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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中"Object（对象类型）"的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来"变成"原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）

# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);

# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中“Object（对象类型）”的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来“变成”原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）
```

```
# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);
```

```
# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中"Object（对象类型）"的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来"变成"原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）

# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);

# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中“Object（对象类型）”的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来“变成”原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）
```

```
# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);
```

```
# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中"Object（对象类型）"的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来"变成"原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）

# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);

# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中“Object（对象类型）”的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来“变成”原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）
```

```
# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);
```

```
# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中"Object（对象类型）"的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来"变成"原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）

# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);

# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中“Object（对象类型）”的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来“变成”原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）
```

```
# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);
```

```
# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中"Object（对象类型）"的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来"变成"原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）

# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);

# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中“Object（对象类型）”的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来“变成”原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）
```

```
# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);
```

```
# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中"Object（对象类型）"的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来"变成"原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）

# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);

# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中“Object（对象类型）”的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来“变成”原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）
```

```
# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);
```

```
# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中"Object（对象类型）"的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来"变成"原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）

# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);

# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中“Object（对象类型）”的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来“变成”原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）
```

```
# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);
```

```
# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中"Object（对象类型）"的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来"变成"原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）

# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);

# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。

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。

```
// null是对象
> typeof(null)
'object'

// 类可以派生自null
> MyClass = class extends null {}
[Function: MyClass]

// 对象可以创建自null
> x = Object.create(null);
{}
```

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。

```
# JavaScript中“Object（对象类型）”的原型是一个原子对象
> Object.getPrototypeOf(Object.prototype)
null

# 任何对象都可以通过将原型置为null来“变成”原子对象
> Object.setPrototypeOf(new Object, null)
{}
```

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）：

```
# 空索引数组
> a = Object.setPrototypeOf(new Array, null)
{}

# 空关联数组
> x = Object.setPrototypeOf(new Object, null)
{}
```

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：

```
# （续上例）
```

```
# 数组的长度
> a.length
0

# 索引数组的属性
> Object.getOwnPropertyDescriptors(a)
{ length:
   { value: 0,
     writable: true,
     enumerable: false,
     configurable: false } }
```

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：

```
# （续上例）

# 使索引数组支持迭代
> a[Symbol.iterator] = Array.prototype[Symbol.iterator]
[Function: values]

# 展开语法（以及其他运算）
> [...a]
[]
```

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：

```
> class MyClass extends null {}
> Object.getPrototypeOf(MyClass.prototype)
null

> Object.getPrototypeOf(Object.prototype)
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：

```
> class MyClass extends null {}

# 这是一个原子的函数类
> Object.setPrototypeOf(MyClass, Function);

# f()是一个函数，并且是原子的
> f = new MyClass;
> f(); // 可以调用
> typeof f; // 是"function"类型

# 这是一个原子的日期类
> Object.setPrototypeOf(MyClass, Date);
```

```
# d是一个日期对象，并且也是原子的
> d = new MyClass;
> Date.prototype.toString.call(d); // 它有内部槽用于存放日期值
'Mon Nov 04 2019 18:27:27 GMT+0800 (CST)'

# a是一个原子的数组类
> Object.setPrototypeOf(MyClass, Array);
> a = new MyClass;
...
```

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：

```
# 非派生类（没有extends声明的类）
> class MyClass {}
> Object.setPrototypeOf(MyClass.prototype, null)
> new MyClass
{}

# 一般函数/构造器
> function AClass() {}
> Object.setPrototypeOf(AClass.prototype, null)
> new MyClass
{}
```

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。