说明：文中的粗体 **“...对象”** 指普通对象， **“...函数”** 指函数对象。

> 一生二，二生三，三生万物。

null

起初，在 javaScript 的世界中只有 null ，他无色无味，无量无体，人类穷尽所有的方法也无法感知他的存在，他是真正的超越了康德的哲学作为 **空** 存在于世。

对象

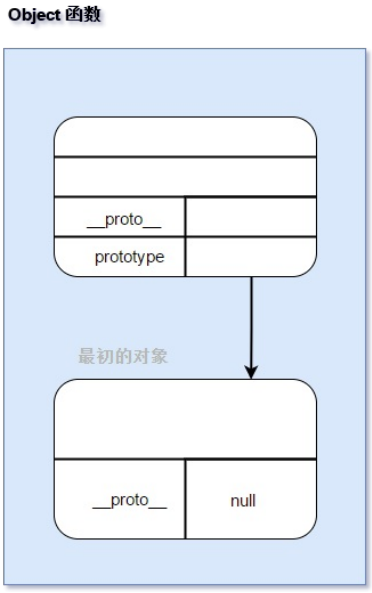
也许像宇宙演变一样，在奇点时刻一个 **对象** 从 null 中脱胎而出。他的结构简单让人过目不忘，而其中最令人记忆深刻的是他在自己的身体内烙印了父辈的名字：

```
{
  toString: 'toString',
  valueOf: 'valueOf',
  __proto__: null
}
```

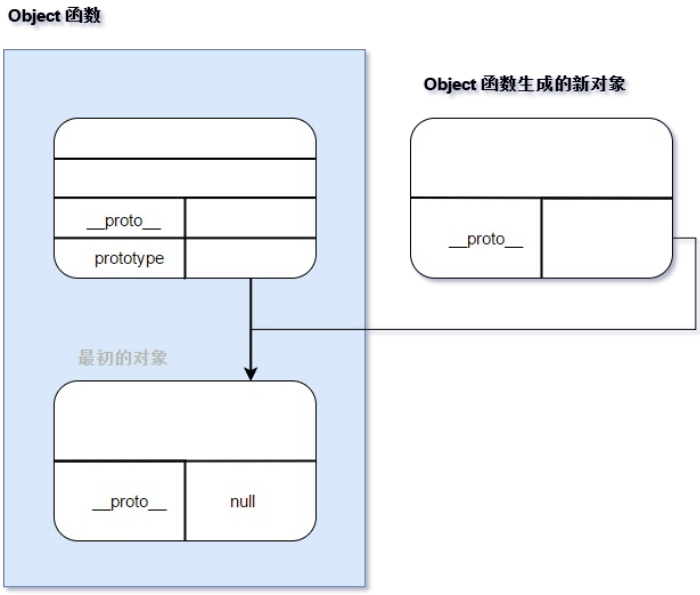
如果说 null 是空是死寂，那这些键值对就是 JavaScript 世界中的精灵，让整个世界充满生机。非常凑巧的是 JS 世界地缘辽阔、气候适宜特别适合 **对象** 的繁衍，因此 **对象** 的数量随着时间的推移而不断增加，与此同时 **对象** 的结构也变得复杂起来：

```
{
  key0: 'value0',
  __proto__: {
    toString: 'toString',
    valueOf: 'valueOf',
    __proto__: null
  }
}
```

不难发现这些 **对象** 的父辈都是最初的对象，人类的好奇心促使我们追源溯本，去揭示 **对象** 的繁衍过程，而了解这个过程并不曲折因为我们很快发现了 **Object 函数** 。原来 **最初的对象** 为了生存和繁衍将自己整体包裹在 **Object 函数** 内，这种行为就像是基因为了繁衍将自己包裹在人这个外壳内。这里我们也可以像绘制基因图一样给出 **Object 函数** 的结构图，但为了简便只给出示意图：



Object 函数 是由键值对组成，所以她也是 **对象**，一个会 **生成对象的对象** 。她的 prototype 属性指向 **最初的对象**，利用这一点她会在生成新对象过程中将 **最初的对象** 的名字放入新对象的 __proto__ 属性中：



我们获取这个关键步骤的主要证据是一个等式：

```
const o = new Object()
// const o = {}
o.__proto__ === Object.prototype // true
```

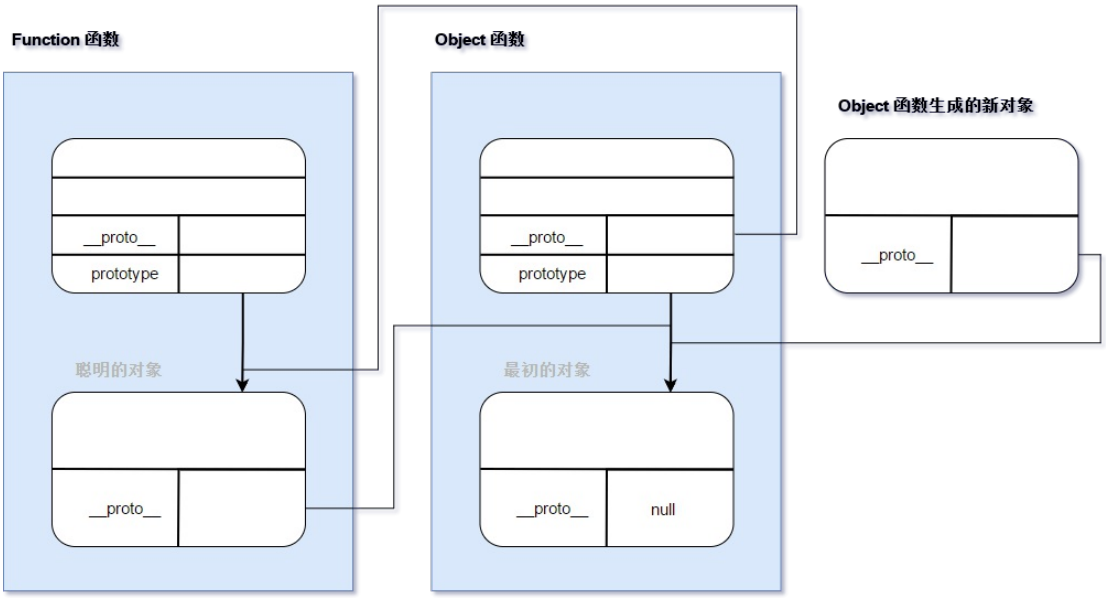
根据这个关键步骤推理得知，由某个函数生成的对象，她的父辈就是这个函数所包裹的对象（函数原型）。

Function 函数

Object 函数 也是 **对象**，她不像 **最初的对象** 是在无法理解的奇点时刻产生，那么她又是谁的延续？Easy，看看她的 __proto__ 属性指向谁，因为我们已经知道了 **对象** 的规律，即 __proto__ 属性指向她的父辈：

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

Function 函数 的 prototype 属性所指的 **对象** 是 **Object 函数** 的父辈，我们就称他 **聪明的对象** 。看来 **最初的对象** 并不聪明，将自己包裹在函数内的想法，是 **聪明的对象** 最先发明的。他将自己包裹在一个被称为 **Function** 的函数内。**Function 函数** 的结构与 **Object 函数** 类似，对比一下他们的示意图：

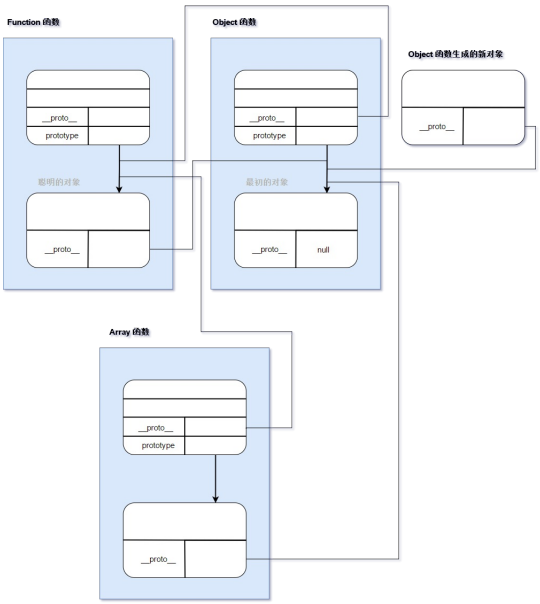


真相大白，**聪明的对象** 在 **最初的对象** 基础上将自己进行了改造，并将自己包裹在 **Function 函数** 内，而 **Function 函数** 是生产函数的函数，它又生产出了 **Object 函数** ，最后 **最初的对象** 将自己包裹在 **Object 函数** 内。

```
// 聪明对象的父辈是最初的对象
Function.prototype.__proto__ === Object.prototype // true
// Object 函数的父辈是聪明的对象
Object.__proto__ === Function.prototype // true
```

其他

通过不懈的努力，我们发现与 **聪明的对象** 一样对 **最初的对象** 进行了改造的还有 Array.prototype ,String.prototype , Number.prototype 等**内置对象** ， 他们也将自己包裹在相应的 **内置函数** 内。 如 Array.prototype 把自己包裹在 **Array 函数** 内：



最后我们还需要思考的是 **聪明的对象** 是如何包裹自己的，也就是 **Function 函数** 的产生。这里需要的小技巧估计大家都已经猜到了，是的看看她的 __proto__ 属性：

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

WTF unction。 **Function 函数** 的父辈就是 **聪明的对象** ，函数就是对象 。

总结

- 对象有两种，一种是普通对象，一种是函数对象, 函数对象继承自普通对象（看上面代码，就是最靠近的那个）。
- 函数对象由 **Function 函数** 生成，因此她们的 __proto__ 属性都指向 Function.prototype ，包括 **Function 函数** 自己，还包括自定义的函数（因为自定义函数都是 **Function 函数** 生成的）。

```
const fn = new Function()
// const fn = {}
fn.__proto__ === Function.prototype // true
```

- 除了 **Function 函数** 外，函数对象生成的是普通对象（没有 prototype 属性）。