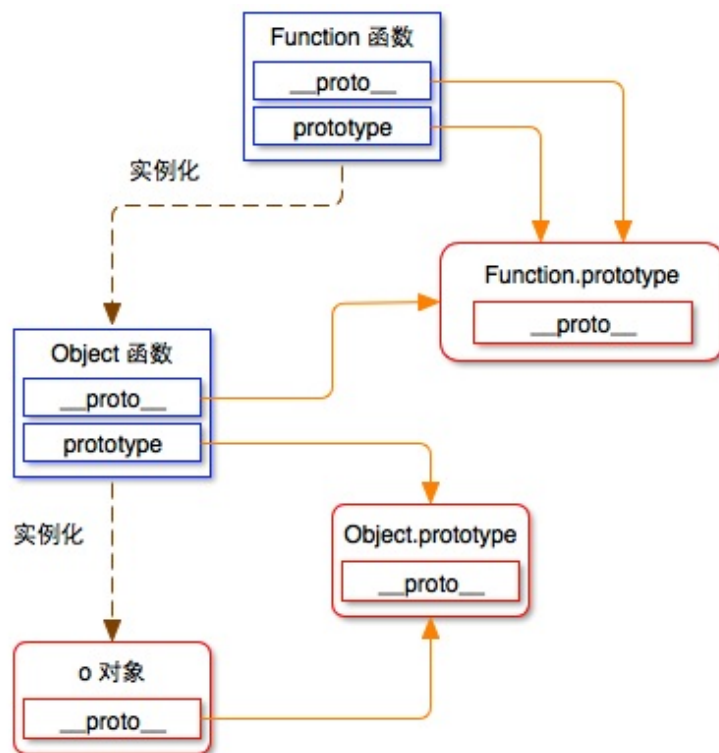
再来看一个特殊的构造函数 `Object` . 分析由它创建处理的对象继承结构. 看下面代码:

```
var o = new Object();
```

简单分析一下:

- 构造函数 `Object` 是 `Function` 的实例, 继承自 `Function.prototype` .
- 实例 `o` 继承自 `Object.prototype` .

利用图绘制出逻辑:

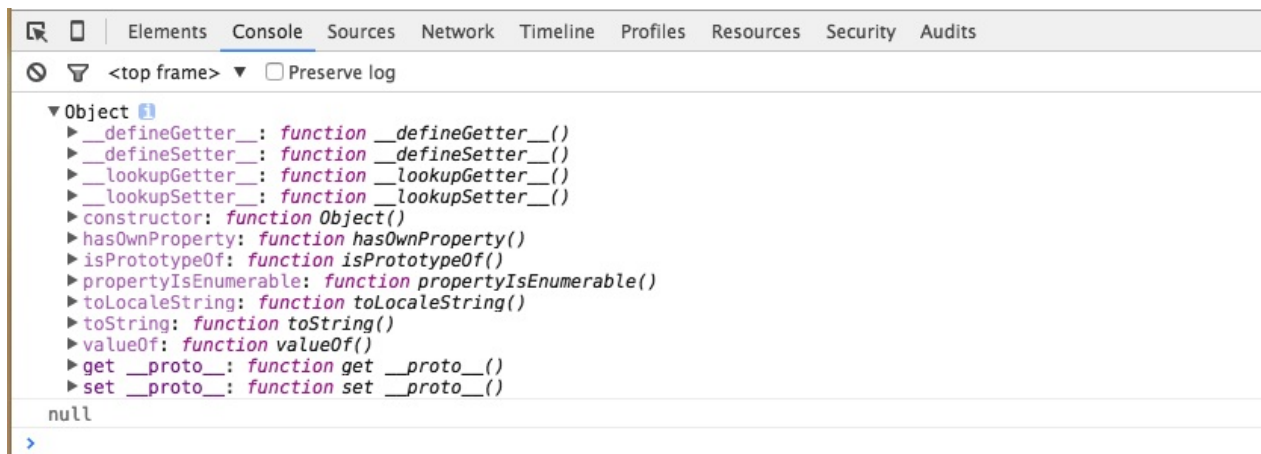


## 原型继承自谁？

可以发现两段结构逻辑似乎是一样的, 但是仔细观察, 对象的原型对象并不相同. 那么问题是原型作为一个对象, 继承自谁呢？

```
function Foo() {}
var foo = new Foo();
var o = new Object();
console.log(Foo.prototype.__proto__);
console.log(Object.prototype.__proto__);
```

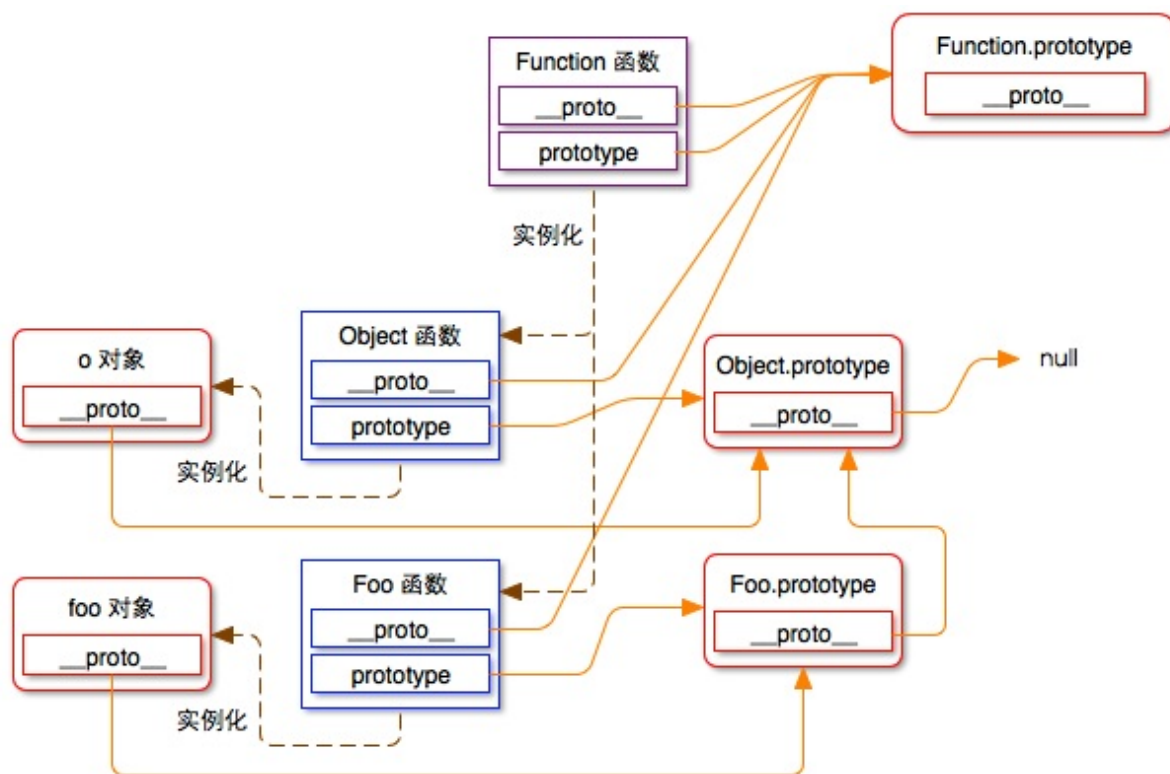
执行结果为:



可见 `Object.prototype` 作为对象, 其 `__proto__` 为 `null`. 意味着找不到其原型, 它是最根本的对象. 对象 `foo` 的原型对象 `Foo.prototype` 就是 `Object.prototype`.

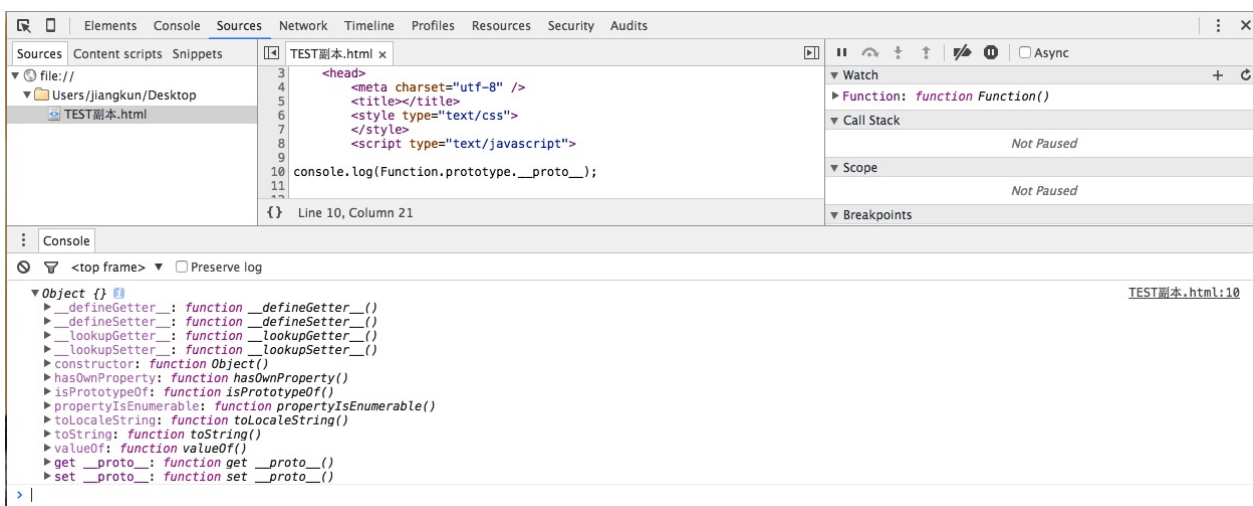
```
console.log(Foo.prototype.__proto__ === Object.prototype); // => true
```

整合两张逻辑结构:



比较复杂, 但是慢慢还是理得清楚. 接下来还剩下一个问题,

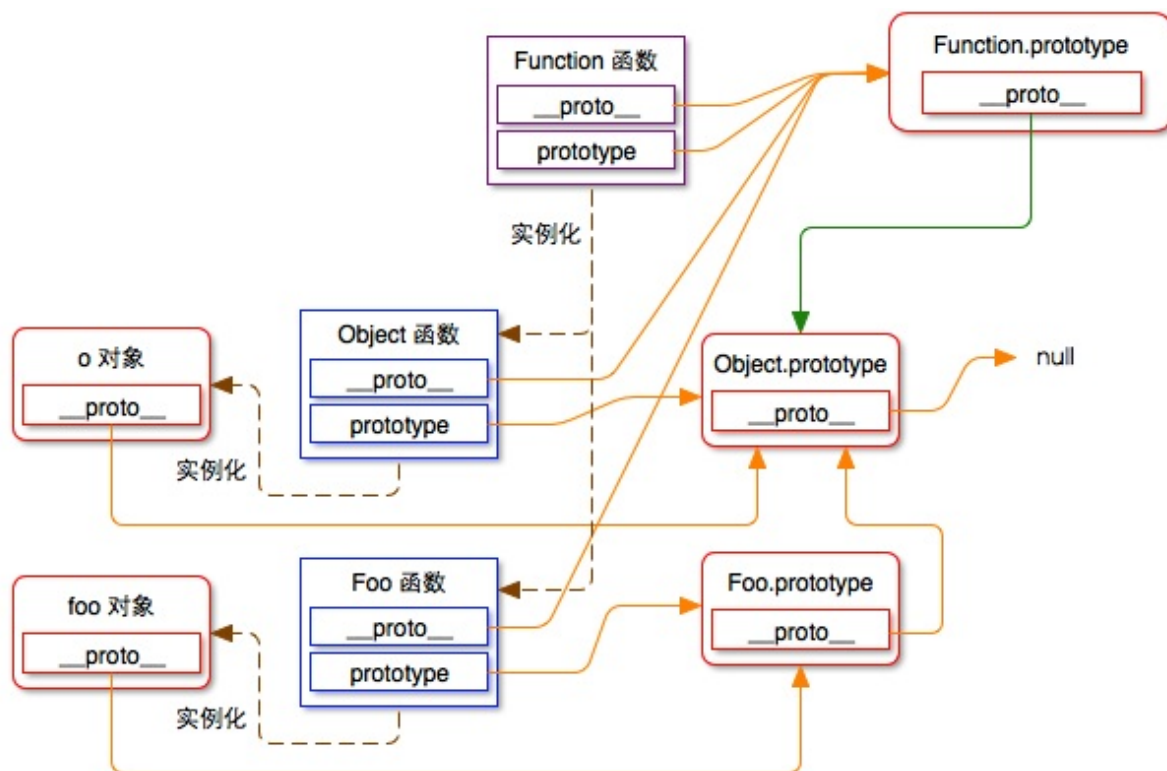
`Function.prototype` 作为对象, 它的 `__proto__` 指向谁呢? 打印出来查看:



可见, `Function.prototype` 的原型对象就是 `Object.prototype` :

```
console.log(Function.prototype.__proto__ === Object.prototype); // =>
```

整理一下前面的图片, 可以得到:

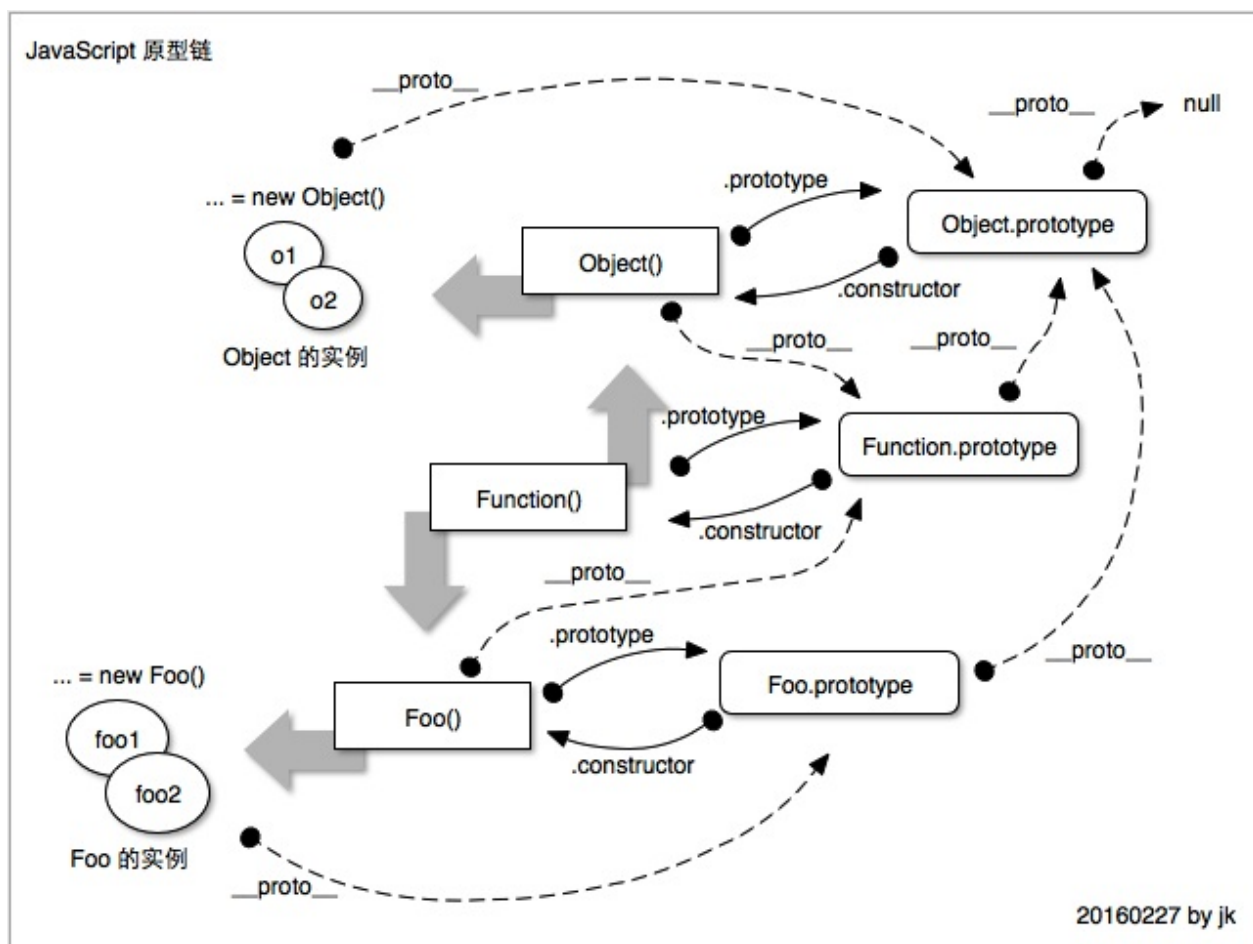


根据前面的一步一步分析, 可以发现在 JavaScript 中所有的数据之间都有联

系, 是一个来源于 `Object.prototype` 的树结构.

## 原型链

由前面的分析, 可以发现在 JavaScript 中, 任何一个数据都来源于 `Object.prototype`. 因此任何一个对象到 `Object.prototype` 都可以找到一条线索. 我们称这条线索就是原型链. 将前面的图再调整一下位置:



那么访问对象数据的时候就会对原型链进行搜索.

## 属性搜索原则

接下来我们来看属性搜索原则. 先来看这一段代码:

```
function Foo() {  
  this.num = 123;  
}
```

```
}  
Foo.prototype.num = 456;  
var foo = new Foo();  
console.log(foo.num);
```

这一段代码执行结果是什么呢？要解决这个问题，我们需要知道属性是怎么访问的。

在 JavaScript 中，如果访问对象的属性，首先会检索当前对象中是否存在属性，如果存在直接调用。如果不存在则会往其原型上找，如果原型中也没有该属性，那么继续往原型的原型中去找。直到找到，或者直到全局作用域。如果找到，直接获得该结果，如果没有找到则返回 `undefined`。

因此这一段代码结果是 `123`。

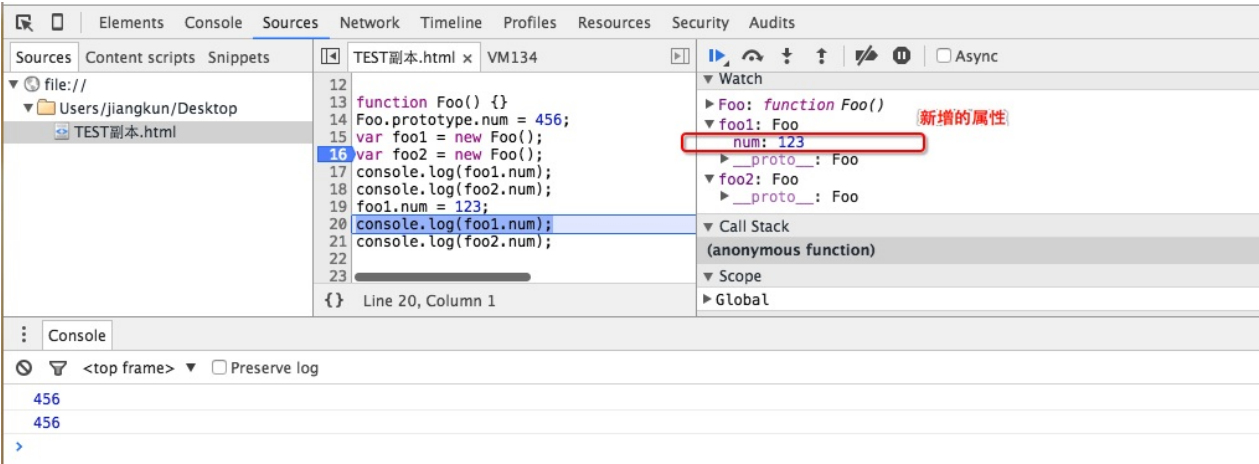
那么问题来了。根据原型继承的原则，如果原型中的数据被一个对象修改了，那么另一个对象会得到多少呢？

```
function Foo() {}  
Foo.prototype.num = 456;  
var foo1 = new Foo();  
var foo2 = new Foo();  
console.log(foo1.num);  
console.log(foo2.num);  
foo1.num = 123;  
console.log(foo1.num);  
console.log(foo2.num);
```

这里运行的结果又是多少呢？从原型继承的角度看，原型中的数据应该被 `foo1` 所修改，因此结果应该是 `123`，但是并非如此。此时修改的只有 `foo1` 的数据，原型中的数据并没有被修改。这是为什么呢？

实际上，在 JavaScript 中，访问属性的时候有一个原则。如果是读取数据，那么按照属性搜索原则就可以获得数据。但是如果是修改数据，那么就会在当前作用域上创建一个该名字的属性，并对其赋值。并非去修改原型中的数据。





## 小结

- 自定义对象的原型逻辑
- 函数的原型逻辑
- 原型链的基本结构
- 属性搜索原则

## 函数的四中调用模式

---

- 函数的四中调用模式
- `this` 的含义
- 四中调用模式的异同
- 构造函数的返回值
- 上下文调用

了解函数的调用过程有助于深入学习与分析 JavaScript 代码. 接下来主要介绍 JavaScript 中函数的四种使用形式.

在 JavaScript 中, 函数是一等公民, 函数在 JavaScript 中是一个数据类型, 而非像 C# 或其他描述性语言那样仅作为一个模块来使用. 函数有四种调用模式, 分别是: 函数调用形式, 方法调用形式, 构造器形式, 以及 `apply` 形式. 这里所有的调用模式中, 最主要的区别在于关键字 `this` 的意义. 下面分别介绍这个几种调用形式.

## 函数调用形式

---

函数调用形式是最常见的形式, 也是最好理解的形式. 所谓函数形式就是一般声明函数 后直接调用即是. 例如:

```
function func() {  
    console.log("Hello JK!");  
}  
func();
```

或者

```
var func = function () {  
    console.log("蒋坤, JavaScript 高级");  
};
```

```
func();
```

这两段代码都会在浏览器的控制台中显示字符串中的文字. 这个就是函数调用.

可以发现函数调用很简单, 就是平时学习的一样. 这里的关键是, 在函数调用模式中, 函数里的 `this` 关键字指全局对象, 如果在浏览器中就是 `window` 对象. 例如:

```
var func = function () {  
    console.log(this);  
};  
func();
```

此时执行结果为:



## 方法调用模式

函数调用模式很简单, 是最基本的调用方式. 但是同样的是函数, 将其赋值给一个对象的成员以后, 就不一样了. 将函数赋值给对象的成员后, 那么这个就不在称为函数, 而应该叫做方法. 例如:

```
var func = function () {  
    console.log("我是一个函数么?");  
};  
var o = {};  
o.fn = func; // 注意没有圆括号  
o.fn(); // 调用
```

此时, `o.fn` 则是方法, 不是函数了. 实际上 `fn` 的方法体与 `func` 是一模一

样的, 但是这里有个微妙的不同. 看下面的代码:

```
// 接上面的代码
console.log(o.fn === func);
```

打印结果是 `true`, 表明两个函数是一样的东西. 但是修改一下函数的代码, 细微的不同就有了:

```
var func = function () {
    console.log(this);
};
var o = {};
o.fn = func;
console.log(o.fn === func); // 比较
func(); // 调用
o.fn(); // 调用
```

执行结果为:



这里的运行结果是, 两个函数是相同的, 因此打印结果是 `true`. 但是由于两个函数的调用是不一样的, `func` 的调用, 打印的是 `Window` 对象, 而 `o.fn` 的打印结果是 `Object` 对象.

这里便是函数调用与方法调用的区别. 函数调用中, `this` 专指全局对象 `window`, 而在方法中 `this` 专指当前对象. 即 `o.fn` 中的 `this` 指的就是对象 `o`.

## 构造器调用模式

同样是函数, 在单纯的函数模式下, `this` 表示 `window`; 在对象方法模式下,

`this` 指的是当前对象. 除了这两种情况, JavaScript 中函数还可以是构造器. 将函数作为构造器 来使用的语法就是在函数调用前面加上一个 `new` 关键字. 如代码:

```
// 定义一个构造函数
var Person = function () {
    this.name = "蒋坤";
    this.sayHello = function () {
        console.log("你好, 我是 " + this.name);
    };
};
// 调用构造函数, 创建对象
var p = new Person();
// 使用对象
p.sayHello();
```

上面的案例首先创建一个构造函数 `Person`, 然后使用构造函数创建对象 `p`. 这里使用 `new` 语法. 然后在使用对象调用 `sayHello()` 方法. 这个使用构造函数创建对象的案例比较简单. 从案例可以看到, 此时 `this` 指的是对象本身.

除了上面简单的使用以外, 函数作为构造器还有几个变化. 分别为:

1. 所有需要由对象使用的属性, 必须使用 `this` 引导
2. 函数的 `return` 语句意义被改写, 如果返回非对象, 就返回 `this`

## 构造器中的 `this`

我们需要分析创建对象的过程, 方能知道 `this` 的意义. 如下面代码:

```
var Person = function () {
    this.name = "jk";
};
var p = new Person();
```

这里首先定义了函数 `Person`, 下面分析一下整个执行:

1. 程序在执行到这一句的时候, 不会执行函数体, 因此 JavaScript 的解释器并不知道这个函数的内容
2. 接下来执行 `new` 关键字, 创建对象, 解释器开辟内存, 得到对象的引用, 将新对象的引用交给函数
3. 紧接着执行函数, 将传过来的对象引用交给 `this`. 也就是说, 在构造方法中, `this` 就是刚刚被 `new` 创建出来的对象
4. 然后为 `this` 添加成员, 也就是为对象添加成员
5. 最后函数结束, 返回 `this`, 将 `this` 交给左边的变量

分析过构造函数的执行以后, 可以得到, 构造函数中的 `this` 就是当前对象.

## 构造器中的 `return`

在构造函数中 `return` 的意义发生了变化, 首先如果在构造函数中, 如果返回的是一个对象, 那么就保留原意. 如果返回的是非对象, 比如数字, 布尔和字符串, 那么就返回 `this`, 如果没有 `return` 语句, 那么也返回 `this`. 看下面代码:

```
var ctr = function () {  
    this.name = "jk";  
    return {  
        name: "蒋坤"  
    };  
};  
var p = new ctr();  
console.log(p.name);
```

执行代码, 这里打印的结果是 "蒋坤". 因为构造方法中返回的是一个对象, 那么保留 `return` 的意义, 返回内容为 `return` 后面的对象. 再看下面代码:

```
var ctr = function () {  
    this.name = "jk";  
    return "蒋坤";  
};  
var p = new ctr();
```

```
console.log(p);  
console.log(p.name);
```

执行结果为:



代码运行结果是, 先弹打印 `ctr {name: "jk"}`, 然后打印 `"jk"`. 因为这里 `return` 的是一个字符串, 属于基本类型, 那么这里的 `return` 语句无效, 返回的是 `this` 对象. 因此第一个打印的是 `ctr` 对象, 而第二个不会打印 `undefined`.

## Apply 调用模式

除了上述三种调用模式以外, 函数作为对象还有 `apply` 方法与 `call` 方法可以使用. 这便是第四种调用模式, 我称其为 `apply` 模式.

首先介绍 `apply` 模式, 首先这里 `apply` 模式既可以像函数一样使用, 也可以像方法一样使用. 可以说是一个灵活的使用方法. 首先看看语法:

```
函数名.apply(对象, 参数数组);
```

这里看语法比较晦涩, 还是使用案例来说明:

- 新建两个 `js` 文件, 分别为 `"js1.js"` 与 `"js2.js"`
- 添加代码

```
// js1.js 文件  
var func1 = function () {  
    this.name = "jk";  
};  
func1.apply(null);
```

```
console.log(name);

// js2.js 文件
var func2 = function () {
    this.name = "jk";
};
var o = {};
func2.apply(o);
console.log(o.name);
```

- 分别运行着两段代码, 可以发现第一个文件中的 `name` 属性已经加载到全局对象 `window` 中. 而第二个文件中的 `name` 属性是在传入的对象 `o` 中. 即第一个相当于函数调用, 第二个相当于方法调用

这里的参数是方法本身所带的参数, 但是需要用数组的形式存储在. 比如代码:

```
var arr1 = [ 1, 2, 3, [ 4, 5 ], [ 6, 7, 8 ] ];
// 将数组扁平化
var arr2 = arr1.concat.apply([], arr1);
```

然后介绍一下 `call` 模式. `call` 模式与 `apply` 模式最大的不同在于 `call` 中的参数不用数组. 看下面代码就清楚了:

```
// 定义方法
var func = function (name, age, gender) {
    this.name = name;
    this.age = age;
    this.gender = gender;
};
// 创建对象
var o1 = {}, o2 = {};
// 给对象添加成员
// apply 模式
var p1 = func.apply(o1, ["jk", 19, "male"]);
// call 模式
var p2 = func.call(o2, "baby", 20, "female");
```



上面的代码, `apply` 模式与 `call` 模式的结果是一样的.

实际上, 使用 `apply` 模式和 `call` 模式, 可以任意的操作控制 `this` 的意义, 在函数 `js` 的设计模式中使用广泛. 简单小结一下, `js` 中的函数调用有四种模式, 分别是: 函数式, 方法式, 构造器式和 `apply` 式. 而这些模式中, `this` 的含义分别为: 在函数中 `this` 是全局对象 `window`, 在方法中 `this` 指当前对象, 在构造函数中 `this` 是被创建的对象, 在 `apply` 模式中 `this` 可以随意的指定. 在 `apply` 模式中如果使用 `null`, 就是函数模式, 如果使用对象, 就是方法模式.

## 案例

下面通过一个案例结束本篇吧. 案例说明: 有一个 `div`, `id` 为 `dv`, 鼠标移到上面去高度增大 2 倍, 鼠标离开恢复. 下面直接上 `js` 代码:

```
var dv = document.getElementById("dv");
var height = parseInt(dv.style.height || dv.offsetHeight);
var intervalId;
dv.onmouseover = function() {
    // 停止已经在执行的动画
    clearInterval(intervalId);
    // 得到目标高度
    var toHeight = height * 2;
    // 获得当前对象
    var that = this;
    // 开器计时器, 缓慢变化
    intervalId = setInterval(function() {
        // 得到现在的高度
        var height = parseInt(dv.style.height || dv.offsetHeight);
        // 记录每次需要变化的步长
        var h = Math.ceil(Math.abs(height - toHeight) / 10);
        // 判断变化, 如果步长为0就停止计时器
        if( h > 0 ) {
            // 这里为什么要用that呢? 思考一下吧
            that.style.height = (height + h) + "px";
        } else {
            clearInterval(intervalId);
        }
    }, 20);
}
```

```
};
dv.onmouseout = function() {
    // 原理和之前一样
    clearInterval(intervalId);
    var toHeight = height;
    var that = this;
    intervalId = setInterval(function() {
        var height = parseInt(dv.style.height || dv.offsetHeight);
        var h = Math.ceil(Math.abs(height - toHeight) / 10);
        if( h > 0 ) {
            that.style.height = (height - h) + "px";
        } else {
            clearInterval(intervalId);
        }
    }, 20);
};
```

## 小结

---

- 函数的四中调用方式
- `this` 的含义
- `apply` 与 `call` 模式的区别

# 对象的创建模式

---

主要内容:

- 使用附加属性创建对象
- 使用构造方法创建对象
- 使用字面量创建对象
- 使用原型创建对象
- 使用工厂方法创建对象
- 使用混合模式创建对象
- 使用寄生模式创建对象

接下来我们将对象的创建方式加一总结

## 使用附加属性创建对象

---

```
var o = new Object();
o.name = "jk";
o.age = 19;
o.gender = "男";
o.sayHello = function () {
    console.log("name = " + o.name +
                ", age = " + o.age +
                ", gender = " + o.gender);
};
// 调用
o.sayHello();
```

## 使用构造方法创建对象

---

```
var Person = function (name, age, gender) {
    this.name = name;
```

```
this.age = age;
this.gender = gender;
this.sayHello = function () {
    console.log("name = " + o.name +
        ", age = " + o.age +
        ", gender = " + o.gender);
};
};
var o = new Person("jk", 19, "男");
o.sayHello();
```

## 使用字面量创建对象

---

```
var o = {
    name: "jk",
    age: 19,
    gender: "男"
    sayHello: function () {
        console.log("name = " + o.name +
            ", age = " + o.age +
            ", gender = " + o.gender);
    }
};
// 调用
o.sayHello();
```

## 使用原型创建对象

---

```
var Person = function() {};
Person.prototype.name = "jk";
Person.prototype.age = 19;
Person.prototype.gender = "男";
Person.prototype.sayHello = function () {
    console.log("name = " + o.name +
        ", age = " + o.age +
        ", gender = " + o.gender);
};
```

```
};  
var o = new Person();  
o.sayHello();
```

## 使用工厂方法创建对象

---

```
var createPerson = function (name, age, gender) {  
    var o = new Object();  
    o.name = name;  
    o.age = age;  
    o.gender = gender;  
    o.sayHello = function () {  
        console.log("name = " + o.name +  
            ", age = " + o.age +  
            ", gender = " + o.gender);  
    };  
    return o;  
};  
var o = createPerson("jk", 19, "男");  
o.sayHello();
```

- 使用混合模式创建对象

```
var Person = function (name, age, gender) {  
    this.name = name;  
    this.age = age;  
    this.gender = gender;  
};  
Person.prototype.sayHello = function () {  
    console.log("Hello, 我是 " + this.name);  
};  
var o = new Person("jk", 19, "男");  
o.sayHello();
```

## 使用寄生模式创建对象

---

```
var Person = function (name, age, gender) {  
    var o = new Object();  
    o.name = name;  
    o.age = age;  
    o.gender = gender;  
    o.sayHello = function () {  
        console.log("name = " + o.name +  
            ", age = " + o.age +  
            ", gender = " + o.gender);  
    };  
    return o;  
};  
var o = new Person();  
o.sayHello();
```

## 小结

---

整理各种创建对象的方式

## 继承的方式

---

主要内容:

- 继承的一般模式
- 原型式继承
- 借用构造函数继承
- 组合式继承
- 寄生式继承

下面是对继承方式的一个总结

### 继承的一般模式

---

```
var baseObj = {
    sayHello: function () {
        console.log("Hello, I'm JK!");
    }
};
var Person = function() {
    this.name = "蒋坤";
};
Person.prototype = baseObj;

var subObj = new Person();
```

- 原型式继承

```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}
```

- 借用构造函数继承

```
function BaseType () {  
    this.name = "jk";  
}  
function SubType () {  
    BaseType.call(this);  
    this.age = 19;  
}
```

- 组合式继承

```
function BaseType (name) {  
    this.name = name;  
}  
BaseType.prototype.sayHello = function () {  
    console.log("Hello, 我是 " + this.name);  
};  
function SubType (name, age) {  
    BaseType.call(this, name); // 继承属性  
    this.age = age;  
}  
SubType.prototype = new BaseType(); // 继承方法
```

- 寄生式继承

```
function createObject (baseObj) {  
    var temp = object(baseObj);  
    temp.sayHello = function () {  
        console.log("hello, i'm jk");  
    };  
    return temp;  
}
```

## 小结



## 整理常见的继承方式

## 闭包案例

---

主要内容:

- 利用闭包实现私有变量
- 利用闭包实现缓存机制

## 正则表达式对象 (略)

---

主要内容:

- 正则表达式的意义
- 元字符
- 匹配判断
- 匹配提取
- 提取组
- 替换

## XMLHttpRequest 对象(略)

---