图解 JavaScript 原型链

邵励治
一辈子从事创造性工作，工夫在诗外

一、理解原型链，要明白的几个概念

概念 1：JavaScript 中一切都是对象

概念 2：JavaScript 中函数也是对象，但对象不一定是函数

概念 3：JavaScript 中对象有三种创建方式：{}、new、Object.create()

> 注意，Object.create() 是一个强大的特性，因为它使得任意对象可以继承，我们在此先标记下它，不要错过

概念 4：JavaScript 中对象有两个属性 __proto__ 与 prototype，它们将构建原型链

概念 5：JavaScript 中只有查询属性时，才会体会到继承的存在，而设置属性与继承无关

概念 6：如果两个对象都从同一个对象上继承了属性，它们便被成为同一个类的实例

二、构造函数

「构造函数」是用来初始化新创建的对象的。

使用关键字 new 来调用构造函数会自动创建一个新对象，因此构造函数本身只需要初始化这个新对象的状态即可。

调用构造函数的一个重要特征是，构造函数的 prototype 属性被用作新对象的 __proto__。

```
// 构造函数示例
function Range(from, to) {
  this.from = from;
  this.to = to;
}

Range.prototype = {
  include: function(x) {
    return this.from <= x && x <= this.to
  }
}
```

值得注意的是：构造函数定义的是类，所以首字母要大写。

知乎 首发于 不想做程序员的产品经理不是好产品经理

我们来从一个日期对象开始，来看看它的原型链是怎么回事儿：

```
const d = new Date()
```

d 的原型链

看上图，我们可以知道，d 的原型链就是：

d => d.__proto__ => d.__proto__.__proto__ => d.__proto__.__proto__.__proto__

那么，一个对象的 __proto__ 是怎么得来的呢？

答案是**由它的原型对象的 prototype 决定的**：

- d 是由 Date 构造函数构建的，所以 d.__proto__ = Date.prototype
- Date.prototype 是由 Object 构造函数构建的，所以 d.__proto__.__proto__ = Date.prototype.__proto__ = Object.prototype
- Object.prototype 是原型链的尽头了，因为 Object.prototype.__proto__ = null

不要搞混

很多人以为原型链是这样子的：

```
d => d.__proto__ = Date.prototype => Date.__proto__ = Object.prototype => Object.__pro
```

这样子的话你会发现，原型链就没有尽头了，最后都是 Function。

所以其实原型链不是构造函数的 __proto__，而是构造函数 prototype 原型对象的 __proto__。

看图说话

这样子说可能有点绕，我们还是看图说话：

红色线的才是 d 的原型链，而不是其他什么的。

四、理解 ES6 的 class 语法其实是原型链的语法糖

其实我认为 class 解决的是「继承」难的问题：

在 class 出现之前，其实「对象」继承「对象」很容易——通过 Object.create() 函数就可以了。

但是「构造函数」继承「构造函数」就比较困难了——而 extends 关键字 + class 关键字解决了这一难题。

我们先来简单看下 class + extends 的简单示例：

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
```

```
}

class Dog extends Animal {
  speak() {
    console.log(this.name + ' barks.');
  }
}

const dog = new Dog("Mitzie");

dog.speak()

> 输出： Mitzie barks
```

extends 的本质

我们来看如下代码中，extends 都做了些什么

```
class A {
  constructor(a) {
    this.a = a;
  }
}
class B extends A {
  constructor(a, b) {
    super(a);
    this.b = b;
  }
}
```

第一步：B.prototype.__proto__ = A.prototype

第二步：B.__proto__ = A

第三步：B 继承了 A 构造函数中的属性和方法（具体实现暂且不表）

所以我们完成了所谓的「构造函数 B」对「构造函数 A」的继承。

——因为它不仅从 A 处获得「__proto__」，还从 A.prototype 处获得了「__prototype__」

而使用如下的 Object.create()：

```
const A = {};
A.prototype = {
  a: "Hello, A!"
}

const B = Object.create(A)
```

只能做到让「对象 B」 继承了「构造函数 A」的原型。

—— B 仅仅从 A 处获得「__proto__」，但没有从 A.prototype 处获得「__prototype__」，B.prototype === undefined!

编辑于 2020-03-08 12:54

知乎 首发于 不想做程序员的产品经理不是好产品经理

欢迎参与讨论

1 条评论 默认 最新

零延迟 ...

感觉讲的好好，本来看这部分挺蒙的，现在感觉懂了不少

2020-11-27 回复 喜欢

文章被以下专栏收录

不想做程序员的产品经理不是好产品经理

推荐阅读

快速读懂 JS 原型链

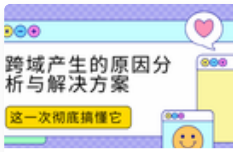
孟思行 发表于大前端技术

前端中 try-catch 捕获不到哪些异常和错误

小蚊子 发表于前端之家

[译]深入React fiber 链表和 DFS

Richa... 发表于手不释卷



跨域（CORS）产生原因分解决方案，这一次彻底搞

五月君 发表于N