

17 | Object.setPrototypeOf(x, null): 连Brendan Eich都认错, 但null值还活着

2019-12-23 周爱民

《JavaScript核心原理解析》

课程介绍 >



讲述：周爱民

时长 14:26 大小 13.23M



你好，我是周爱民。欢迎回来继续学习 JavaScript。

今天是关于面向对象的最后一讲，上次已经说过，今天这一讲要讨论的是原子对象。关于原子对象的讨论，我们应该从null值讲起。

null值是一个对象。

null 值

很多人说 JavaScript 中的null值是一个 BUG 设计，连 JavaScript 之父 Eich 都跳出来对 Undefined+Null 的双设计痛心疾首，说null值的特殊设计是一个“抽象漏洞（abstraction leak）”。这个东西什么意思呢？很难描述，基本上你可以理解为在概念设计层面（也就是抽象层）脑袋突然抽抽了，一不小心就写出这么个怪胎。



NOTE: [“typeof null”的历史](#)，[JavaScript 的设计失误](#)。

然而我却总是觉得不尽如此，因为如果你仔细思考过 JavaScript 的类型系统，你就会发现 null 值的出现是有一定的道理的（当然 Eich 当年脑子是不是这样犯的抽抽也未为可知）。怎么讲呢？

早期的 JavaScript 一共有 6 种类型，其中 number、string、boolean、object 和 function 都是有一个确切的“值”的，而第 6 种类型 Undefined 定义了它们的反面，也就是“非值”。一般讲 JavaScript 的书大抵上都会这么说：

undefined 用于表达一个值 / 数据不存在，也就是“非值（non-value）”，例如 return 没有返回值，或变量声明了但没有绑定数据。

这样一来，“值 + 非值”就构成了一个完整的类型系统。

但是呢，JavaScript 又是一种“面向对象”的语言。那么“对象”作为一个类型系统，在抽象上是不是也有“非对象”这样的概念呢？有啊，答案就是“null”，它的语义是：

null 用于表达一个对象不存在，也就是“非对象”，例如在原型继承中上溯原型链直到根类——根类没有父类，因此它的原型就指向 null。

正如“undefined”是一个值类型一样，“null”值也是一个对象类型。这很对称、很完美，只要你愿意承认“JavaScript 中存在两套类型系统”，那么上面的一切解释就都行得通。

事实上，不管你承不承认，这样的两套类型系统都是存在的。也因此，才有了所谓的**值类型的包装类**，以及对象的 `valueOf()` 这个原型方法。

现在，的确是时候承认 `typeof(null) === 'object'` 这个设计的合理性了。

Null 类型

正如 Undefined 是一个类型，而 undefined 是它唯一的值一样，Null 也是一个类型，且 null 是它唯一的值。



你或许已经发现，我在这里其实直接引用了 ECMAScript 对 Null 类型的描述？的确，ECMAScript 就是这样约定了null值的出处，并且很不幸的是，它还约定了null值是一个原始值（Primitive values），这是 ECMAScript 的概念与我在前面的叙述中唯一冲突的地方。

如果你“能 / 愿意”违逆 ECMAScript 对“语言类型（*Language types*）”的说明，稍稍“苟同”一下我上述的看法，那么下面的代码一定会让你觉得“豁然开朗”。这三行代码分别说明：

1. null 是对象；
2. 类可以派生自 null；
3. 对象也可以创建自 null。

 复制代码

```
1 // null是对象
2 > typeof(null)
3 'object'
4
5 // 类可以派生自null
6 > MyClass = class extends null {}
7 [Function: MyClass]
8
9 // 对象可以创建自null
10 > x = Object.create(null);
11 {}
```

所以，Null 类型是一个“对象类型（也就是类）”，是所有对象类型的“元类型”。

而null值，是一个连属性表没有的对象，它是“元类型”系统的第一个实例，你可以称之为一个原子。


属性表

没有属性表的对象称为 null。而一个原子级别的对象，意味着它只有一个属性表，它不继承自任何其他既有的对象，因此这个属性表的原型也就指向 null。

原子对象是“对象”的最原始的形态。它的唯一特点就是“原型为 null”，其中有一些典型示例，譬如：



1. 你可以使用 `Object.getPrototypeOf()` 来发现，`Object()` 这个构造器的原型其实也是一个原子对象。——也就是所有一般对象的祖先类最终指向的，仍然是一个 `null` 值。
2. 你也可以使用 `Object.setPrototypeOf()` 来将任何对象的原型指向 `null` 值，从而让这个对象“变成”一个原子对象。


 复制代码

```
1 # JavaScript中“Object（对象类型）”的原型是一个原子对象
2 > Object.getPrototypeOf(Object.prototype)
3 null
4
5 # 任何对象都可以通过将原型置为null来“变成”原子对象
6 > Object.setPrototypeOf(new Object, null)
7 {}
```

但为什么要“变成”原子对象呢？或者说，你为什么需要一个“原子对象”呢？

因为它就是“对象”最真实的、最原始的、最基础抽象的那个数据结构：**关联数组**。

所谓属性表，就是关联数组。一个空索引数组与空的关联数组在 JavaScript 中是类似的（都是对象）：

 复制代码

```
1 # 空索引数组
2 > a = Object.setPrototypeOf(new Array, null)
3 {}
4
5 # 空关联数组
6 > x = Object.setPrototypeOf(new Object, null)
7 {}
```

而且本质上来说，空的索引数组只是在它的属性表中默认有一个不可列举的属性，也就是 `length`。例如：

 复制代码

```
1 # （续上例）
2
3 # 数组的长度
4 > a.length
5 0
```



```
6 # 索引数组的属性
7 > Object.getOwnPropertyDescriptors(a)
8 { length:
9   { value: 0,
10     writable: true,
11     enumerable: false,
12     configurable: false } }
13
```

正因为数组有一个默认的、隐含的“length”属性，所以它才能被迭代器列举（以及适用于数组展开语法），因为迭代器需要“额外地维护一个值的索引”，这种情况下“length”属性成了有效的参考，以便于在迭代器中将“0...length-1”作为迭代的中止条件。

而一个原子的、支持迭代的索引数组也可通过添加“Symbol.iterator”属性来得到。例如：

 复制代码

```
1 # （续上例）
2
3 # 使索引数组支持迭代
4 > a[Symbol.iterator] = Array.prototype[Symbol.iterator]
5 [Function: values]
6
7 # 展开语法（以及其他运算）
8 > [...a]
9 []
```

现在，整个 JavaScript 的对象系统被还原到了两张简单的属性表，它们是两个原子对象，一个用于表达索引数组，另一个用于表达关联数组。


当然，还有一个对象，也是所有原子对象的父类实例：null。

派生自原子的类

JavaScript 中的类，本质上是原型继承的一个封装。而原型继承，则可以理解为多层次的关联数组的链（原型链就是属性表的链）。之所以在这里说它是“多层次的”，是因为在面向对象技术出现的早期，在《结构程序设计》这本由三位图灵奖得主合写的经典著作中，“面向对象编程”就被称为“层次结构程序设计”。所以，“层次设计”其实是从数据结构的视角对面向对象中继承特性的一个精准概括。



类声明将“extends”指向 null 值，并表明该类派生自 null。为了使这样的类（例如 MyClass）能创建出具有原子特性的实例，JavaScript 给它赋予了一个特性：MyClass.prototype 的原型指向 null。这个性质也与 JavaScript 中的 Object() 构造器类似。例如：

 复制代码

```
1 > class MyClass extends null {}
2 > Object.getPrototypeOf(MyClass.prototype)
3 null
4
5 > Object.getPrototypeOf(Object.prototype)
6 null
```

也就是说，这里的 MyClass() 类可以作为与 Object() 类处于类似层次的“根类”。通常而言，称为“（所有对象的）祖先类”。这种类，是在 JavaScript 中构建元类继承体系的基础。不过元类以及相关的话题，这里就不再展开讲述了。

这里希望你能关注的点，仅仅是在“层次结构”中，这样声明出来的类，与 Object() 处在相同的层级。

通过“extends null”来声明的类，是不能直接创建实例的，因为它的父类是 null，所以在默认构造器中的“SuperCall（也就是 super()）”将无法找到可用的父类来创建实例。因此，通常情况下使用“extends null”来声明的类，都由用户来声明一个自己的构造方法。

但是也有例外，你思考一下这个问题：如果 MyClass.prototype 指向 null，而 super 指向一个有效的父类，其结果如何呢？

是的，这样就得到了一个能创建“具有父类特性（例如父类的私有槽）”的原子对象。例如：

 复制代码

```
1 > class MyClass extends null {}
2
3 # 这是一个原子的函数类
4 > Object.setPrototypeOf(MyClass, Function);
5
6 # f()是一个函数，并且是原子的
7 > f = new MyClass;
8 > f(); // 可以调用
9 > typeof f; // 是"function"类型
10
```



```
11 # 这是一个原子的日期类
12 > Object.setPrototypeOf(MyClass, Date);
13
14 # d是一个日期对象，并且也是原子的
15 > d = new MyClass;
16 > Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
17 'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'
18
19 # a是一个原子的数组类
20 > Object.setPrototypeOf(MyClass, Array);
21 > a = new MyClass;
22
```

一般函数 / 构造器

由于一般函数可以直接作为构造器，你可能也已经习惯了这种从 ECMAScript 6 之前的 JavaScript 沿袭下来的风格。一般情况下，这样的构造器也可以被称为“（传统的）类”，并且在 ECMAScript 6 中，所谓“非派生类（没有 extends 声明的类）”实际上也是用这样的函数 / 构造器来实现的。

这样的函数 / 构造器 / 非派生类其实是相同性质的东西，并且都是基于 ECMAScript 6 之前的构造器概念来实现类的实例化——也就是构造过程的。出于这样的原因，它们都不能调用 SuperCall（也就是super()）来创建this实例。不过，旧式风格的构造过程将总是使用构造器的.prototype属性来创建实例。因而，让它们创建原子对象的方法也就变得非常简单：把它们的原型变成原子，就可以了。例如：

 复制代码

```
1 # 非派生类（没有extends声明的类）
2 > class MyClass {}
3 > Object.setPrototypeOf(MyClass.prototype, null)
4 > new MyClass
5 {}
6
7 # 一般函数/构造器
8 > function AClass() {}
9 > Object.setPrototypeOf(AClass.prototype, null)
10 > new MyClass
11 {}
```

原子行为



直接施加于原子对象上的最终行为，可以称为原子行为。如同 LISP 中的表只有 7 个基本操作符一样，原子行为的数量也是很少的。准确地说，对于 JavaScript 来说，它只有 13 个，可以分成三类，其中包括：

- 操作原型的，3 个，分别用于读写内部原型槽，以及基于原型链检索；
- 操作属性表的，8 个，包括冻结、检索、置值和查找等（类似于数据库的增删查改）；
- 操作函数行为的，2 个，分别用于函数调用和对象构造。

讲到这里，你可能已经意识到了，所谓“代理对象（Proxy）”的陷阱方法，也正好就是这 13 个。这同样也可以理解为：代理对象就是接管一个对象的原子行为，将它转发给被代理行为处理。

正因为 JavaScript 的对象有且仅有这 13 个原子行为，所以代理才能“无缝且全面地”代理任何对象。

这也是在 ECMAScript 中的代理变体对象（proxy object is an exotic object）只有 15 个内部槽的原因：包括上述 13 个原子行为的内部槽，其他两个内部槽分别指向被代理对象（ProxyTarget）和用户代码设置的陷阱列表（ProxyHandler）。总共 15 个，不多不少。

NOTE: 如果更详细地考察 13 个代理方法，其实严格地说来只有 8 个原子行为，其实其他 5 个行为是有相互依赖的，而非原子级别的操作。这 5 个“非原子行为”的代理方法是 DefineOwnProperty、HasProperty、Get、Set 和 Delete，它们会调用其他原子行为来检查原型或属性描述符。

知识回顾

任何一个对象都可以通过标题中的语法变成原子对象，它可以被理解为**关联数组**；并且，如果它有一个称为“length”的属性，那么它就可以被理解为**索引数组**。我们在上一讲中说过，所有的数据，在本质上来说都可以看成“连续的一堆”，或“不连续的一堆”，所以“索引数组 + 关联数组”在数据结构上就可以表达“所有的数据”。


如果你对有关 JavaScript 的类型系统，尤其是隐于其中的**原子类型**和**元类型**等相关知识感兴趣，可以阅读我的另外一篇博客文章 [《元类型系统是对 JavaScript 内建概念的补充》](#)。




好了，今天的课程就到这里。很高兴你能一路坚持着将之前的十七讲听完，不过对于 JavaScript 语言最独特的那些设计，我们其实才初窥门径。现在，尽管你已经在原子层面掌握了“数据”，但从计算机语言的角度上来看，你只是拥有了一个静态的系统，最重要的、也是现在最缺乏的，是让它们“动起来”。

从下一讲开始，我会与你聊聊“动态语言”，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

分享给需要的人，Ta购买本课程，你将得 20 元

 生成海报并分享

 赞 5  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 16 | $[a, b] = \{a, b\}$: 让你从一行代码看到对象的本质

下一篇 18 | $a + b$: 动态类型是灾难之源还是最好的特性? (上)

学习推荐

JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费 





行问

2019-12-23

多看看技术在历史上是怎么出现的，怎么解决问题的，溯源这种“原型链”让我大呼过瘾。一路学习下来，有完全不懂，有闻所未闻，有懵逼，有茅塞顿开等。今天的这一讲，让我理解了"null" 在实际开发中的合理运用。

作者回复: ^^

多谢多谢。能讲得对大家有用就好。:)



👍 9



Objectivezt

2019-12-26

这是除了加餐课外，我能最快理解的一节课，嗯一定是我进步了😁

作者回复: 一定是进步了+1 😊



👍 3



小炭

2020-11-10

“原子对象”这一概念只有Javascript才会有吗，在C和C++的标准术语也有这个“原子对象”的定义。不知道他们之间有什么区别，或者这个定义的源头来自哪里？

作者回复: 从ecmascript来说，也没有原子对象这个概念。我之所以提到这个，是因为“原子性”这个概念可以用在这里，表明这种对象是“原子性的”。如果你有兴趣读一下《JavaScript语言精髓与编程实践（第三版）》，会对这个概念，以及由此带来的一个类型体系有更深入的了解。

我最早看到这个概念是在李战的《悟透Delphi》这本书中，我记得后来这本书并没有出版。大概是在那个时间点前后，程序员圈子里兴起过一阵关于“语言原子性”的讨论。

又，这已经是快20年前的事情了。



👍 2



蛋黄酱

2020-03-15

> 如果 MyClass.prototype 指向 null，而 super 指向一个有效的父类，其结果如何呢

这配上示例代码，意思是说setPrototypeOf虽然字面上的意思是改变prototype但本质上只改变了super执行的对象？我觉得不对吧？

作者回复: 一共影响三个东西，一个是MyClass和MyClass.prototype中所有方法的super指向，二个是使MyClass的创建过程与super（例如这里的Date）动态绑定起来，三个是MyClass自己的类方法（静态方法）。只不过第三个没有表现在示例中。

这里用setPrototypeOf()改的是MyClass的原型，而不是MyClass.prototype的原型。

共 2 条评论 >

👍 1



卡尔

2021-01-12

老师，我记得有一本书里说，undefined派生于null。老师这句话怎么去理解，他俩到底是什么关系？有什么区别？

作者回复: 那本书一定写错了。

这两者没有直接的关系。在ECMA的概念上，二者都是原始值（primitive values）；在JavaScript的概念上，null是对象，而undefined是值（类型）的数据。无论是哪一种理解，二者都没有派生或类属的关系。

关于二者的更多区别，还是建议看一下绿皮书，专门有一节来讨论这个问题。



新哥

2020-06-21

是时候讲一下 prototype和__proto__了😄



HoSalt

2020-05-25

```
class A {}
```

```
class B extends A {}
```

```
B.__proto__ === A // true
```

```
B.__proto__.__proto__ === Function.prototype // true
```



```
class MyClass extends null {}
```

```
MyClass.__proto__ === Function.prototype // true
```

老师继承自null的类的原型链直接指向了Function.prototype，而其它的是在中间加了一层，这是一种特殊处理？

作者回复: 这是因为MyClass本来就是一个函数，它的原型（缺省）指向Function.prototype是正常的。

X.prototype不应当是一个null值——对于JS来说，置null值是“无效值”。当这个值无效时（例如null/undefined），等义于它使用Object.prototype。你试着找一个其它对象来试一下就知道了。

X.prototype存在无效值的原因是：缺省情况下，这个属性是可写的。因为“对象属性”在历史中可以写成任何值，所以历史上它就没有“属性类型”这样的限制。这是一个继承历史而来的设计。



t86

2020-01-16

老师的功力真的是深，佩服



水木年华

2020-01-05

老师讲的真好，有体会有收获😊



许童童

2019-12-24

老师讲得太好了。

