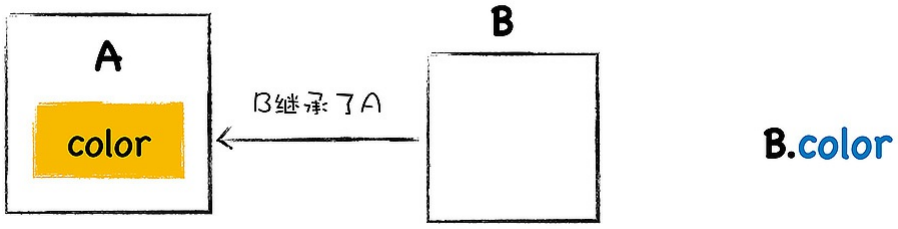
你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

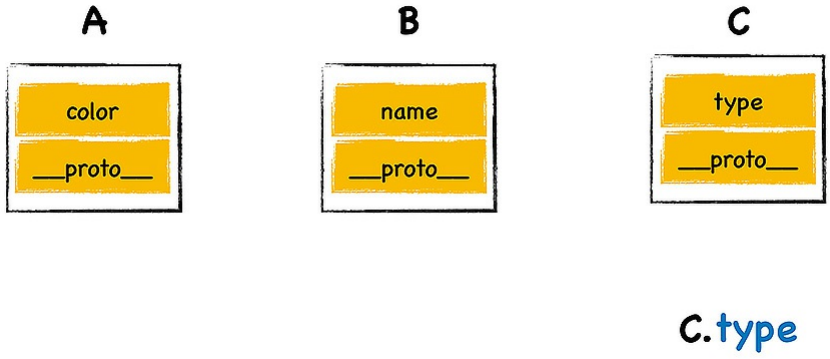
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

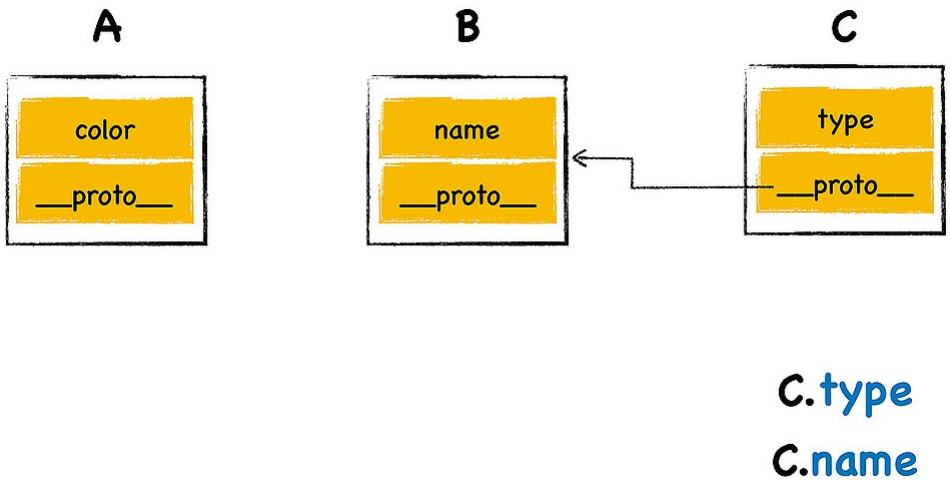


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

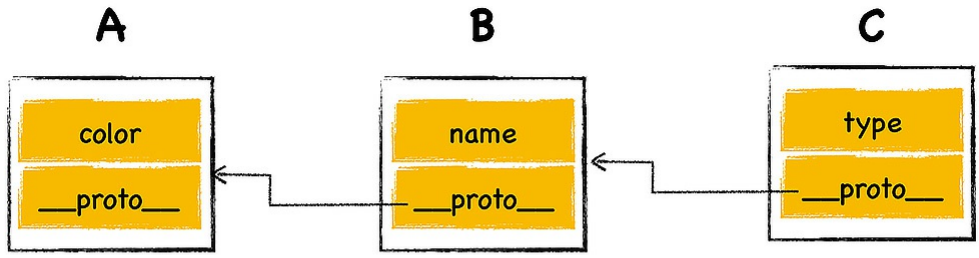
上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：



观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



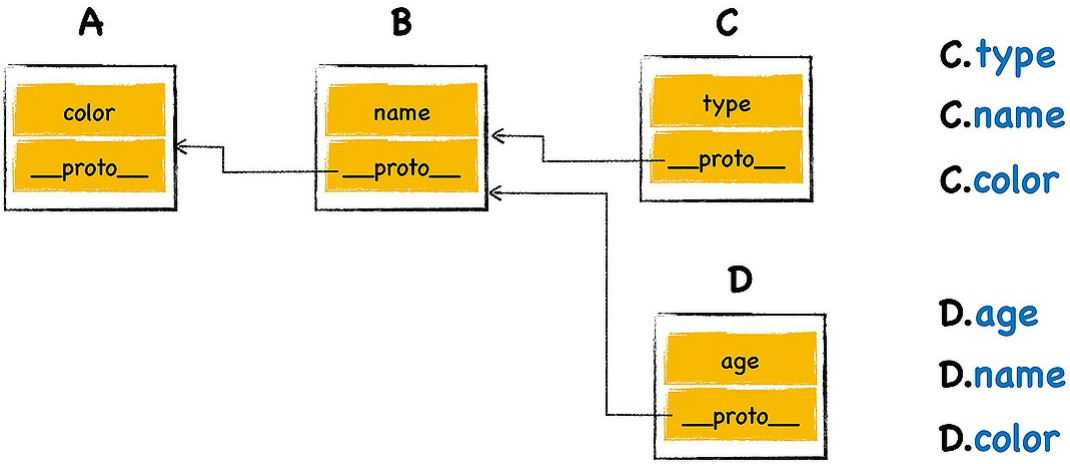
C.type
C.name
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type
C.name
C.color

D.age
D.name
D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过 *_proto_*

来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

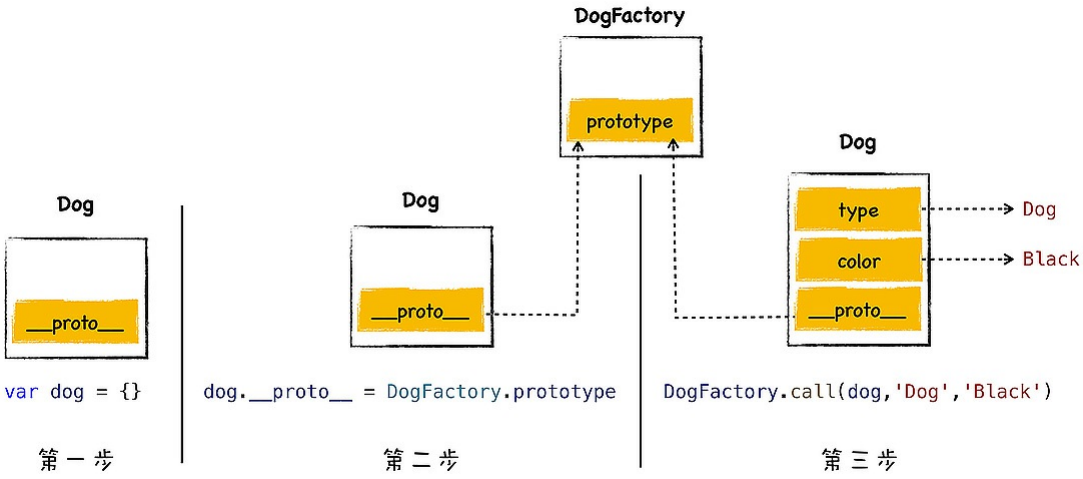
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

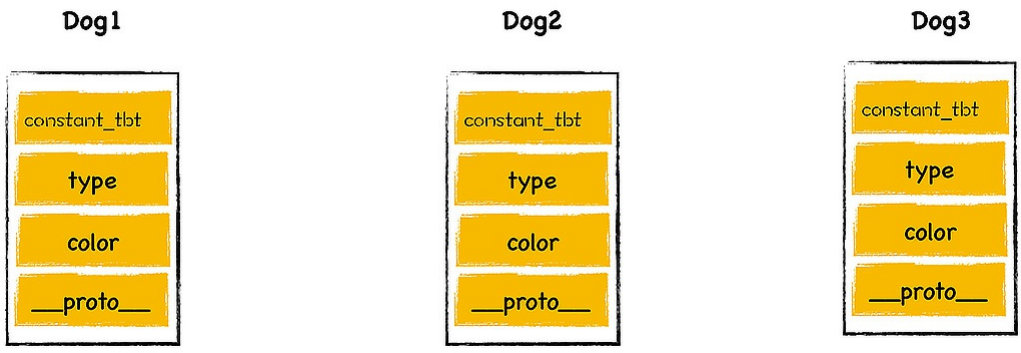
- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

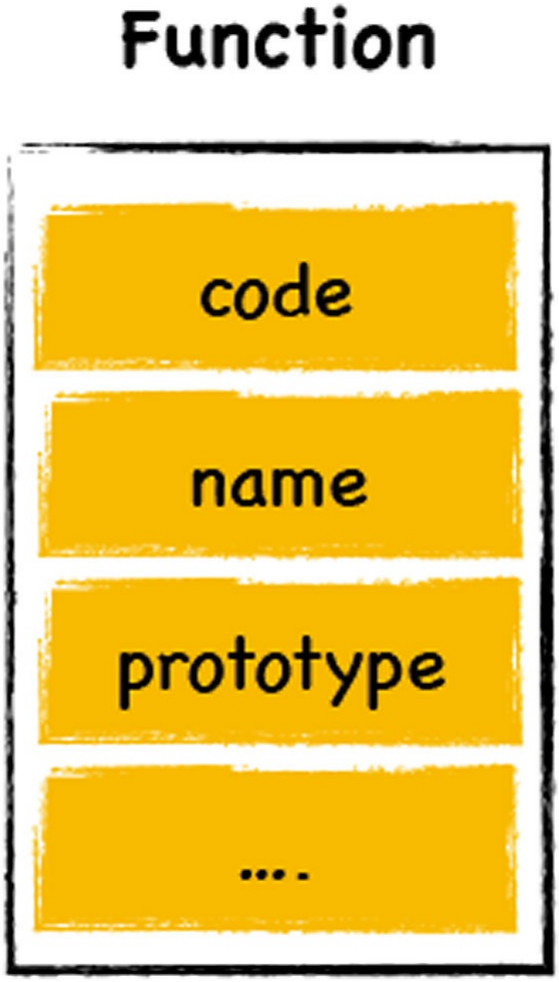
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

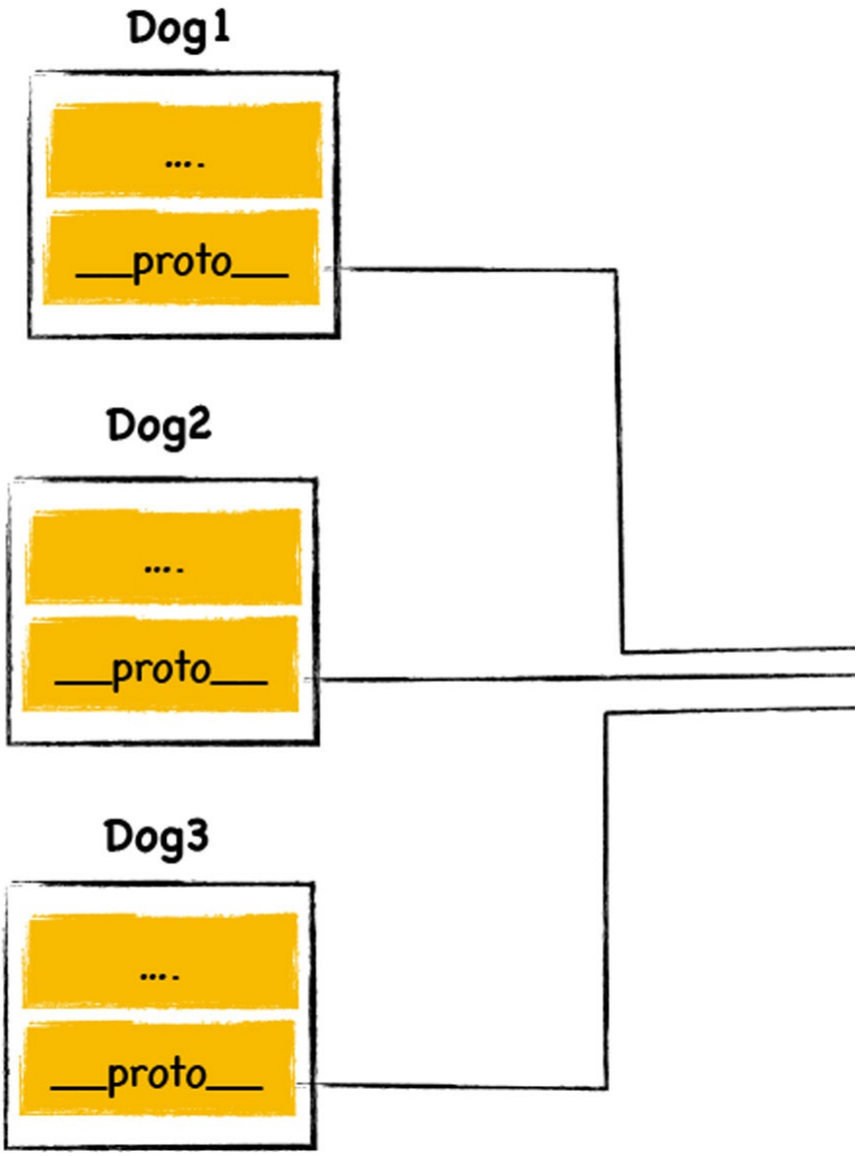
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

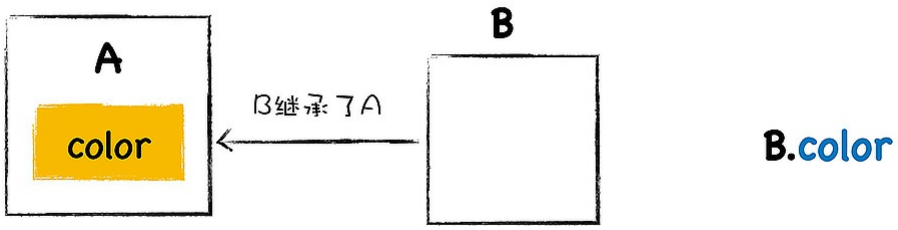
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

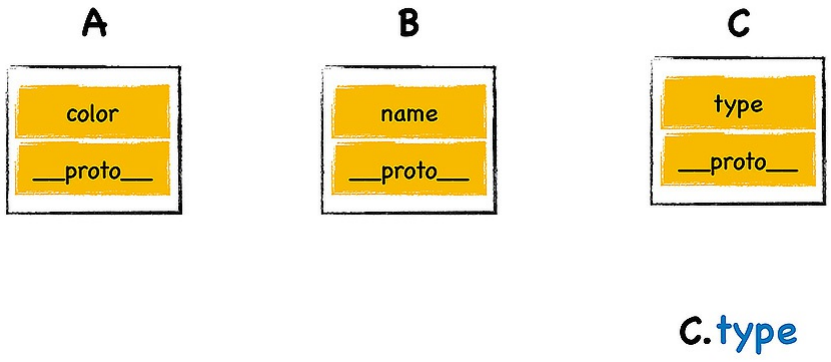
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

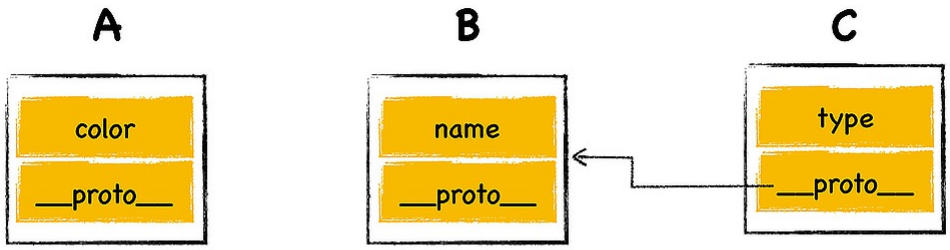


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

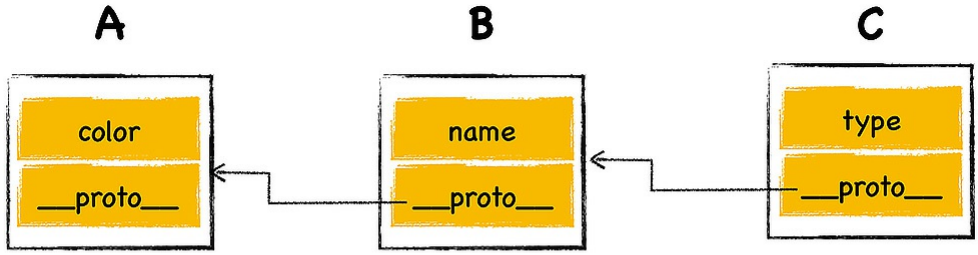


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

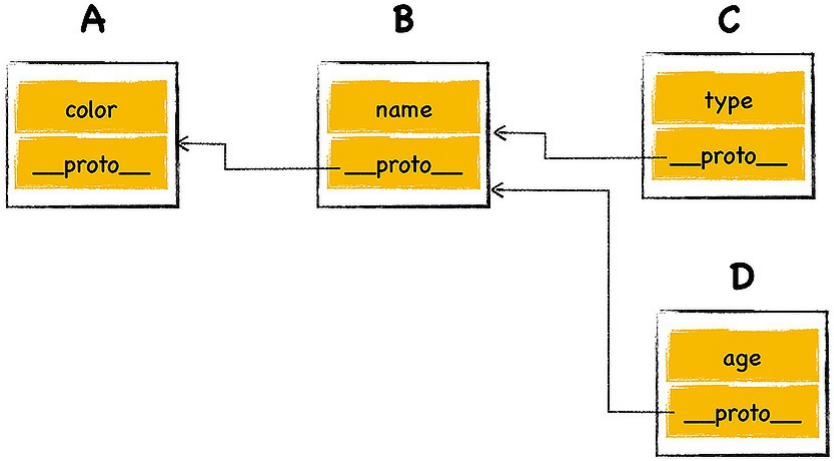
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过 *_proto_* 来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

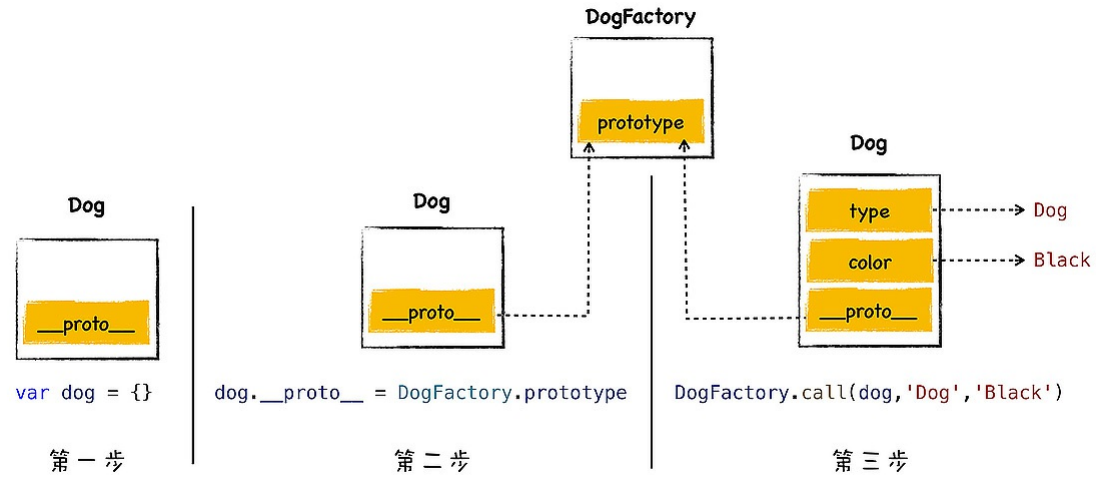
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

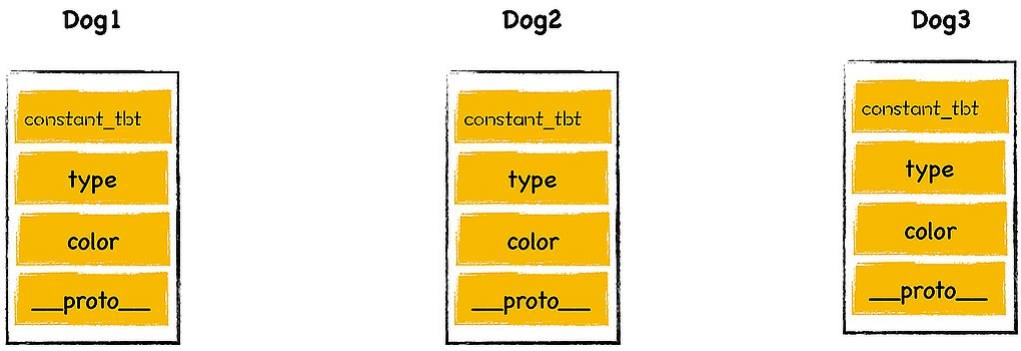
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

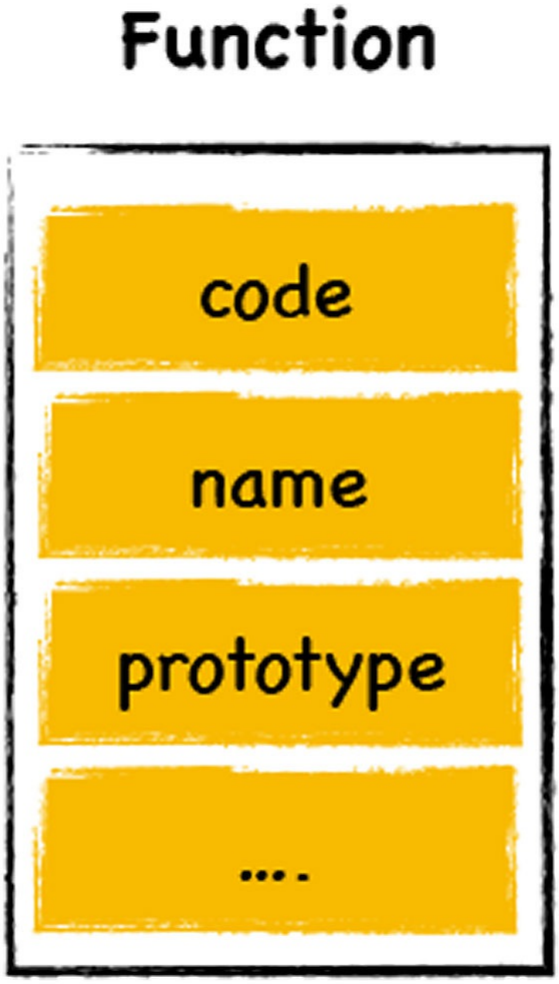
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

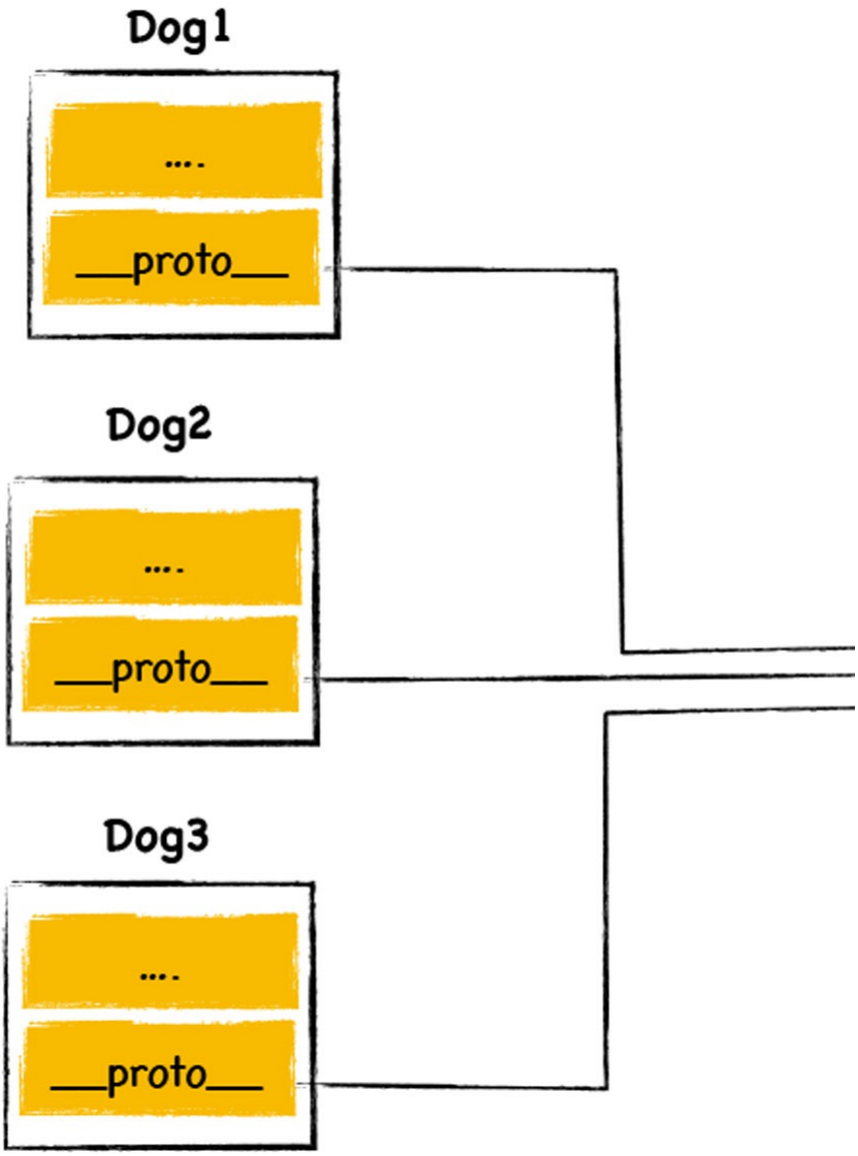
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

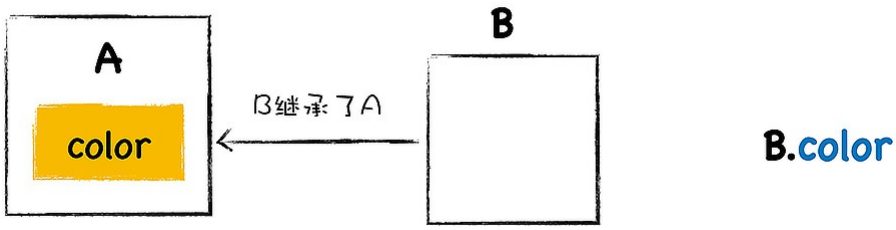
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

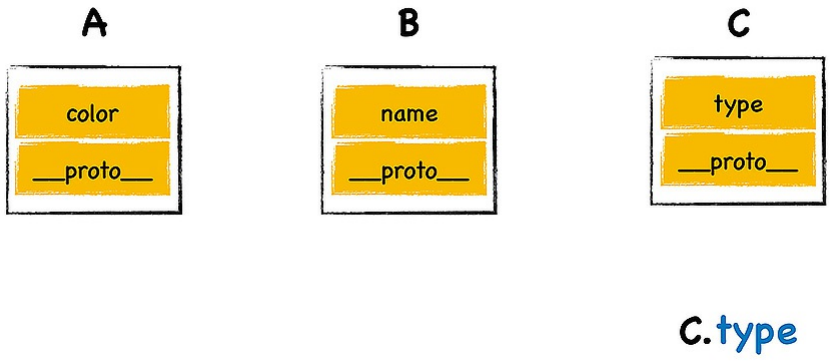
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

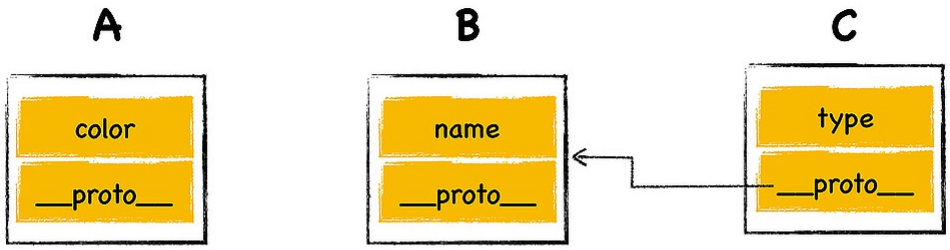


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

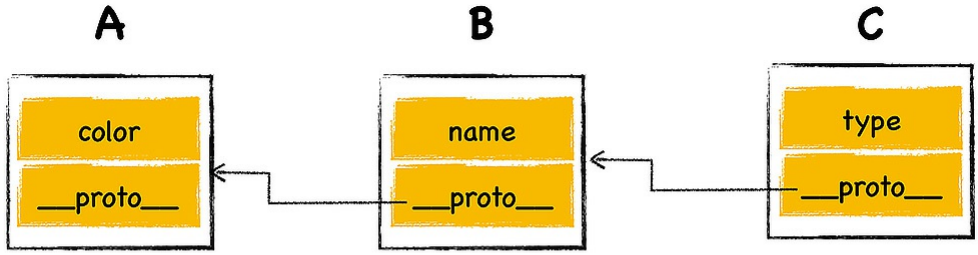


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

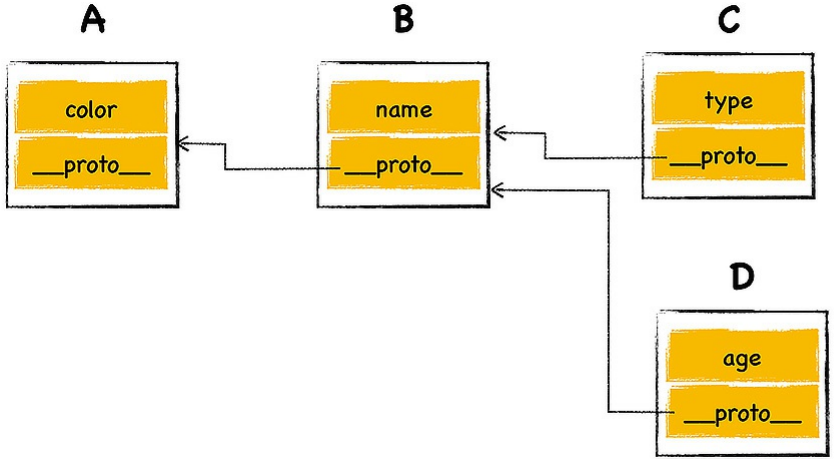
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过*_proto_*来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

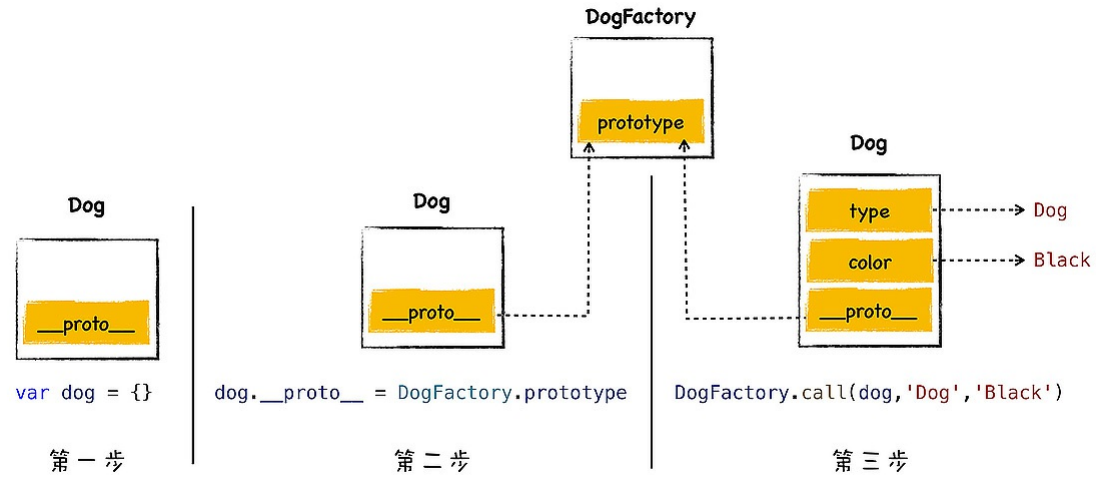
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

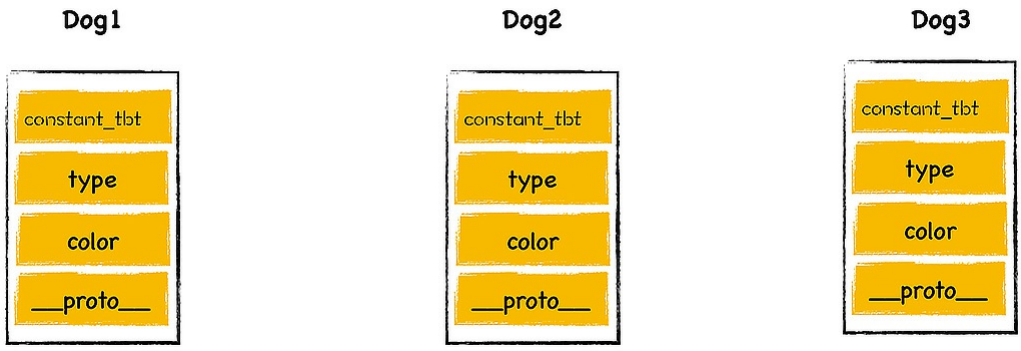
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

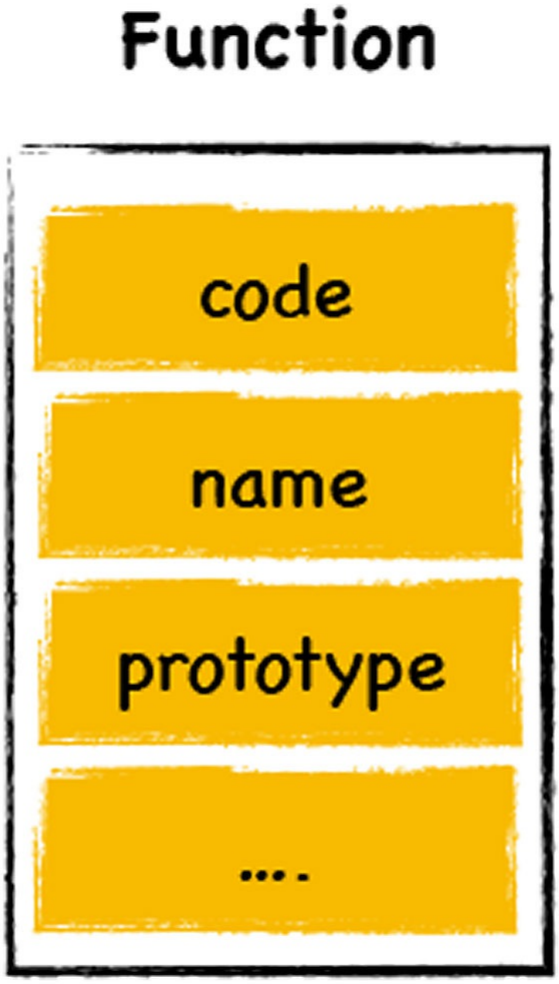
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

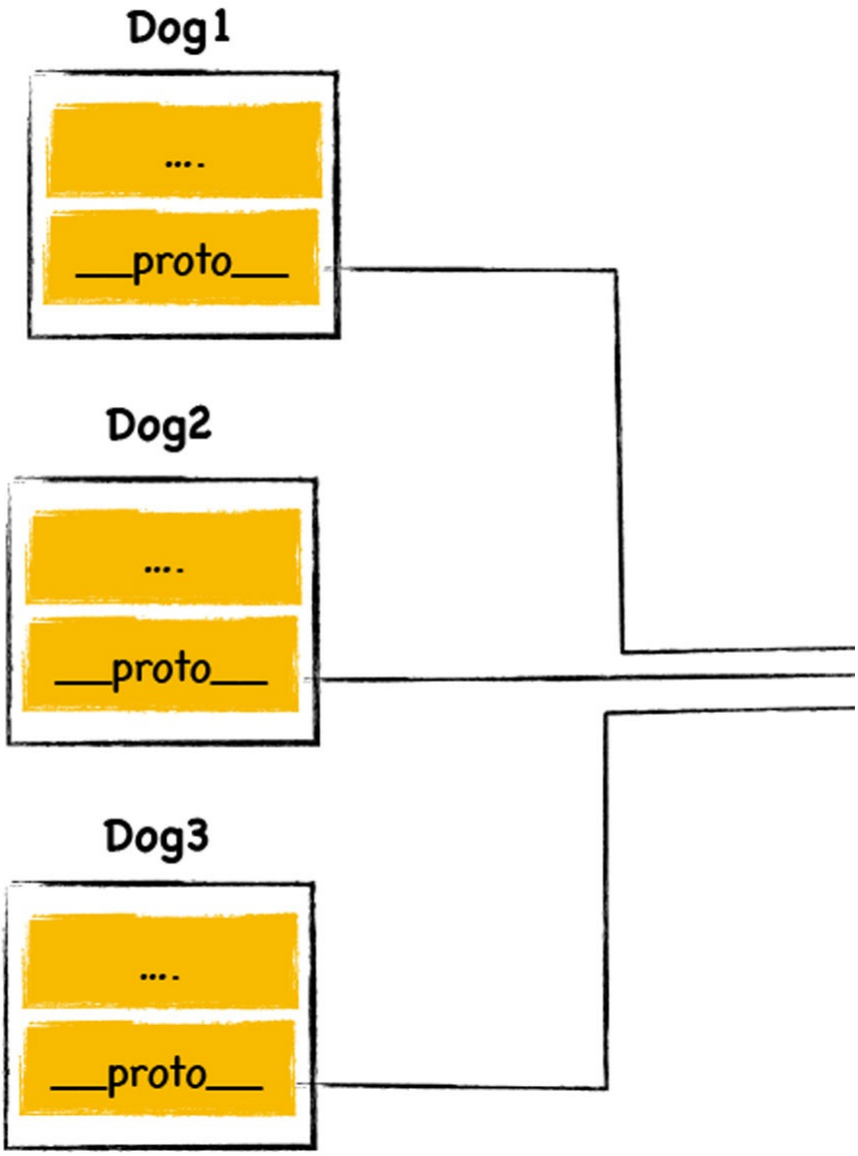
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

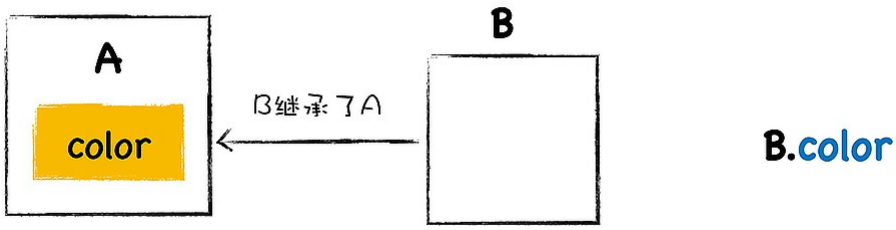
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

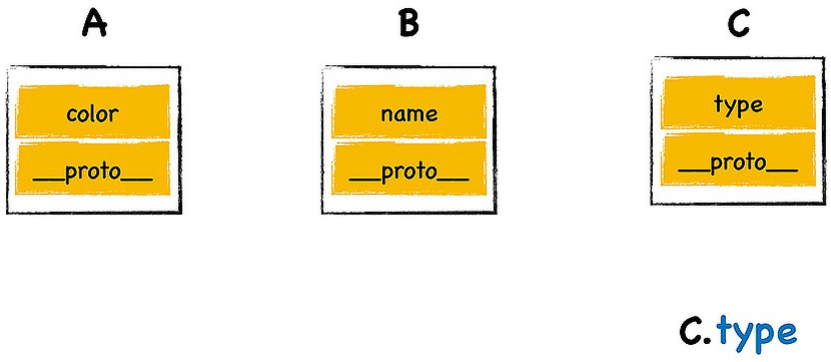
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

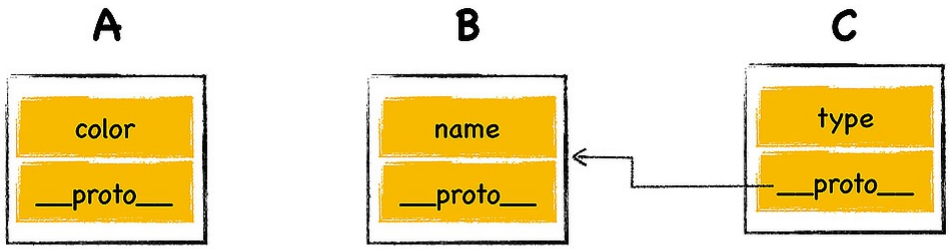


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

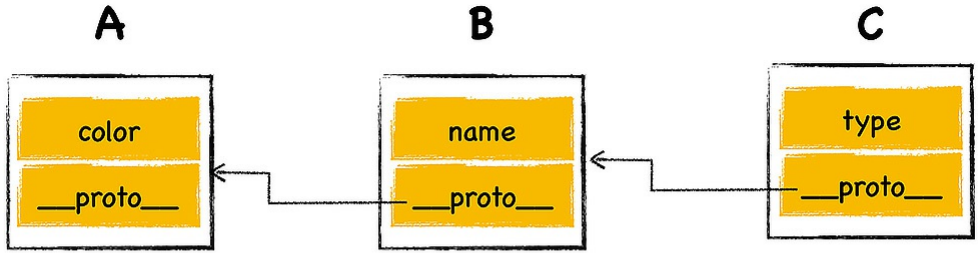


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

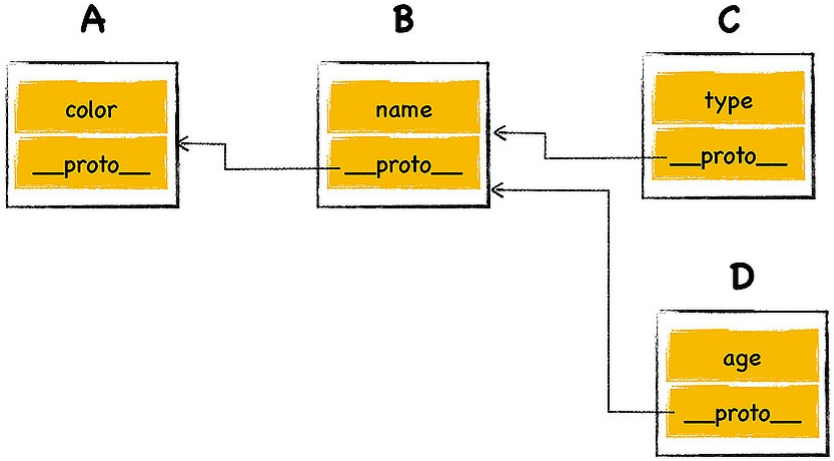
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过*_proto_*来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

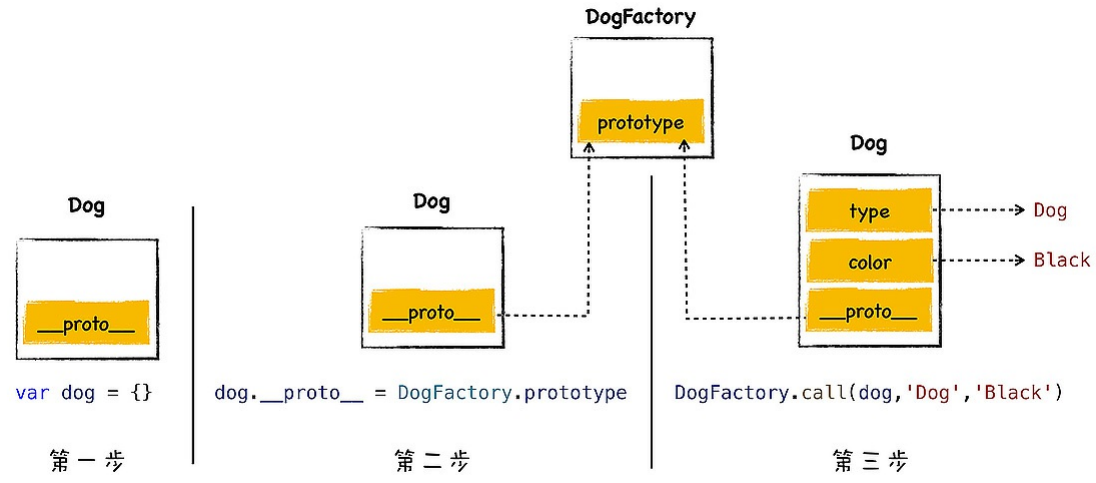
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

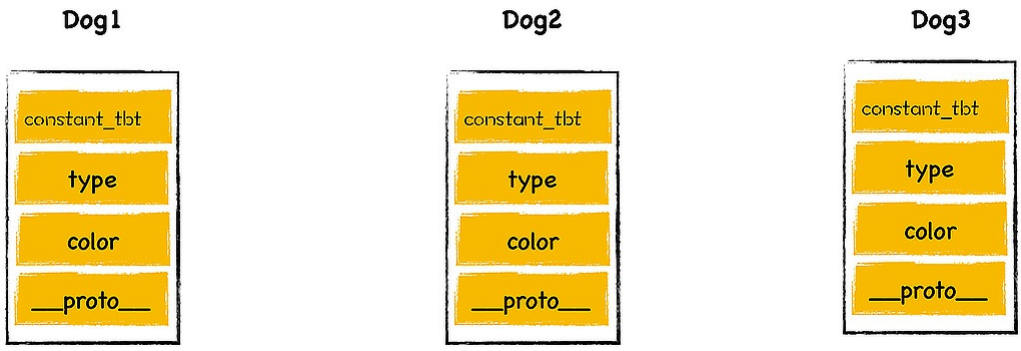
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

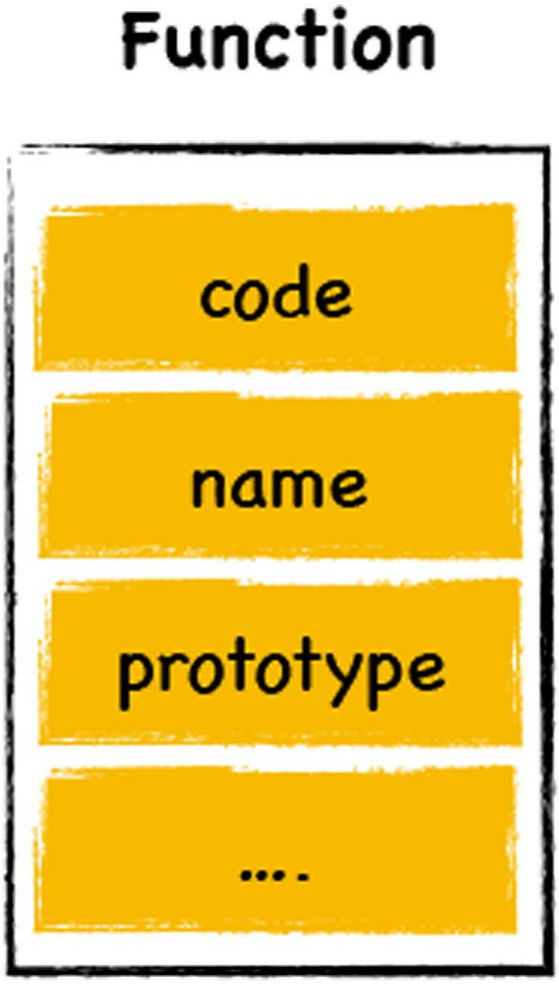
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

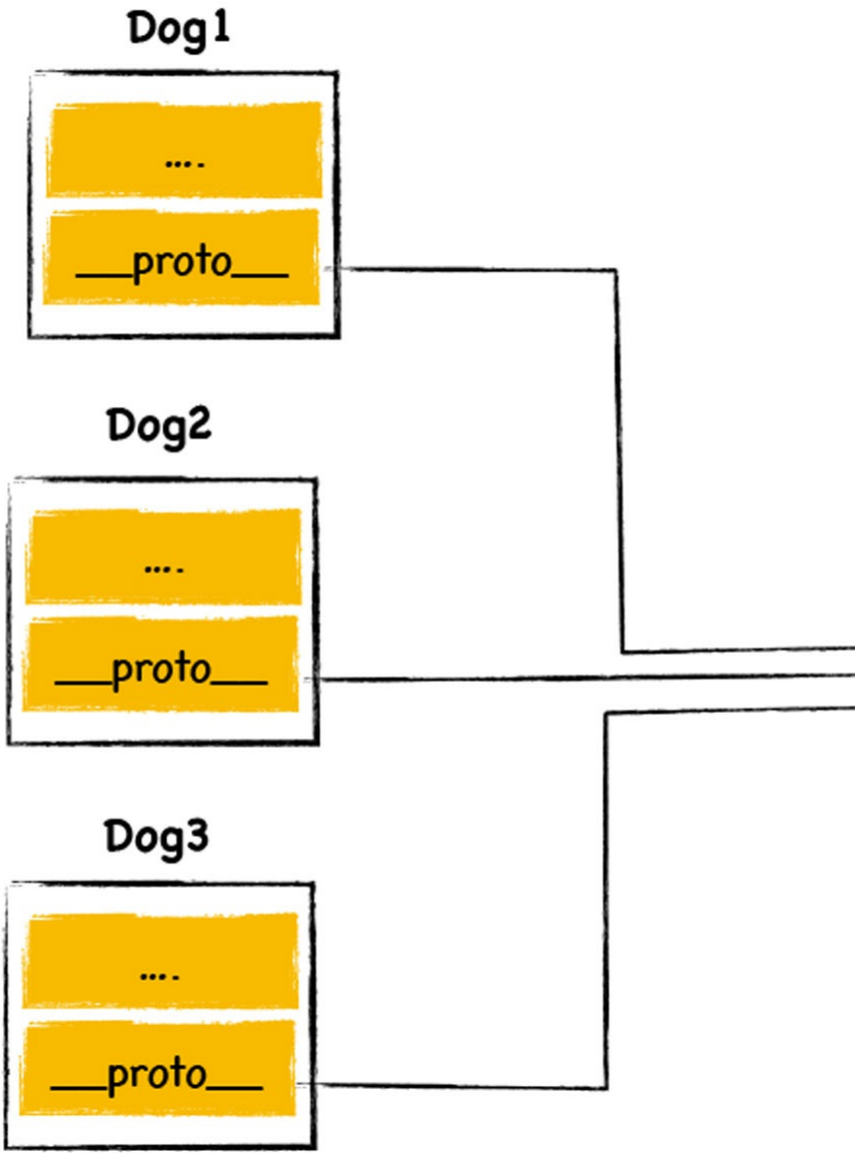
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

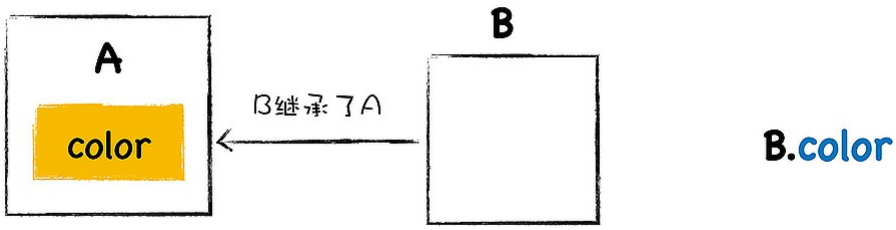
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

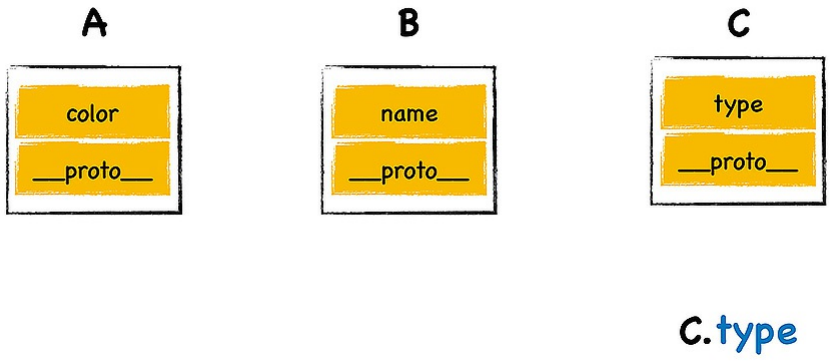
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

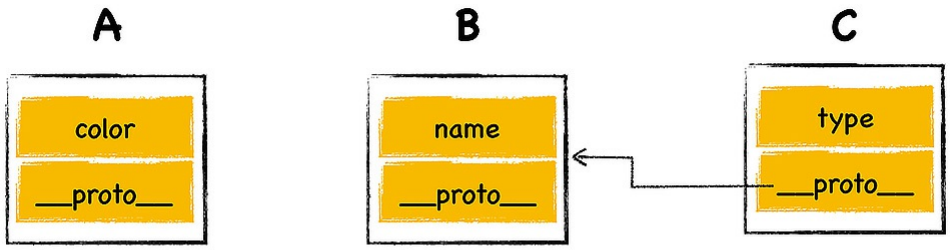


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

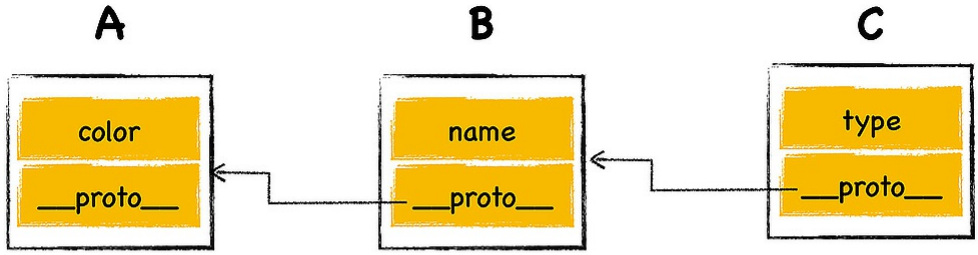


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

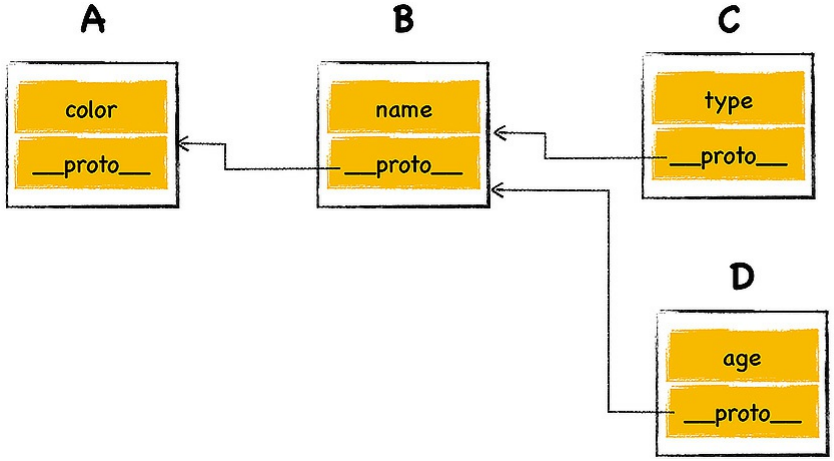
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过*_proto_*来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

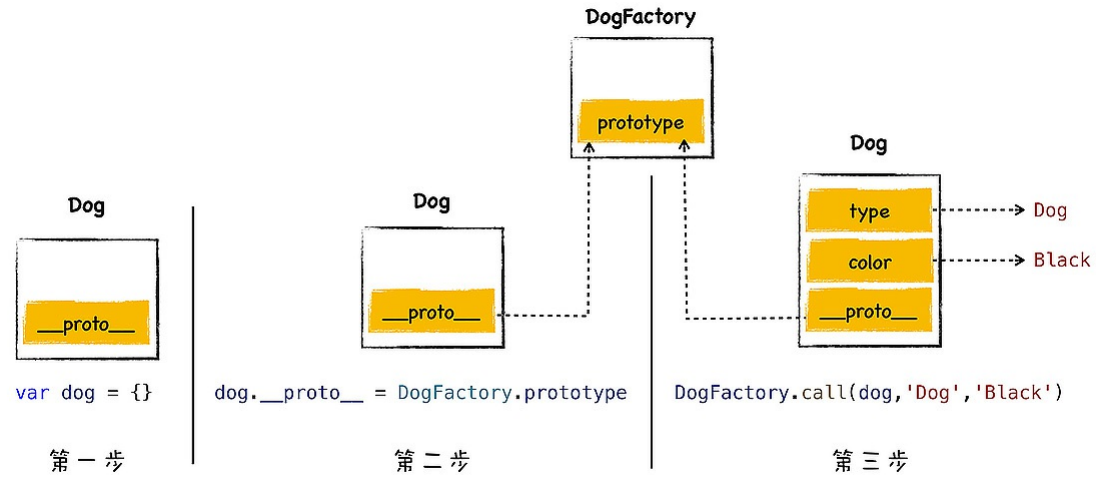
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

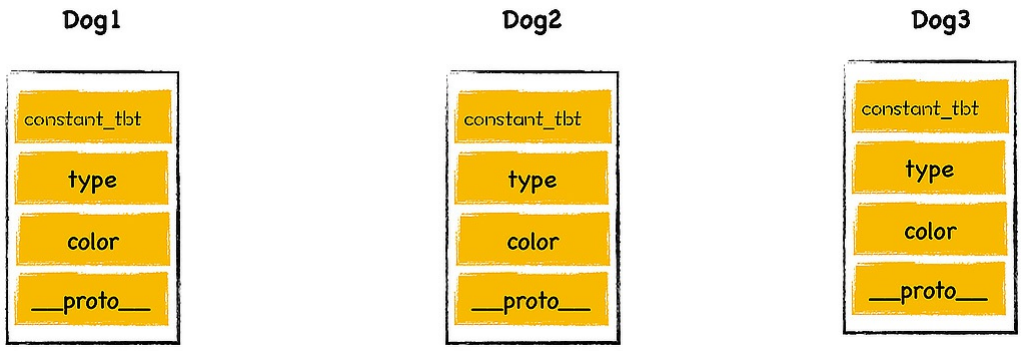
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

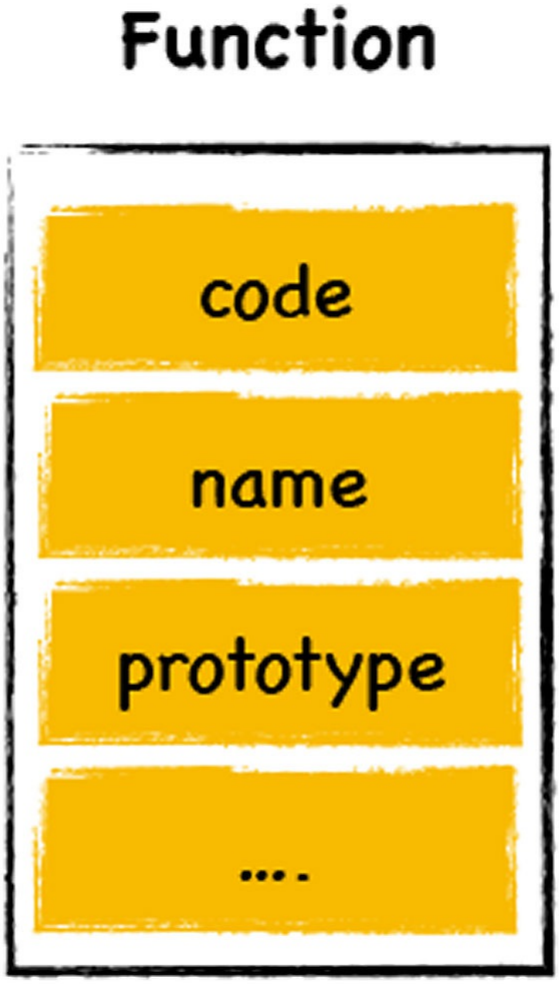
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

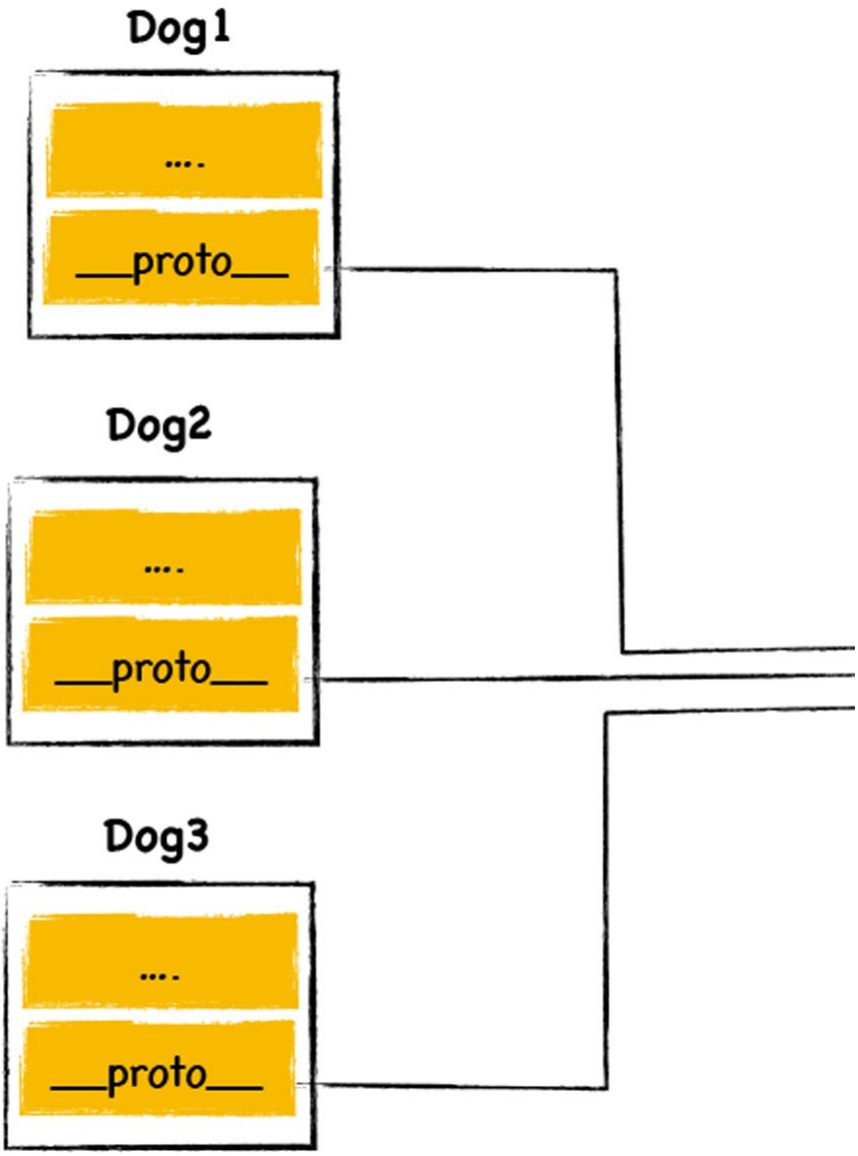
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

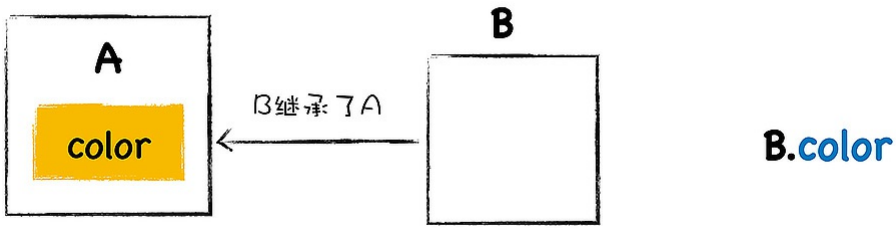
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

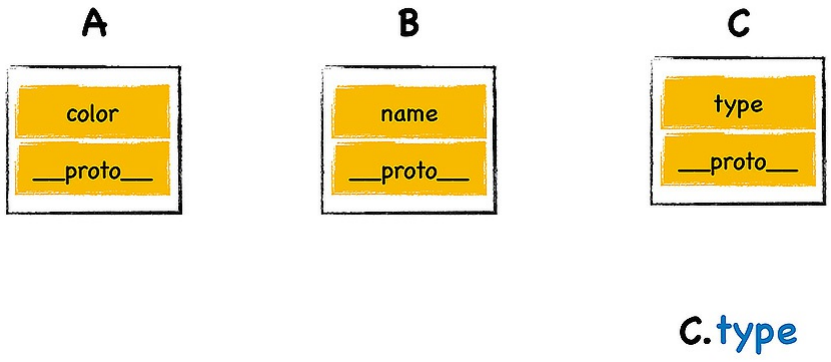
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

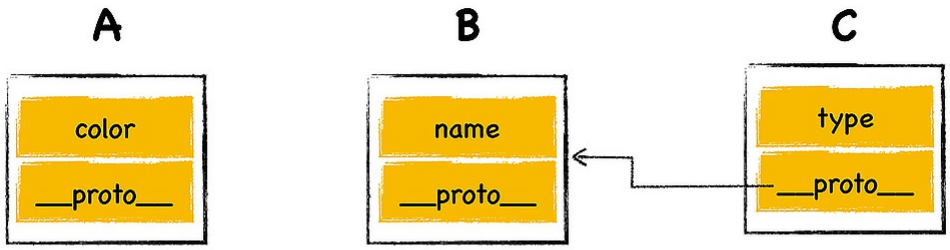


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

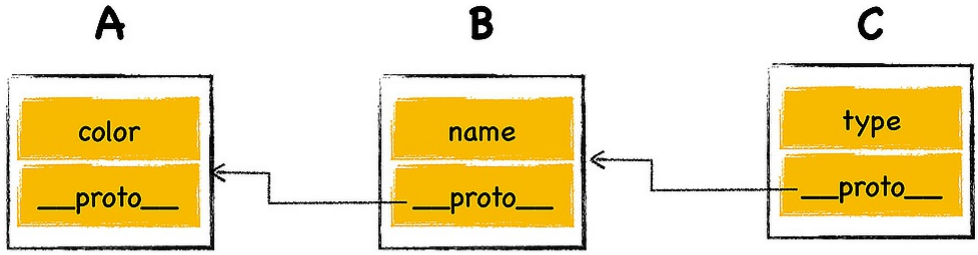


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

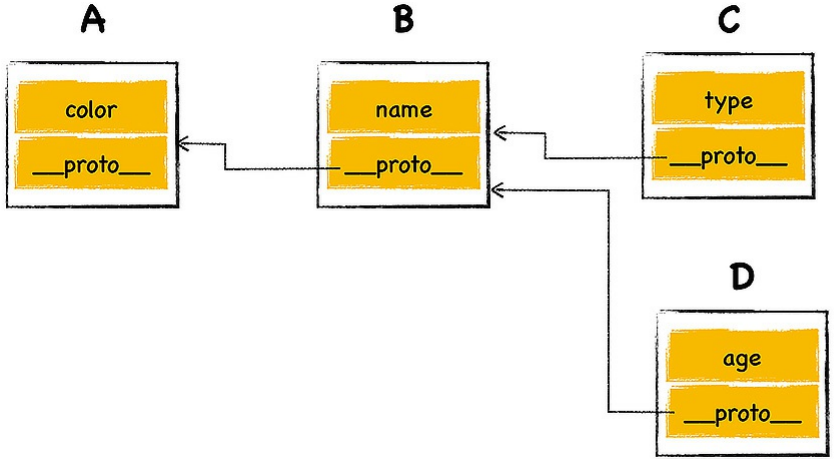
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过*_proto_*来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

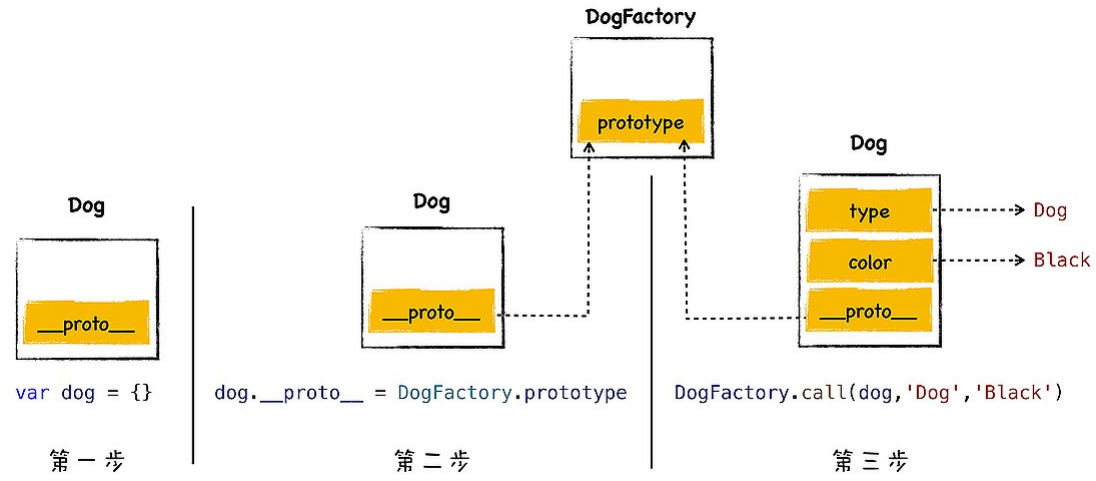
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

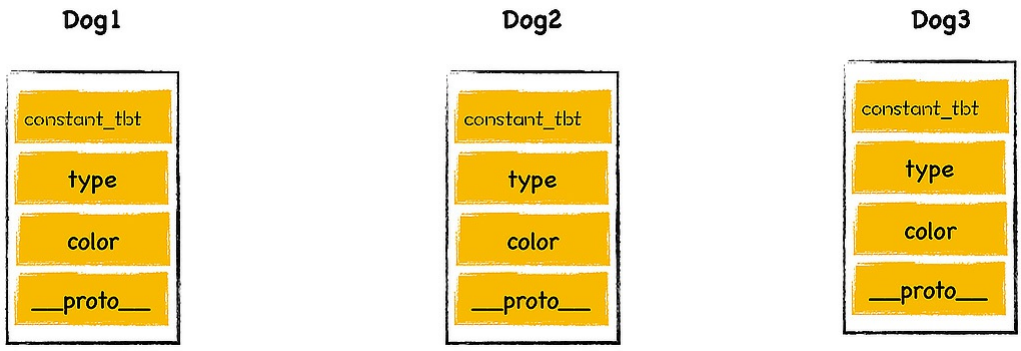
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

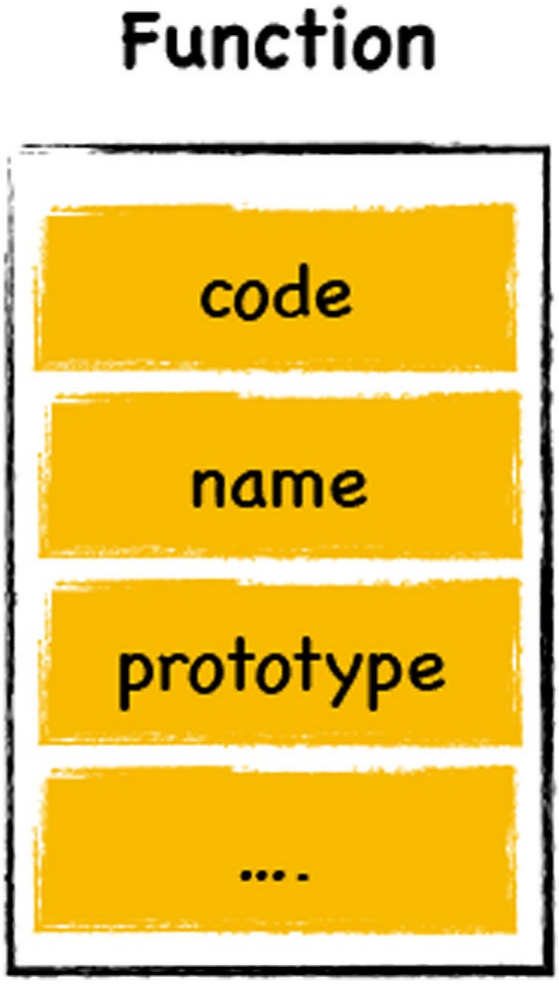
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

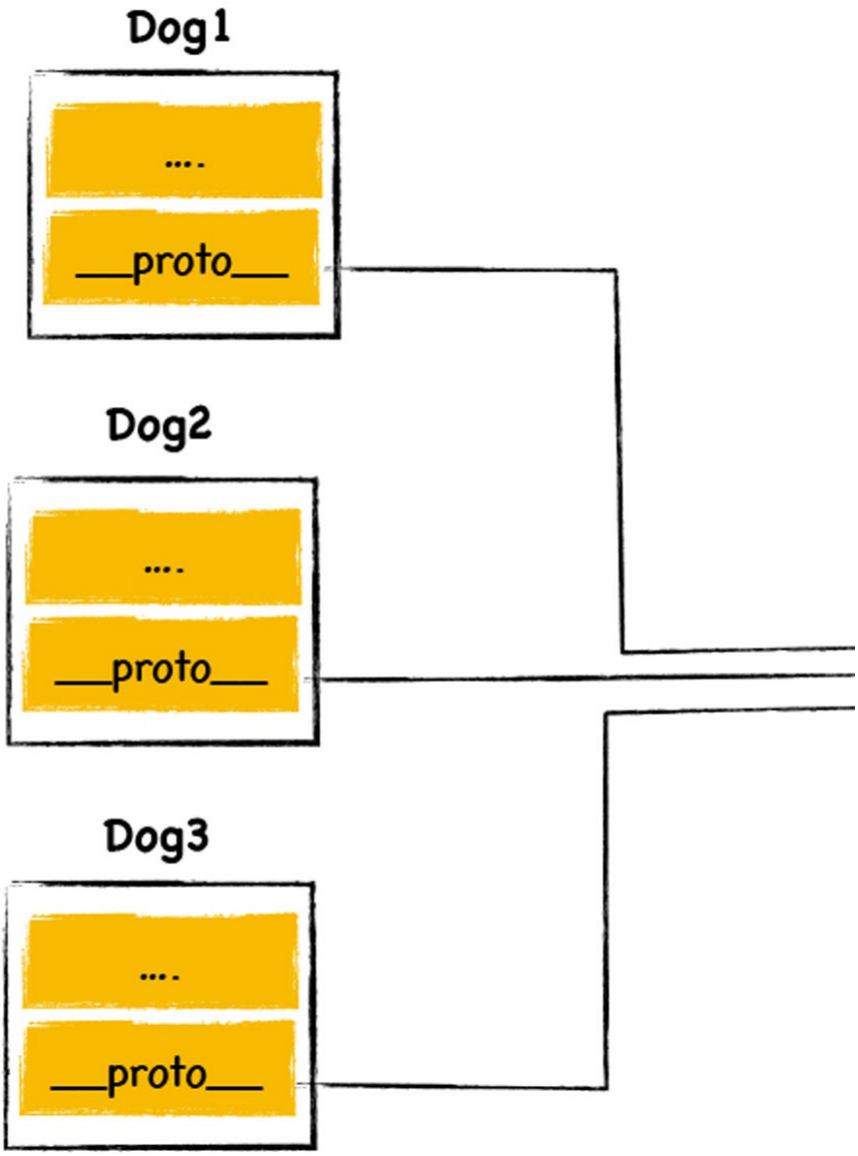
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

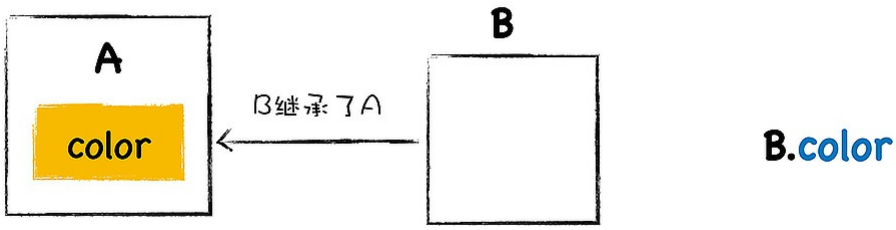
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

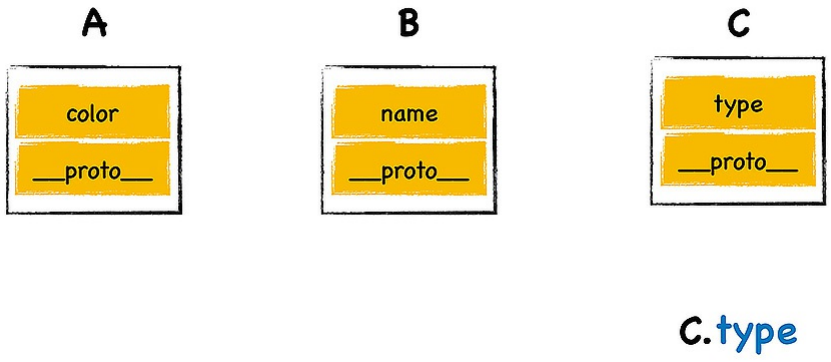
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

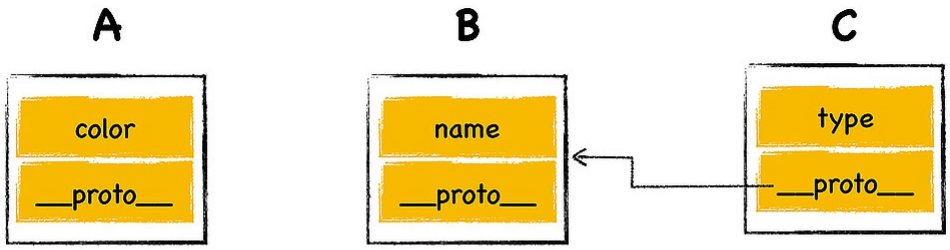


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

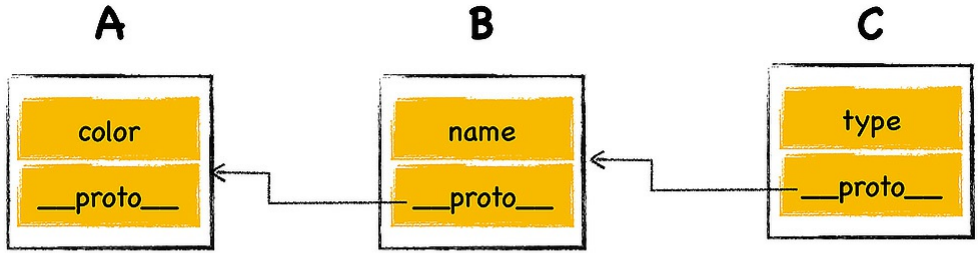


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

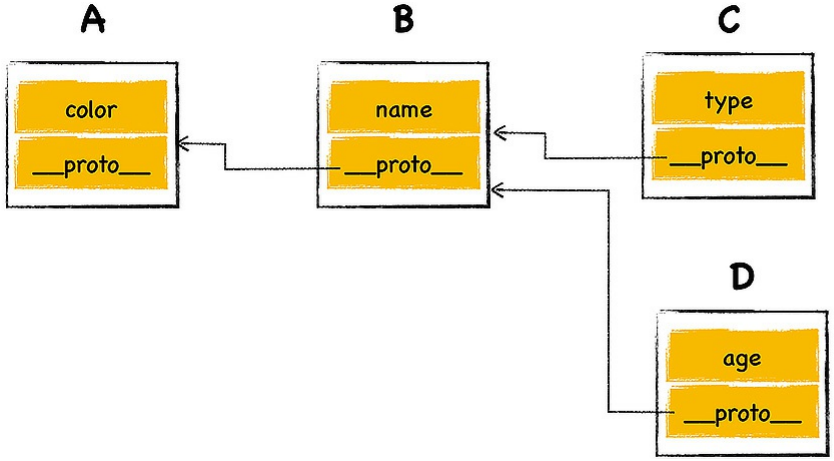
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过*_proto_*来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

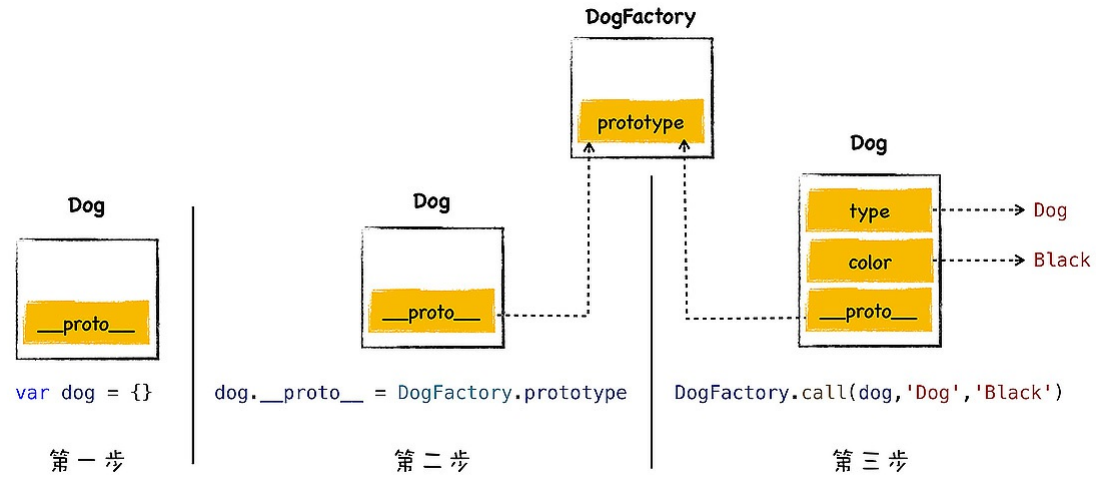
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

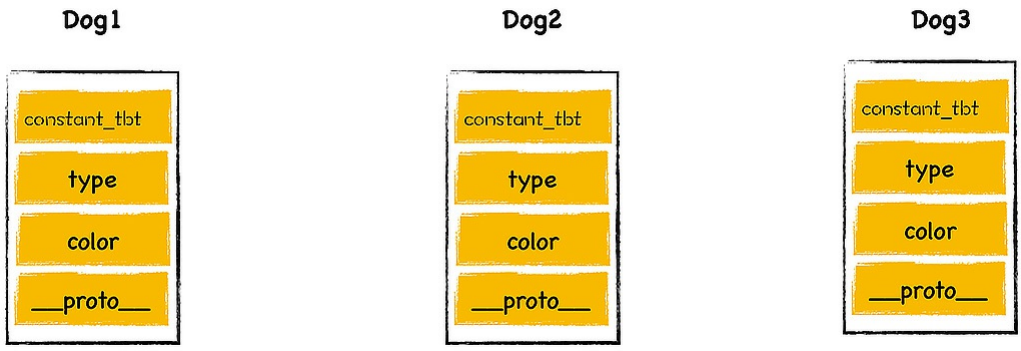
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

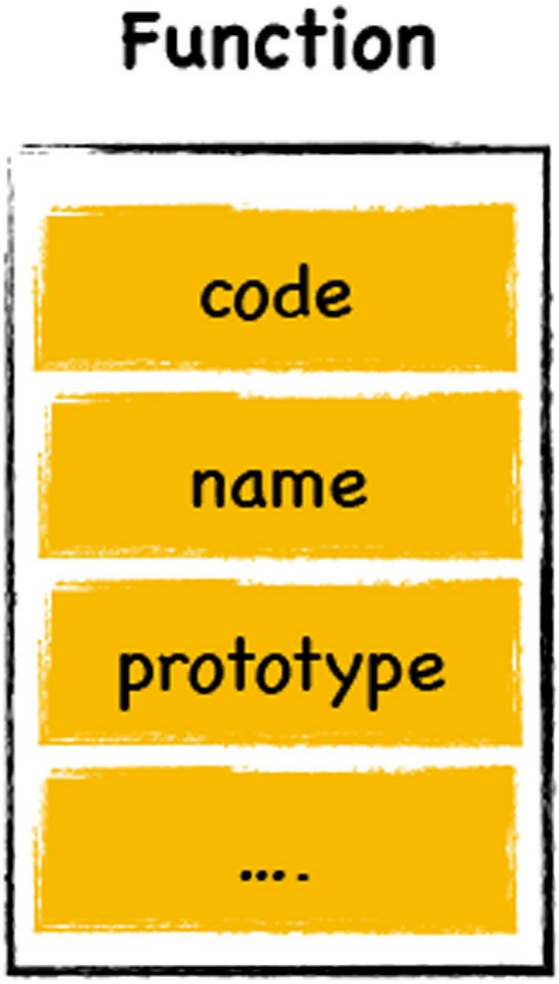
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

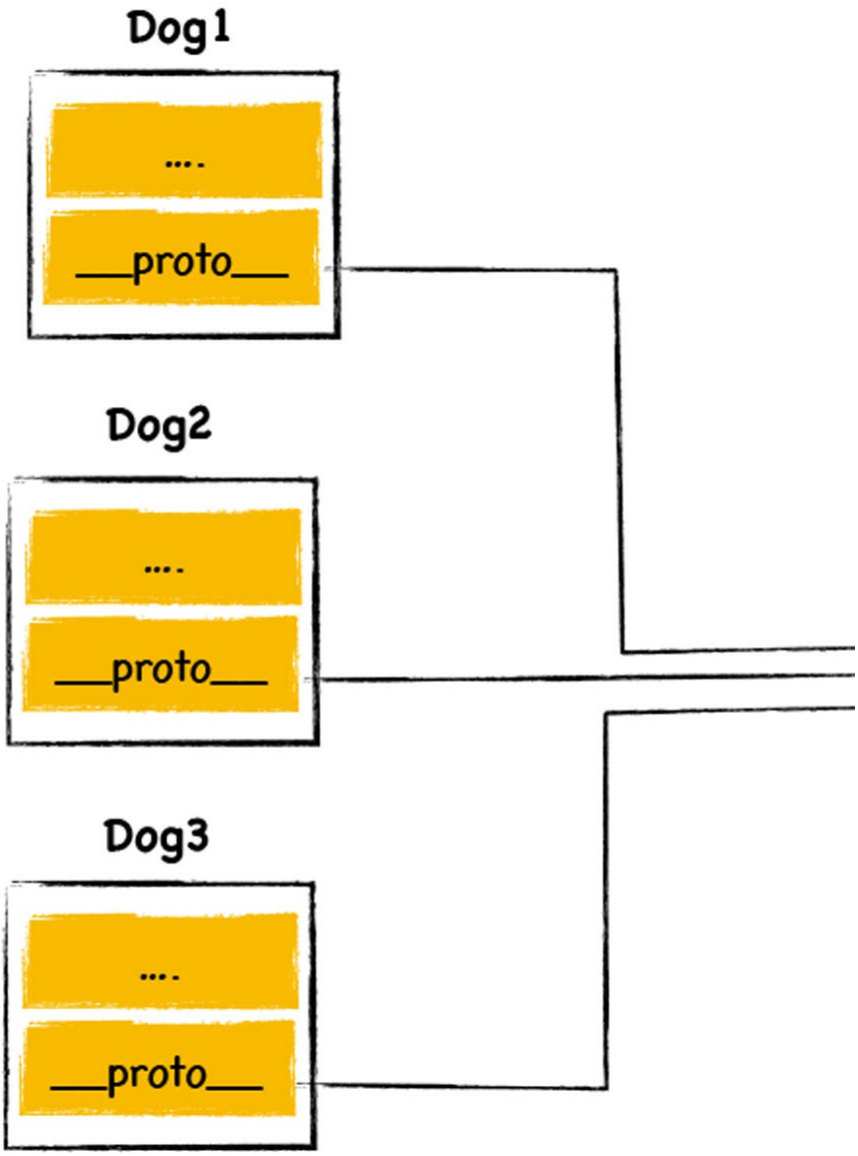
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

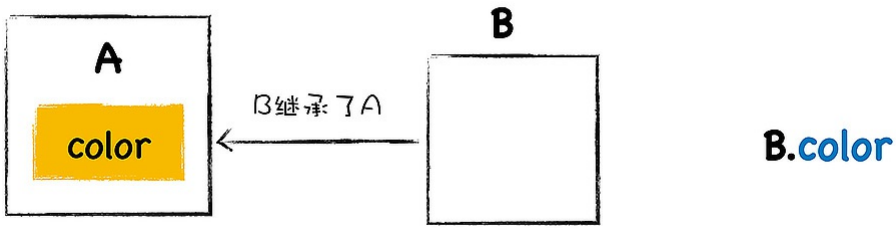
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

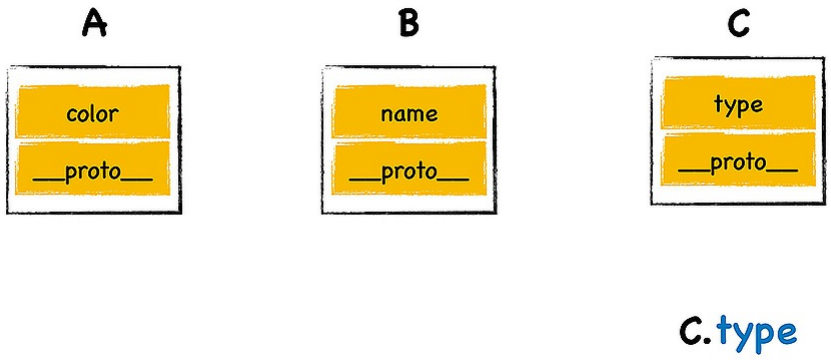
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

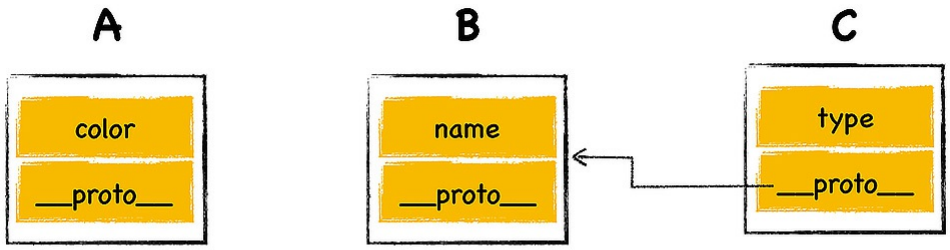


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

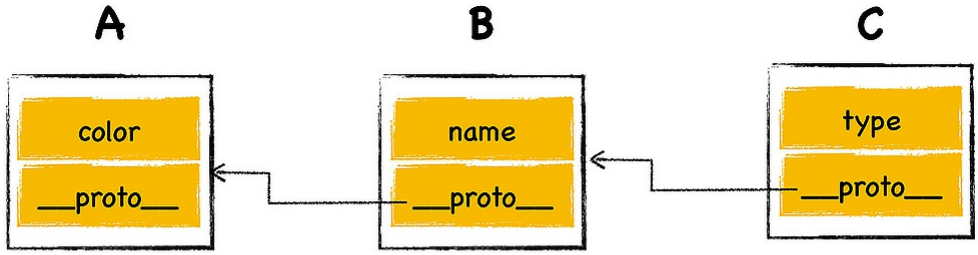


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

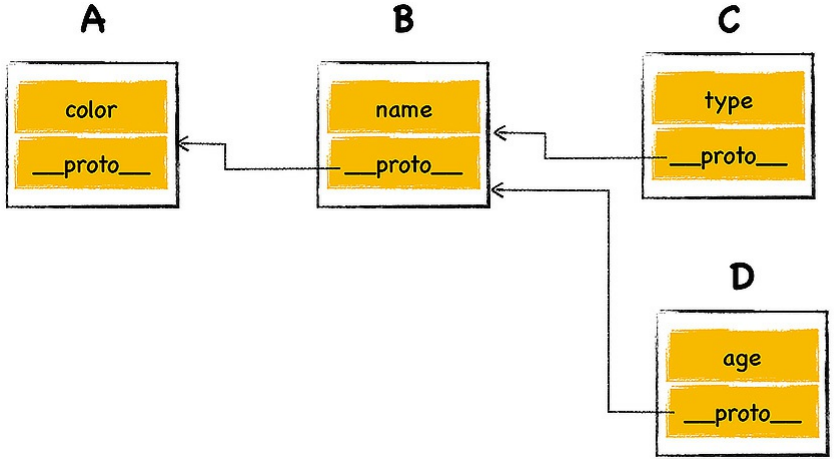
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过 *_proto_* 来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

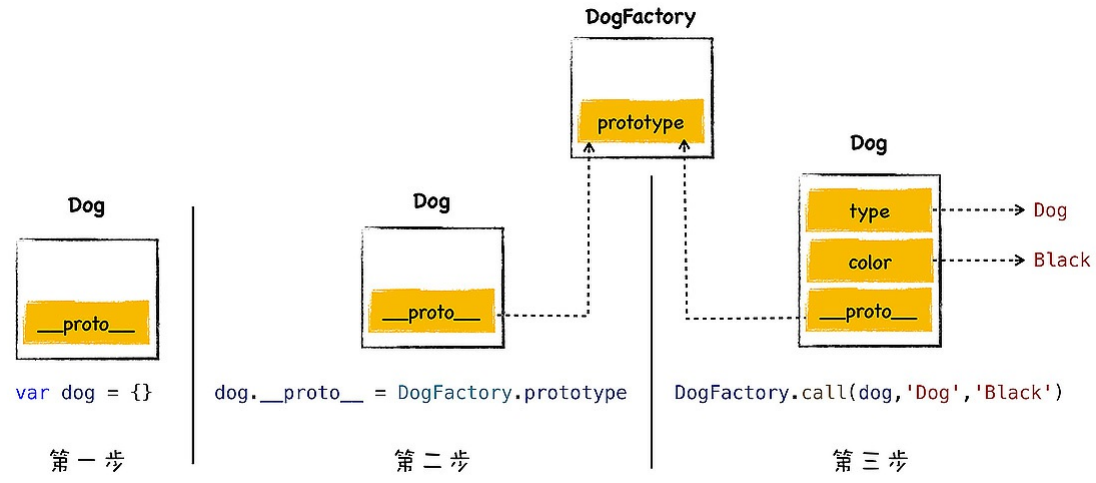
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

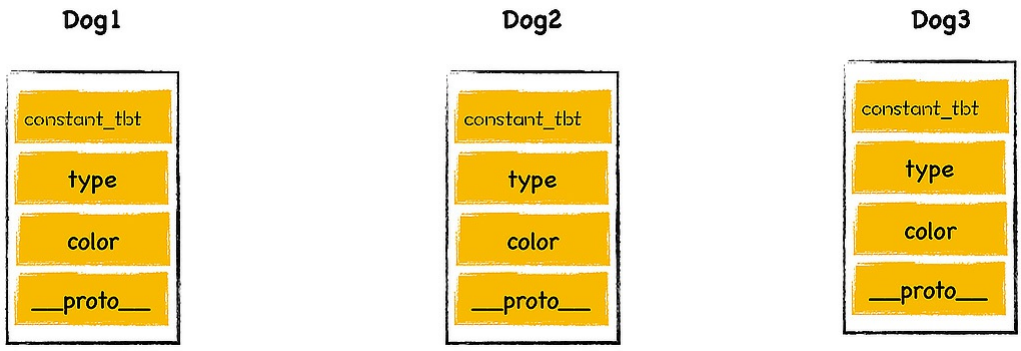
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

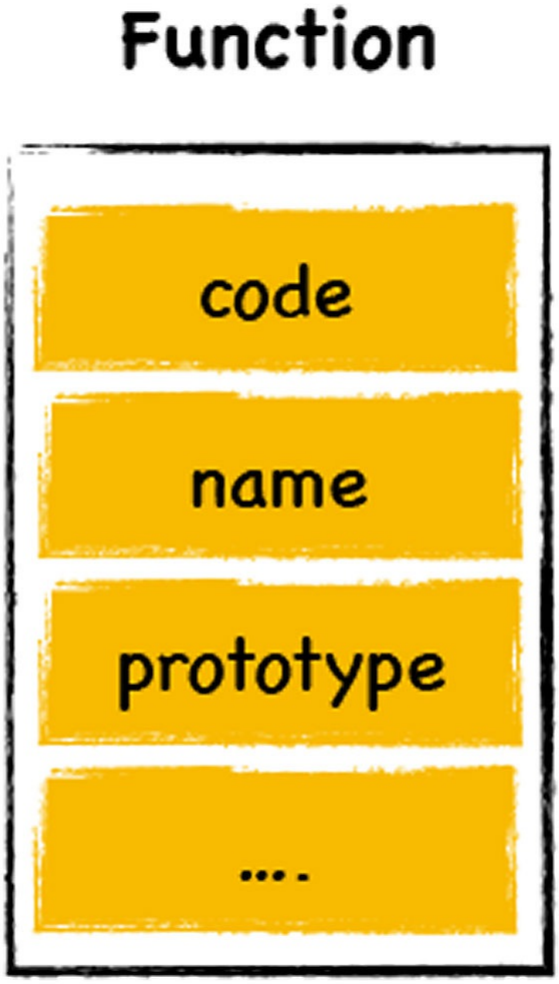
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

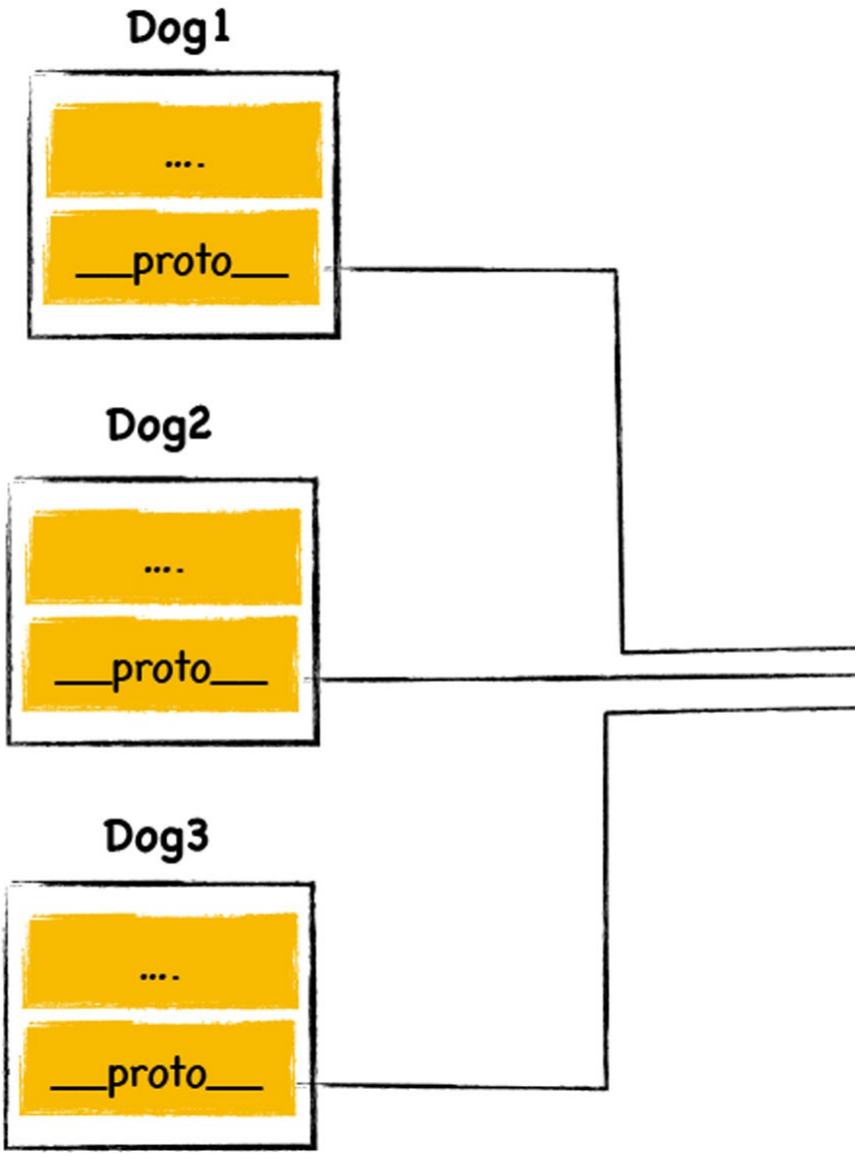
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

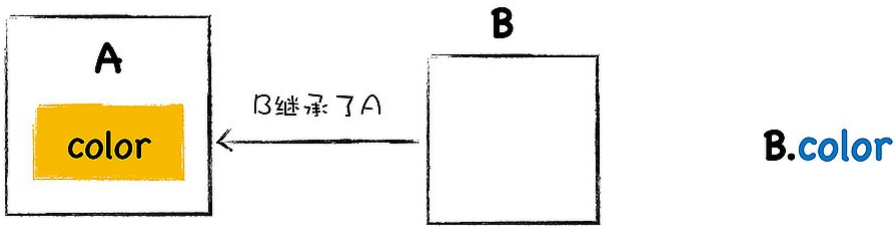
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

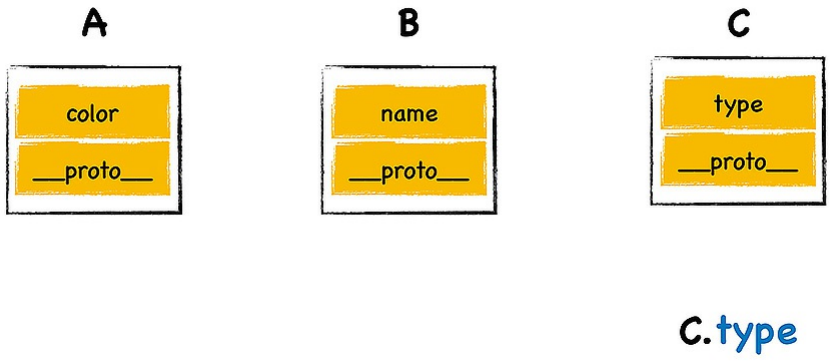
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

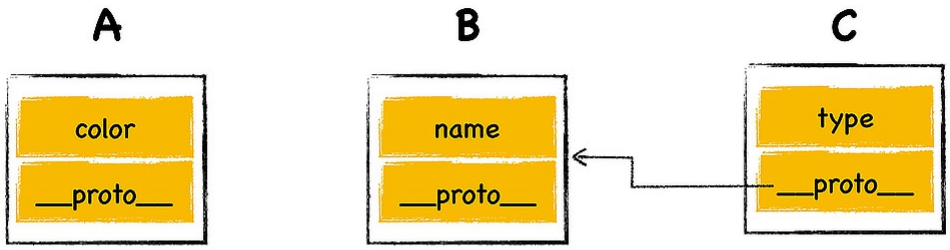


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

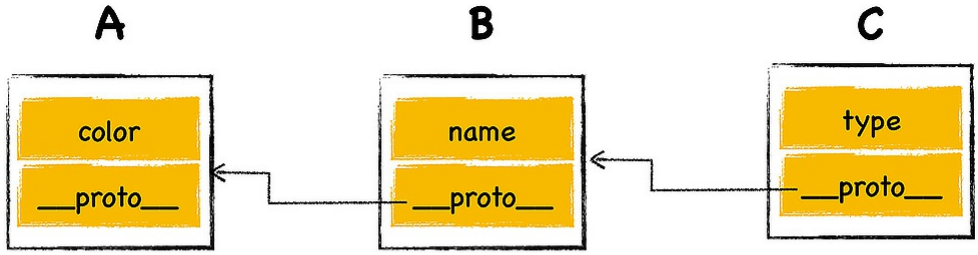


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

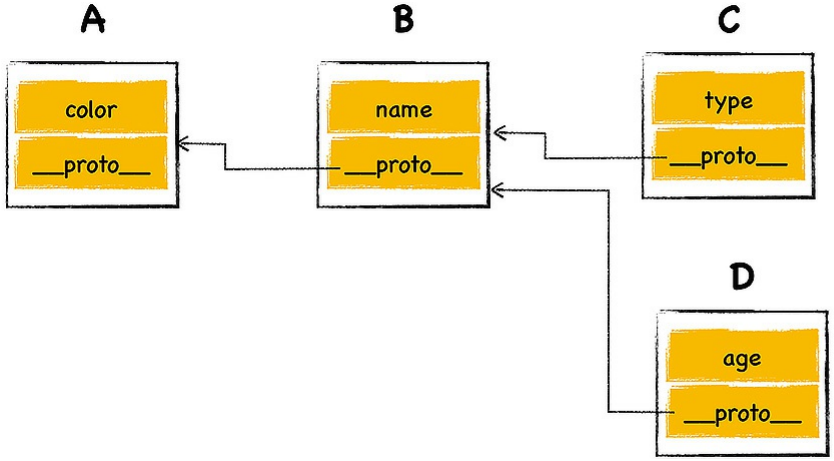
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过 *_proto_* 来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

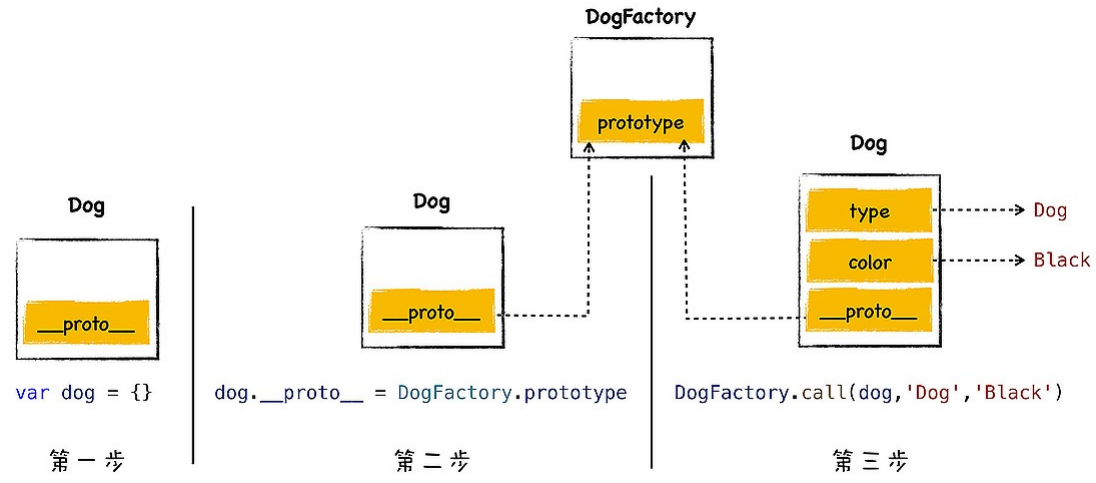
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

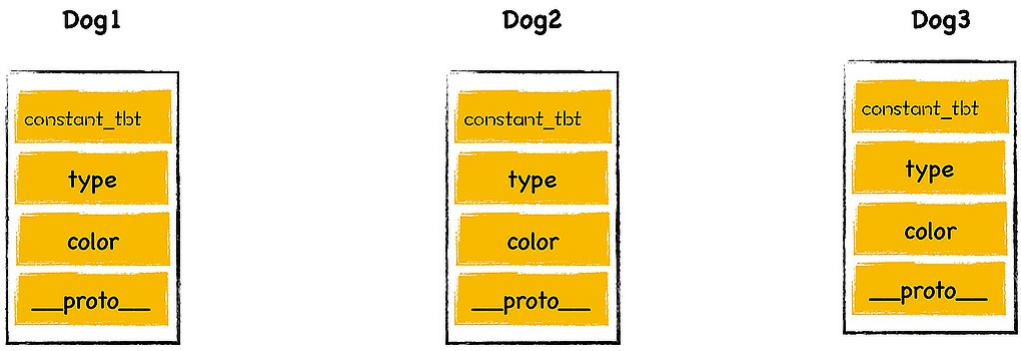
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

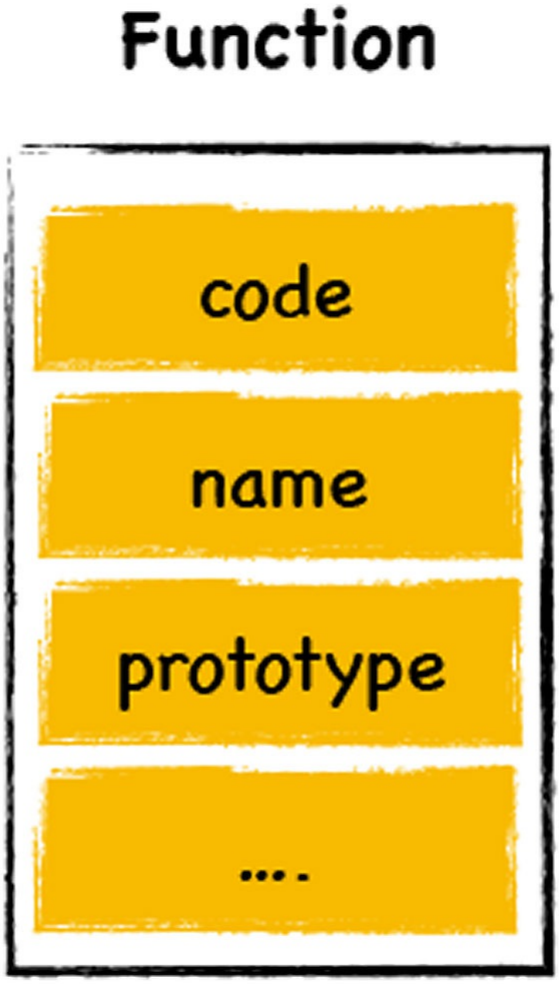
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

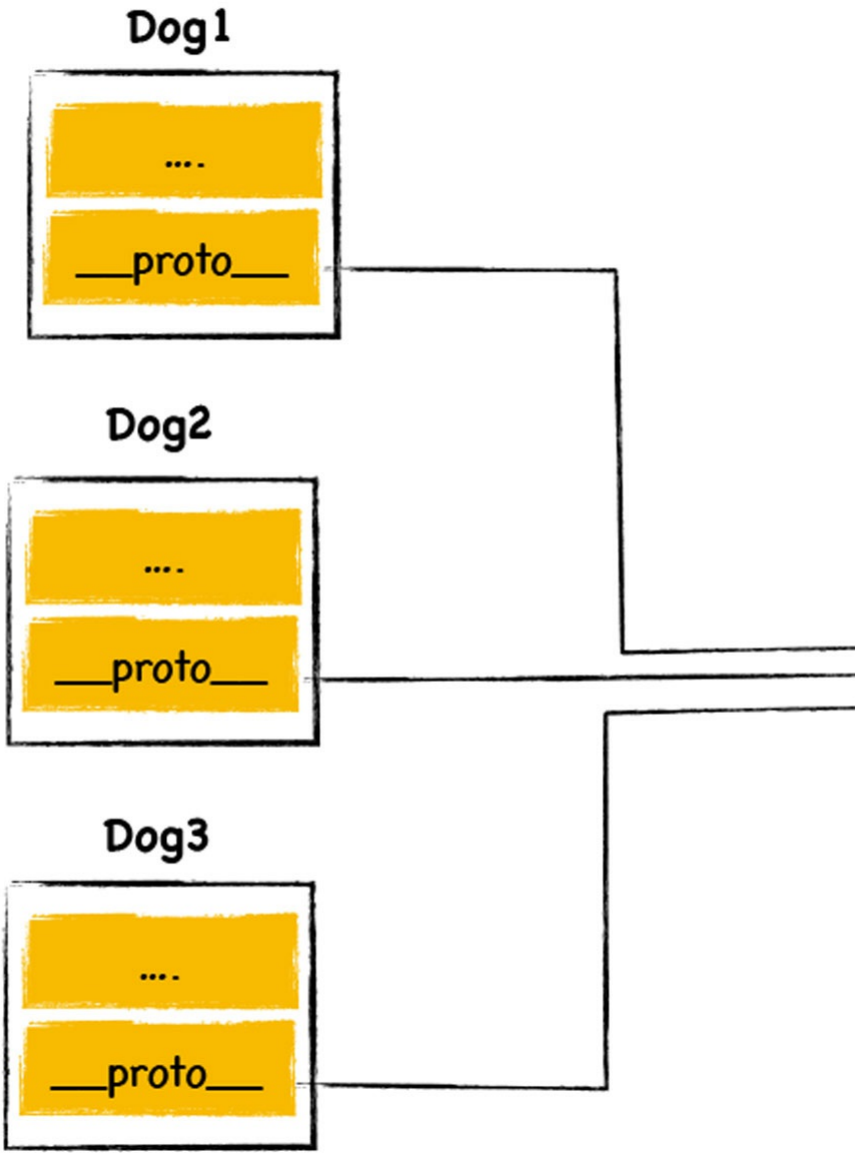
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

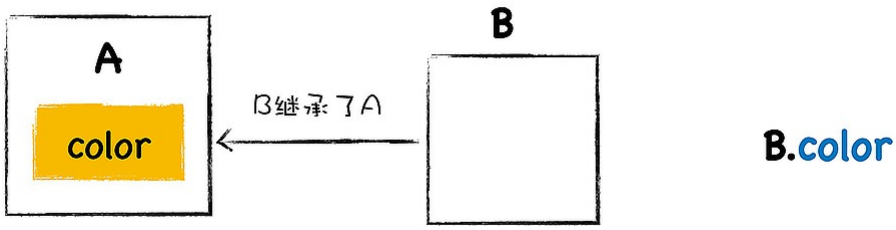
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

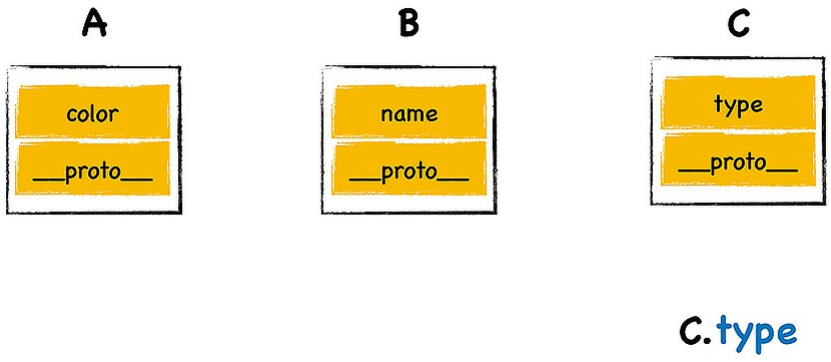
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

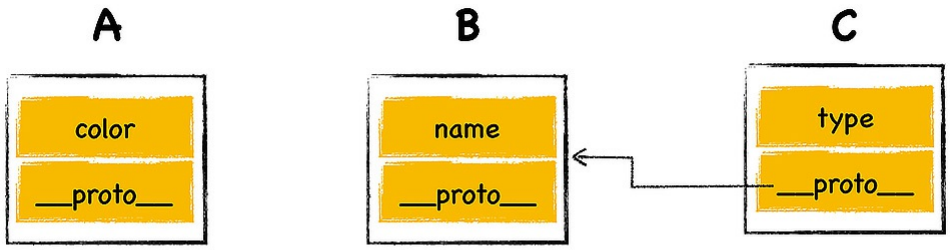


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

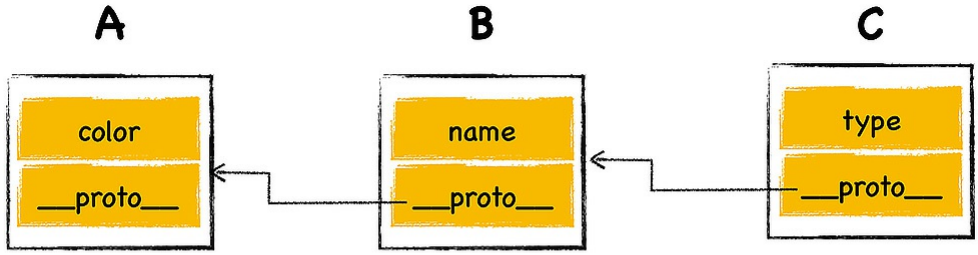


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

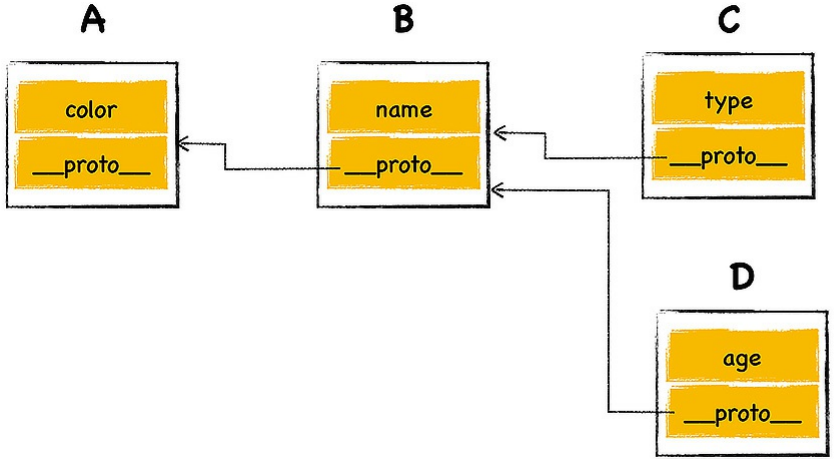
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过*_proto_*来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

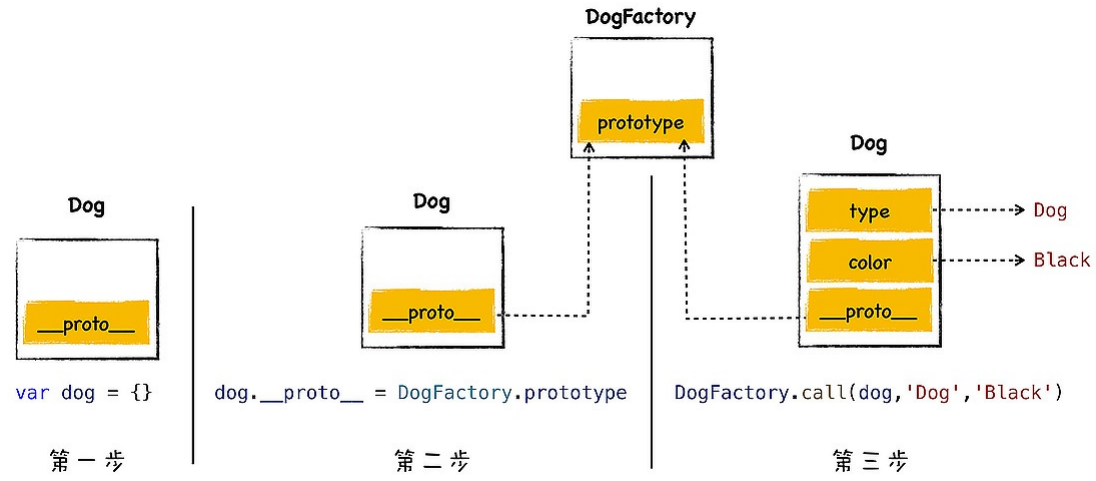
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

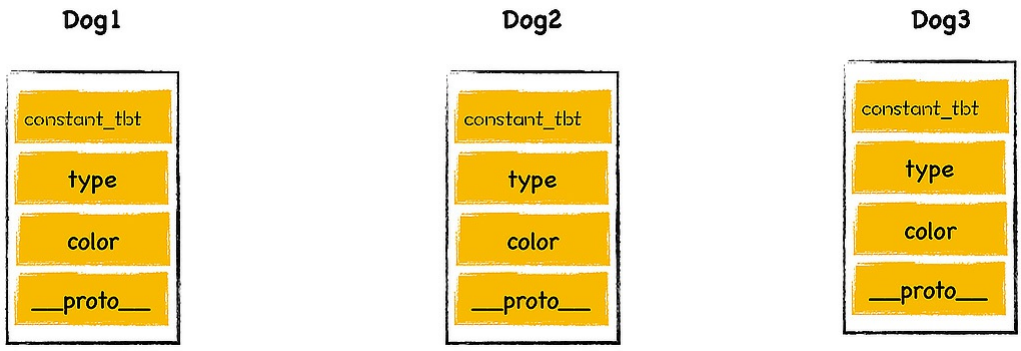
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

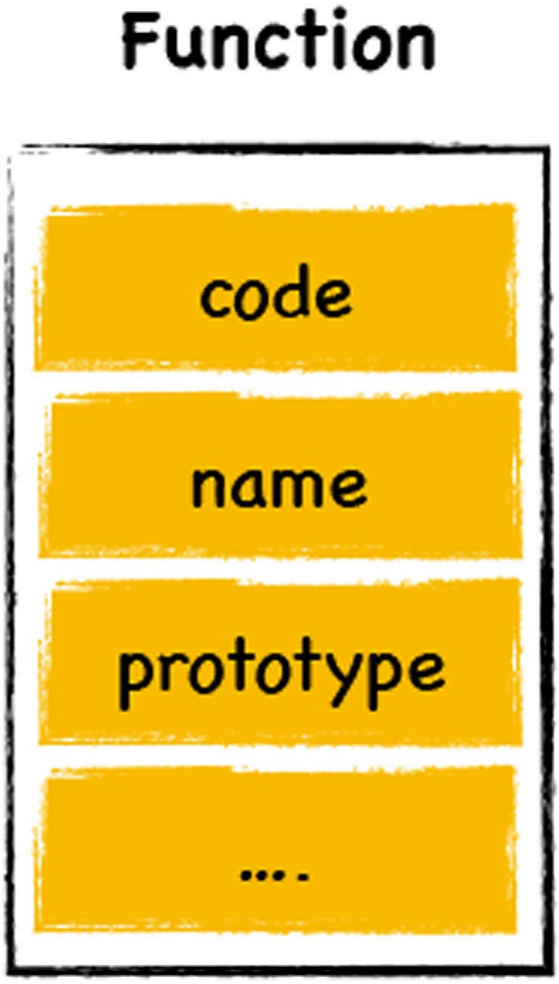
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

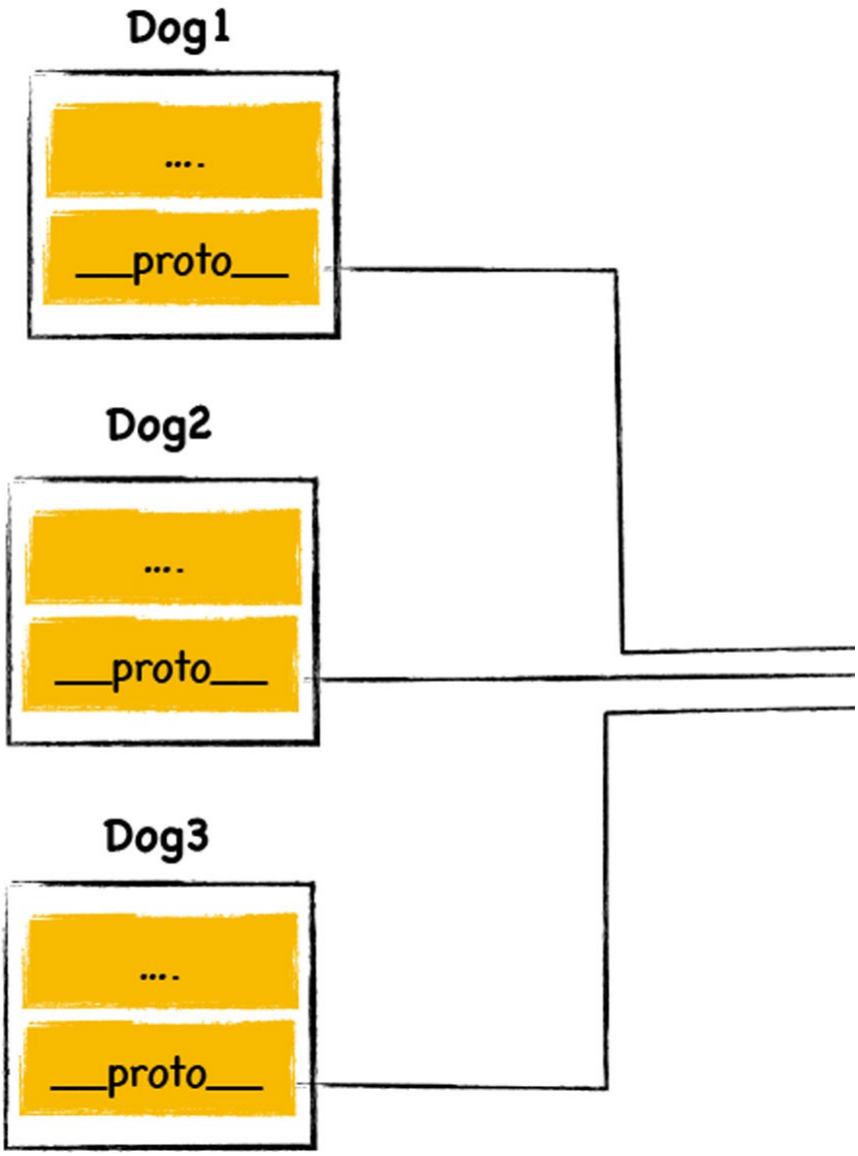
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

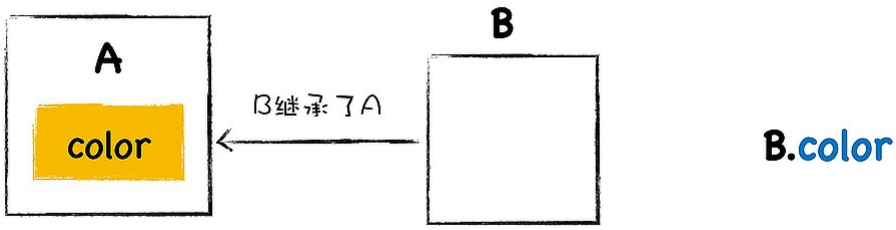
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

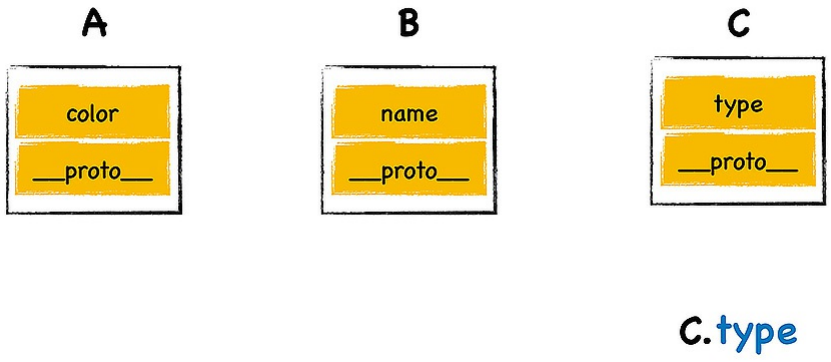
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

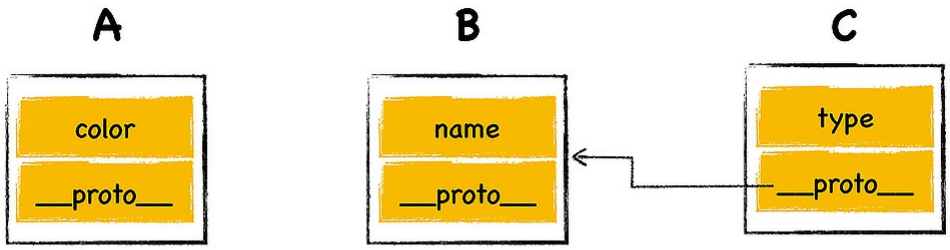


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

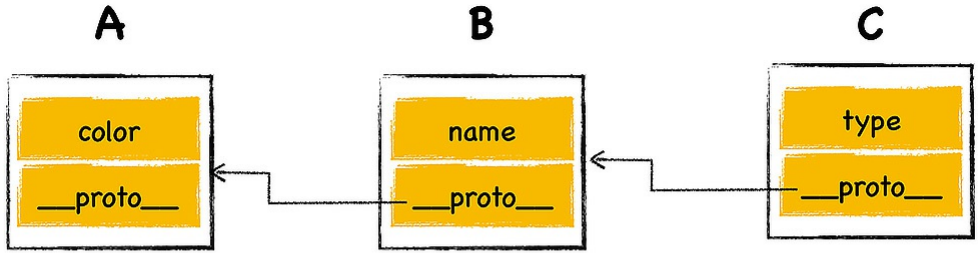


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

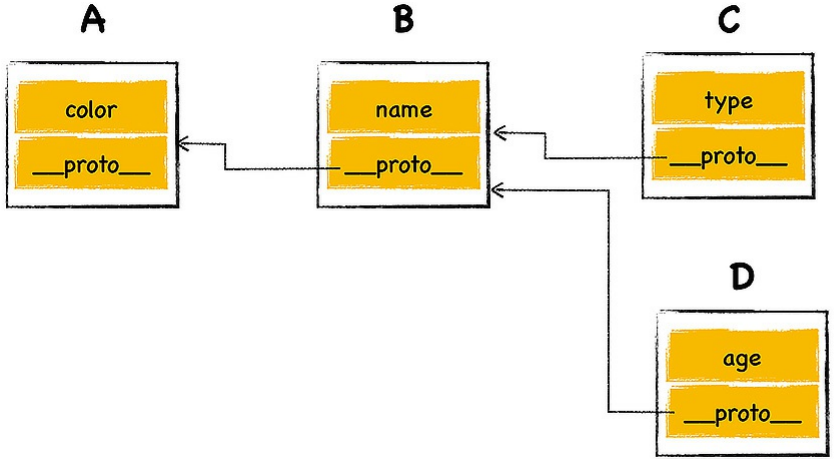
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过 *_proto_* 来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

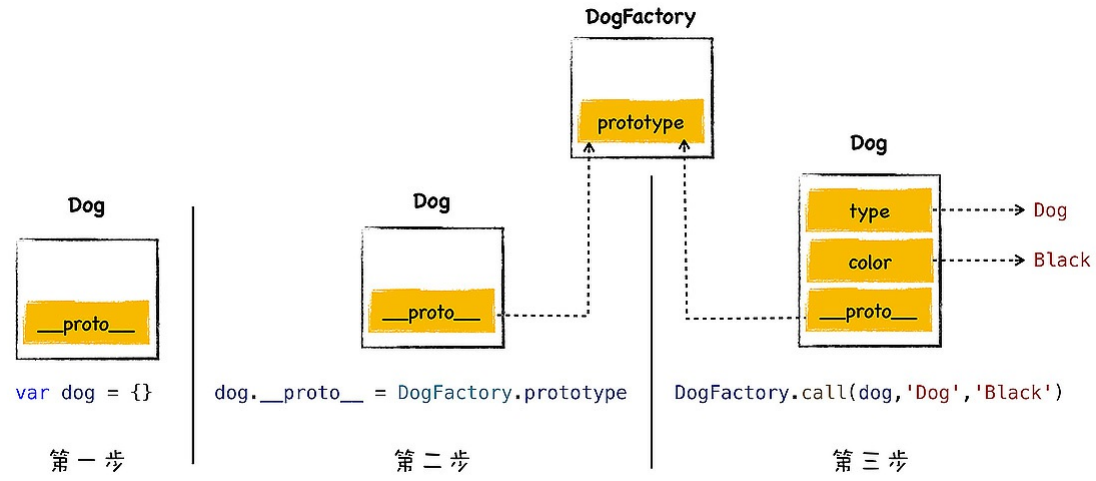
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

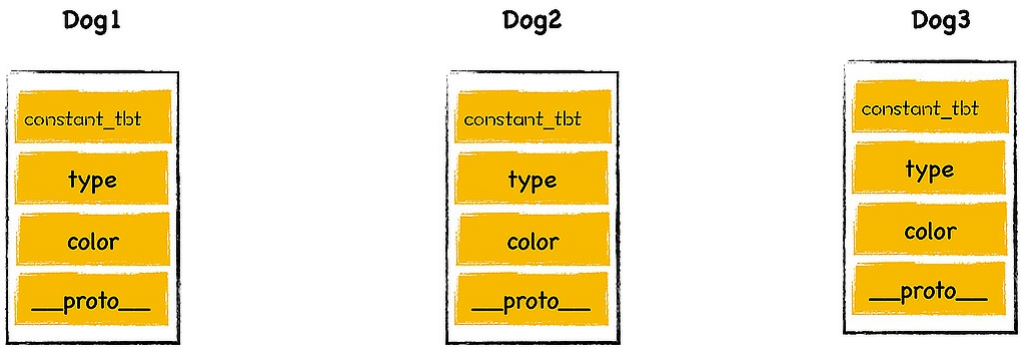
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

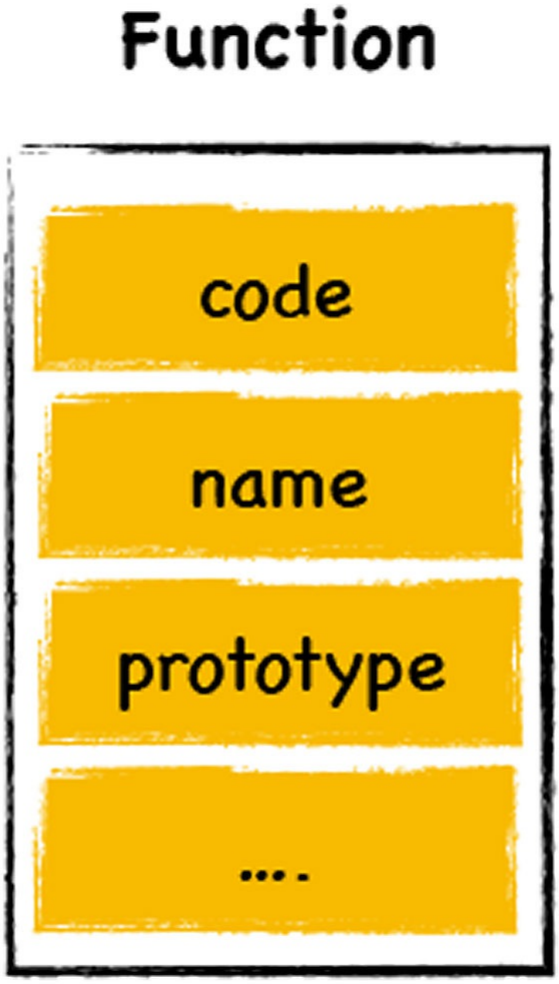
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

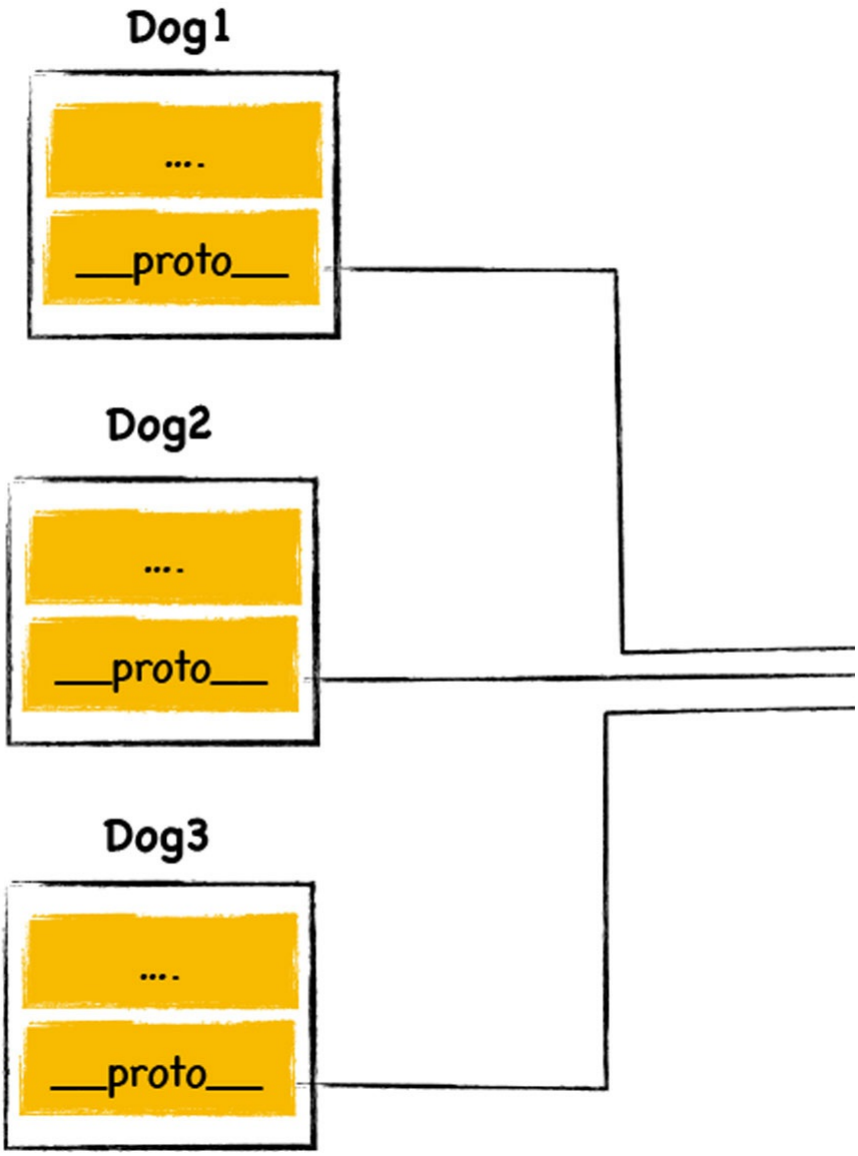
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

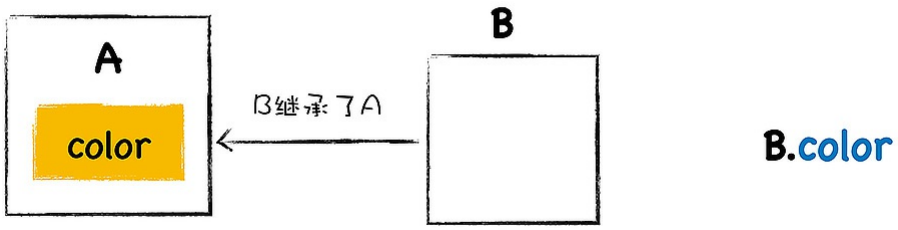
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

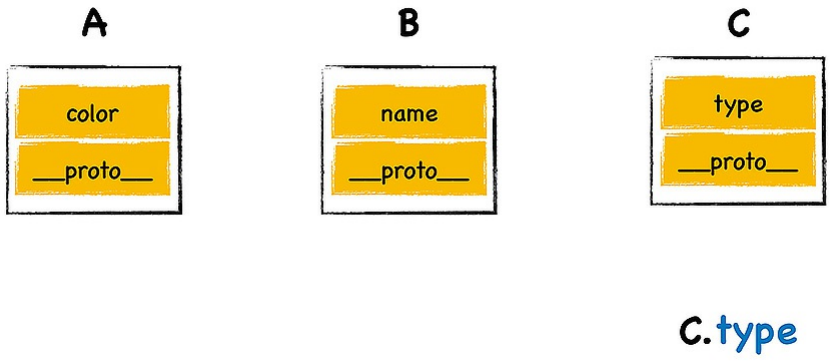
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

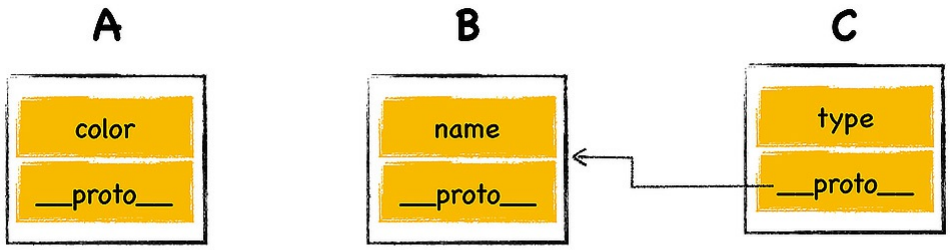


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

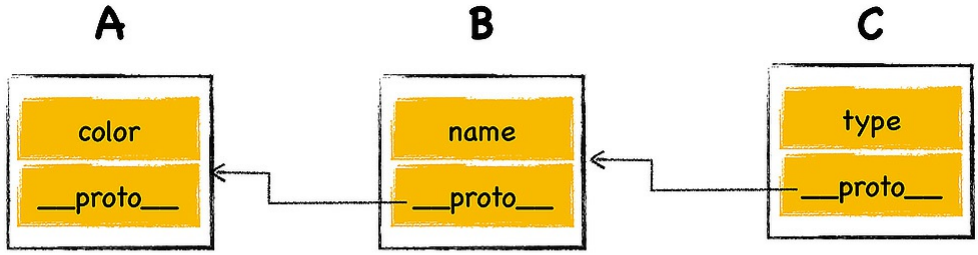


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

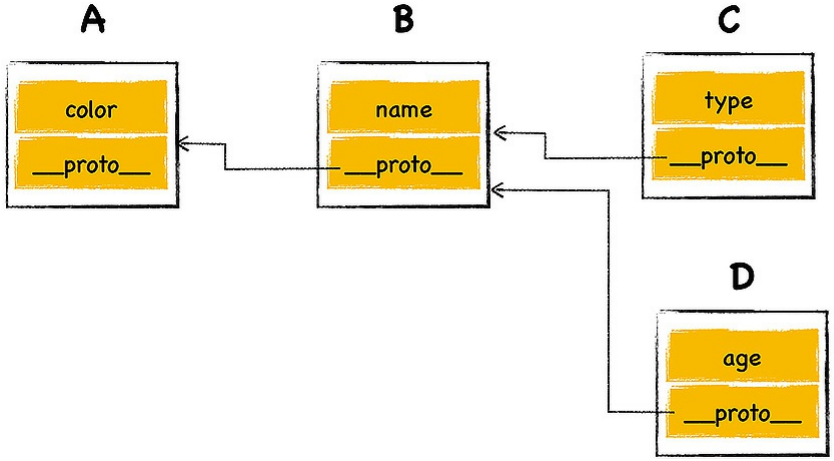
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过 *_proto_* 来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

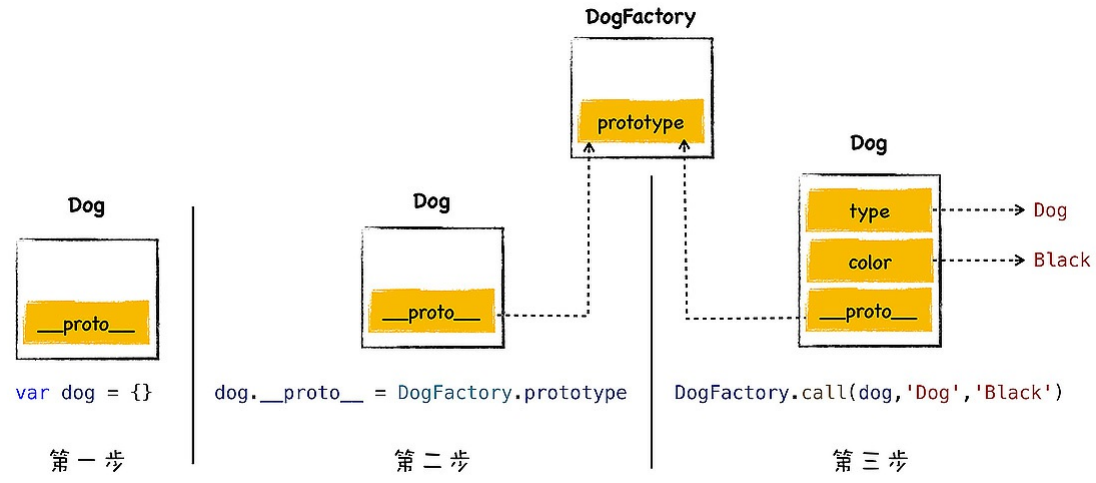
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

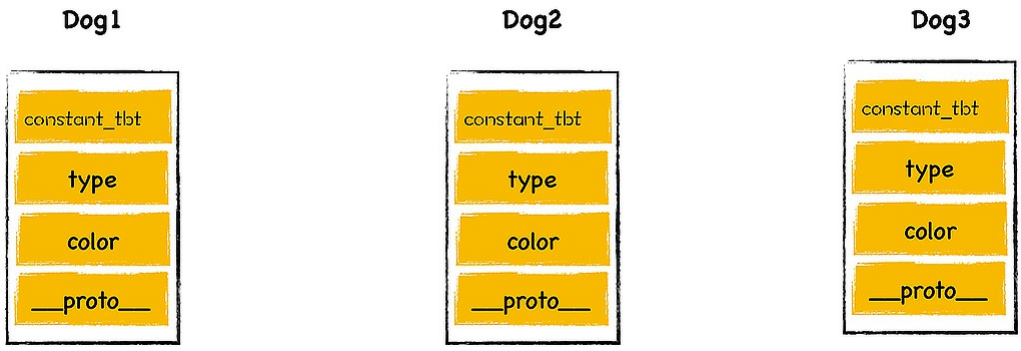
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

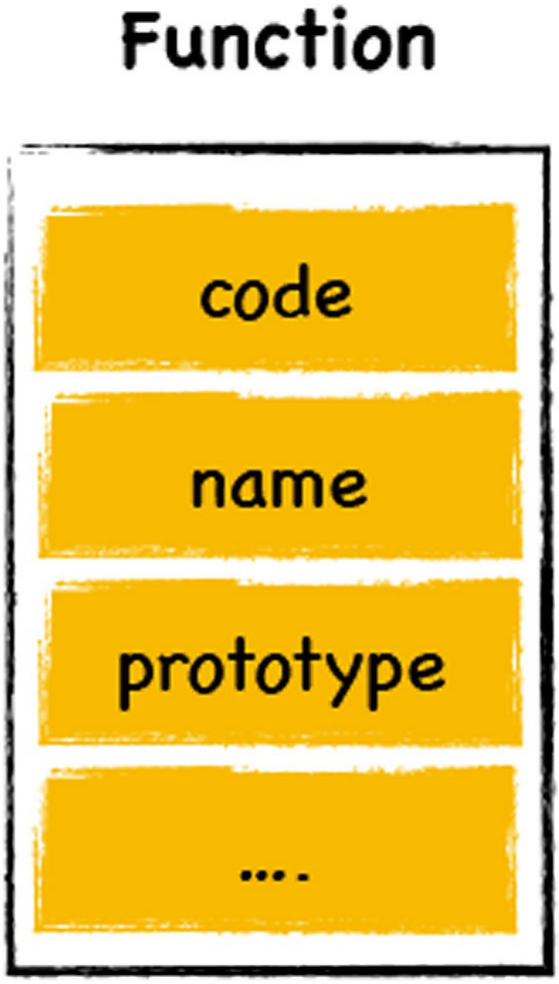
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

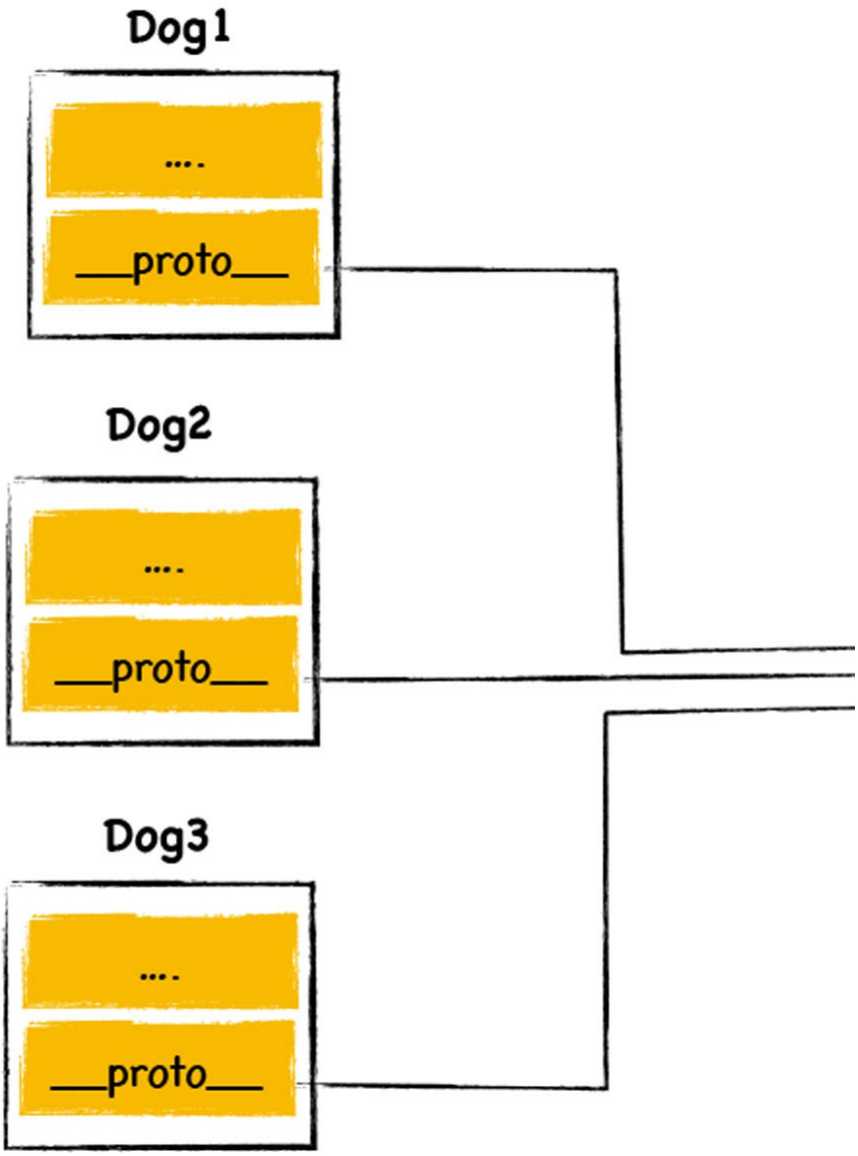
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

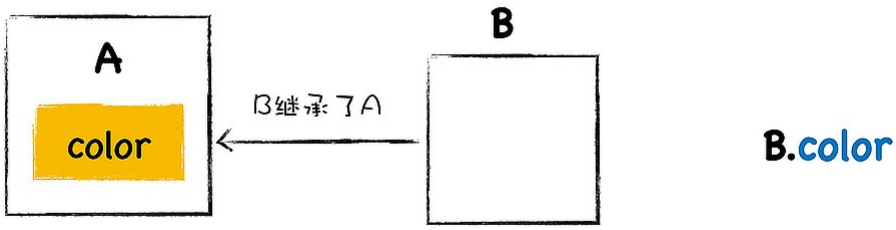
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

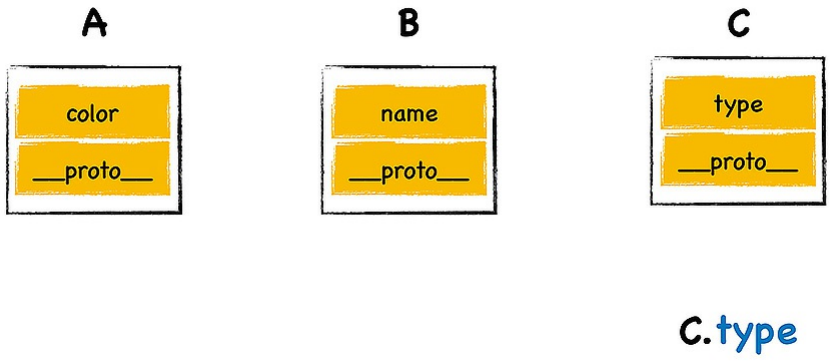
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

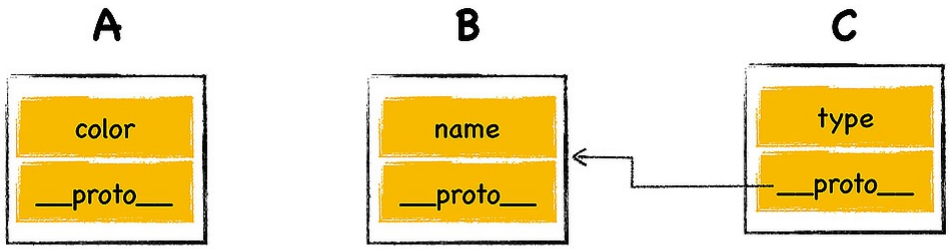


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

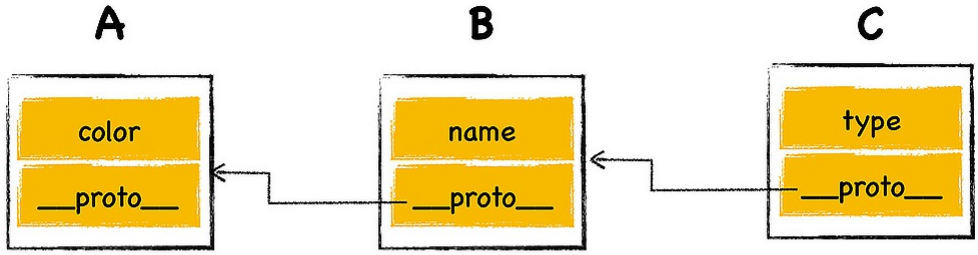


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

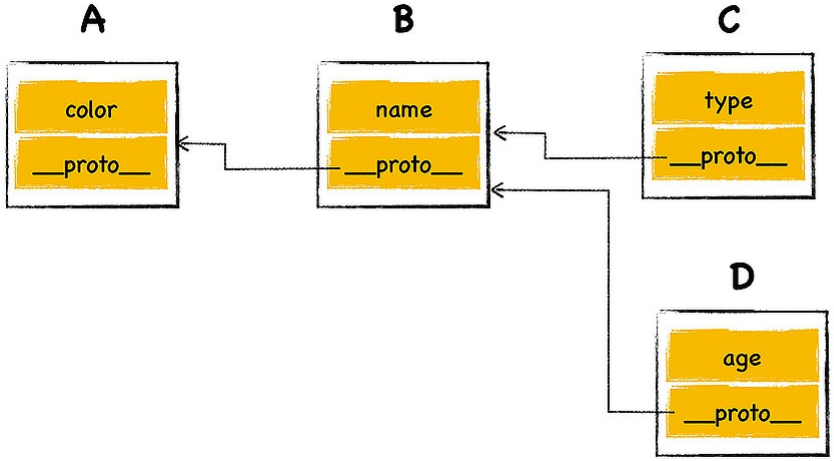
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过*_proto_*来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

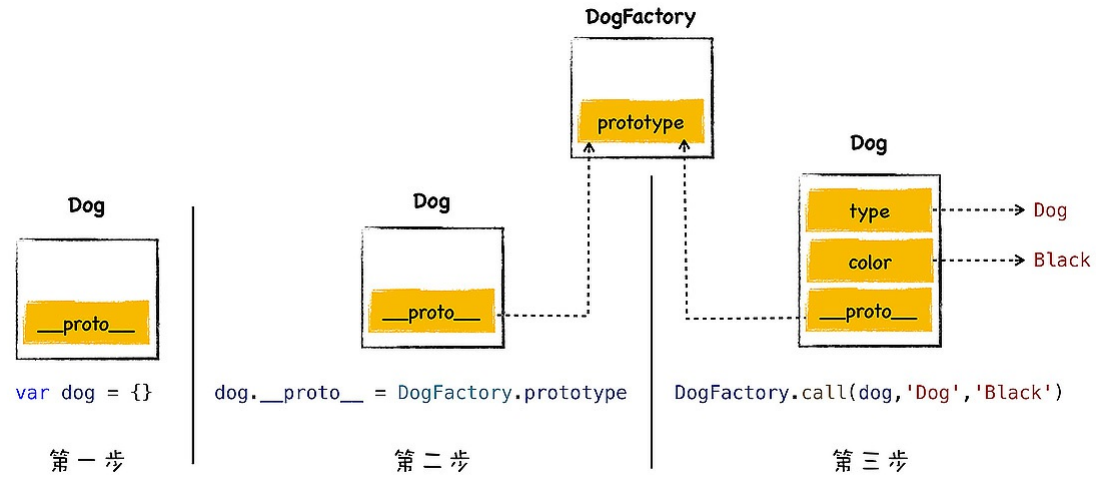
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

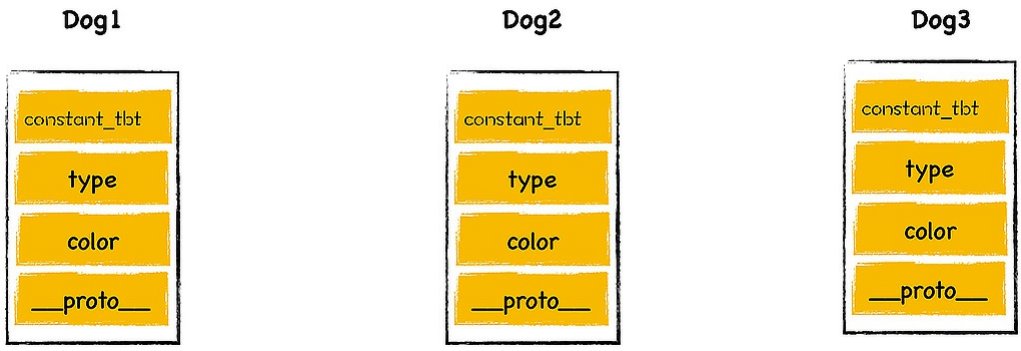
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

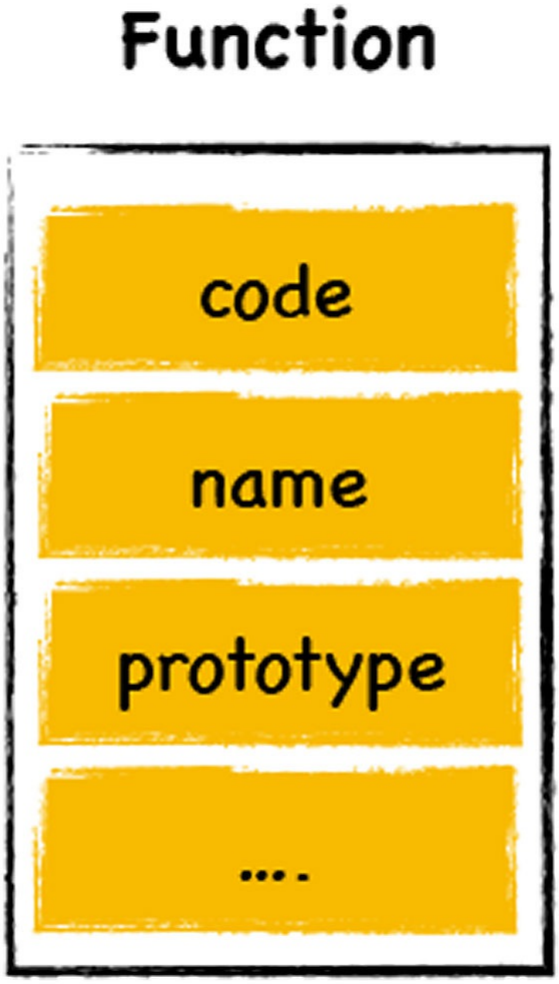
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

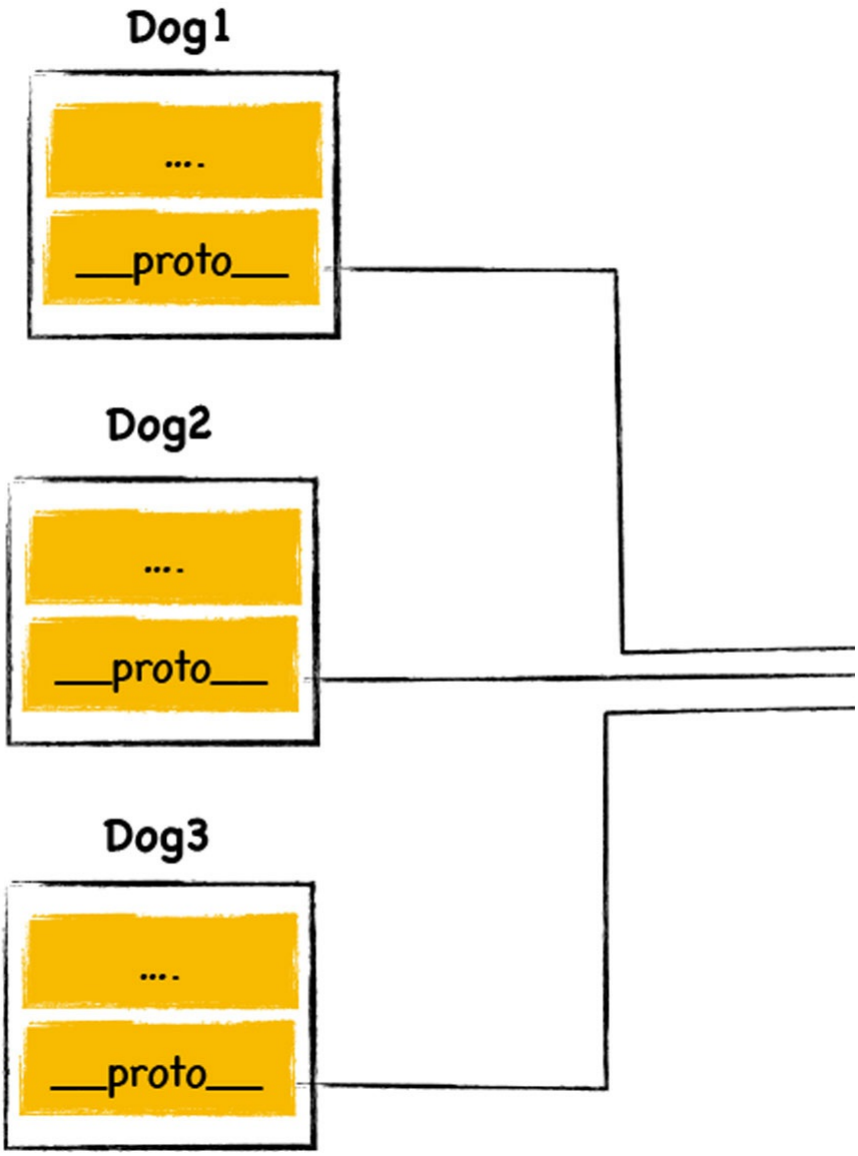
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

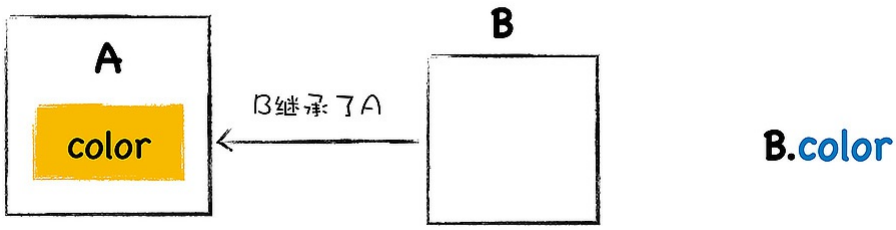
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

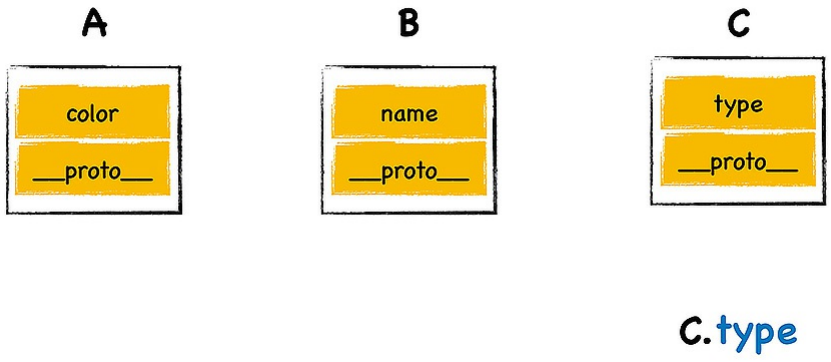
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

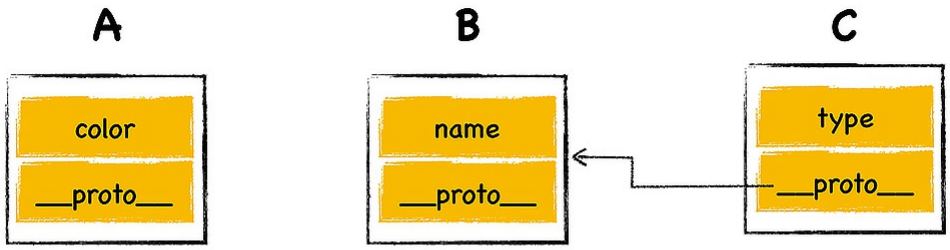


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

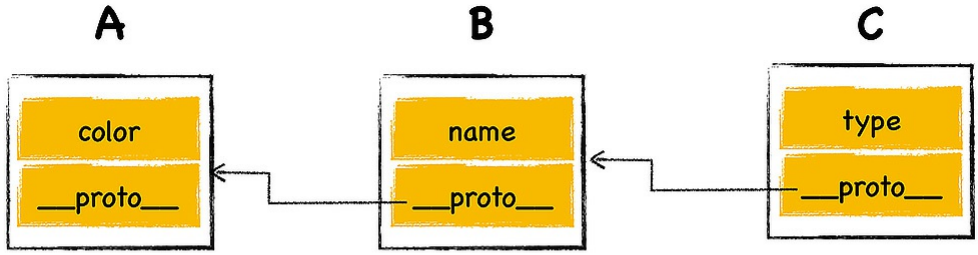


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

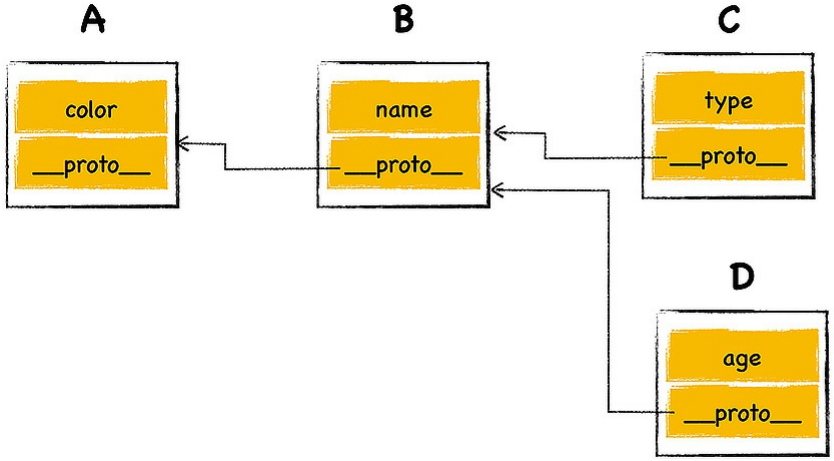
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过*_proto_*来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

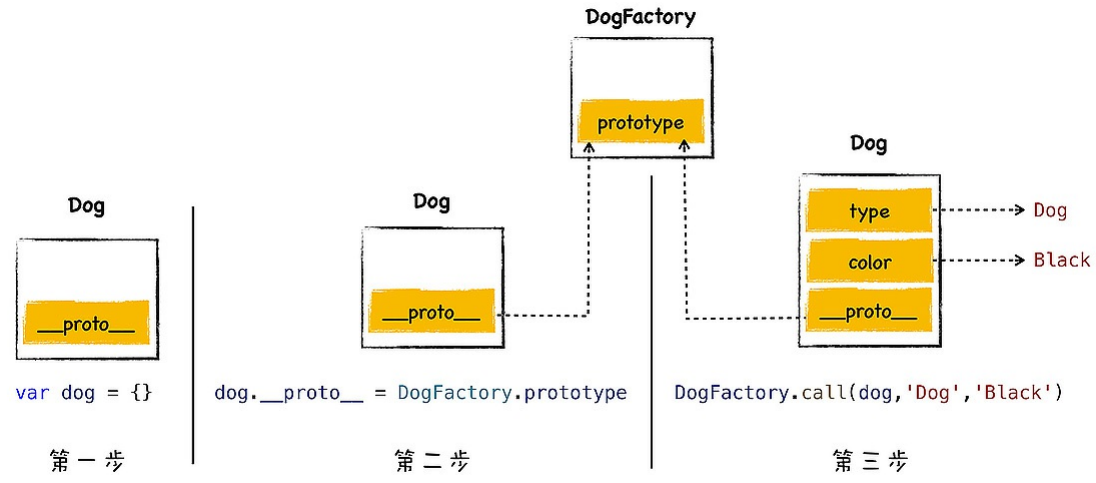
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

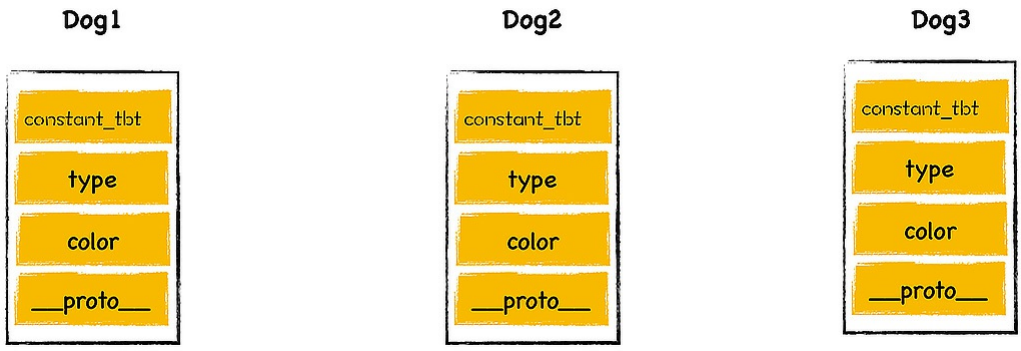
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

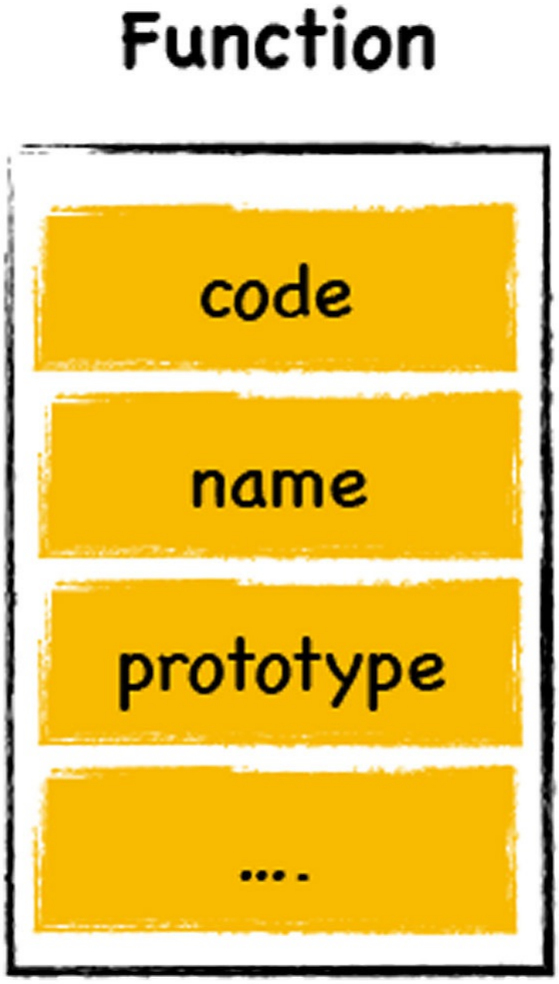
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

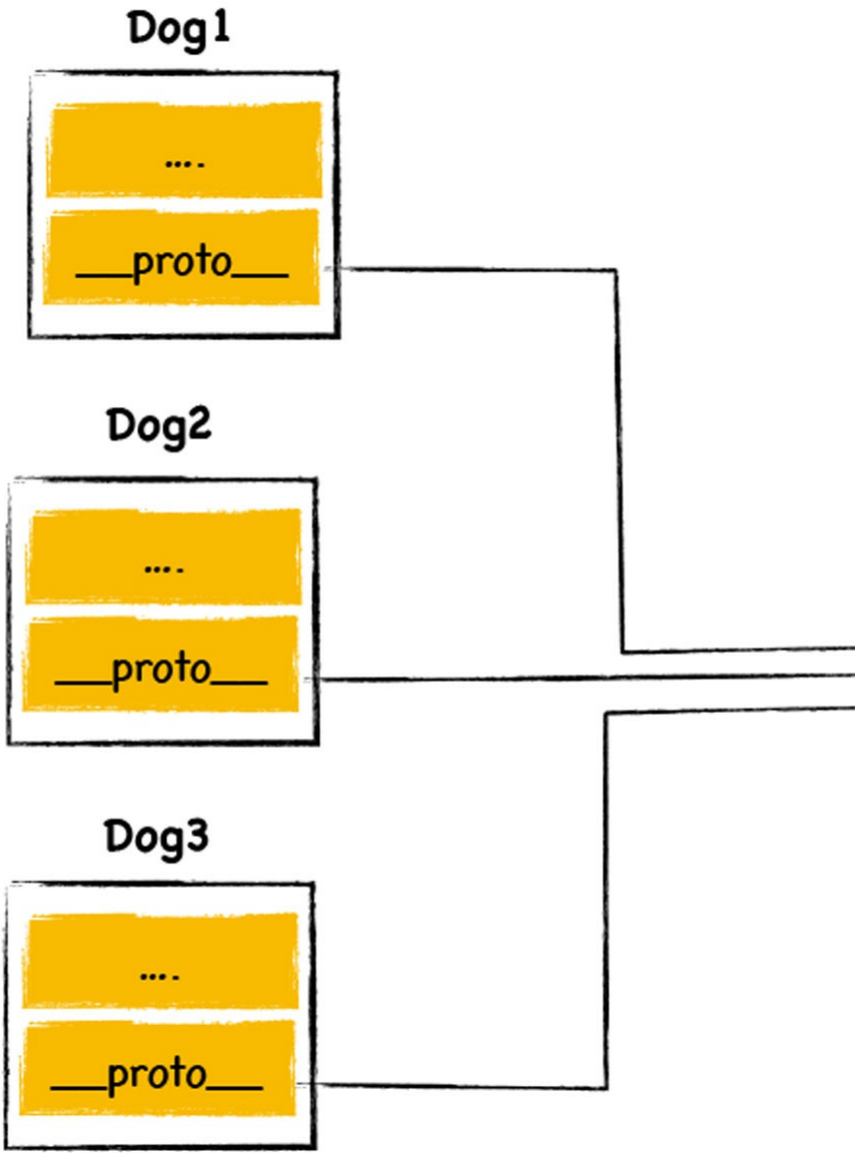
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

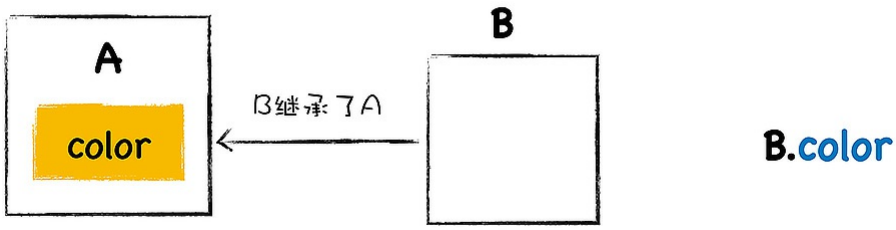
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

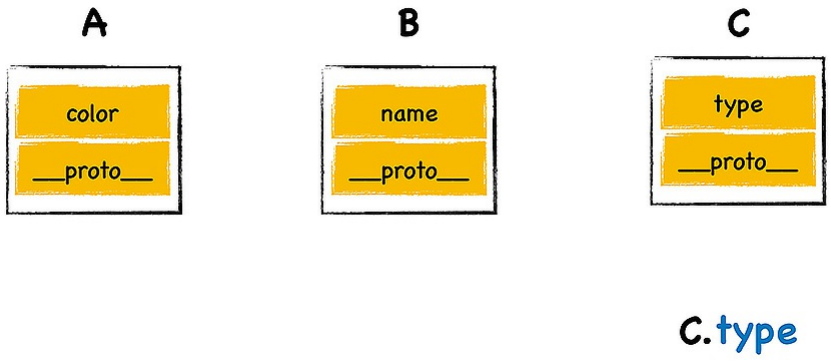
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

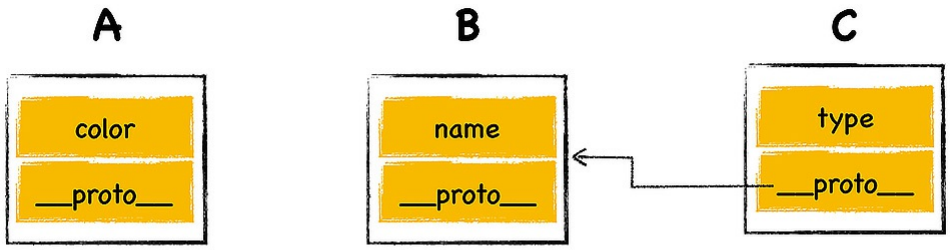


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

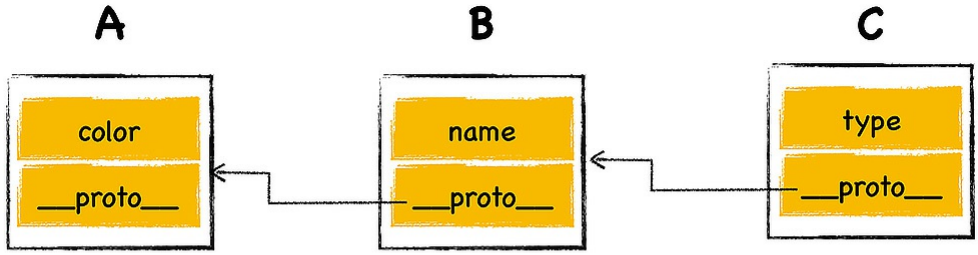


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

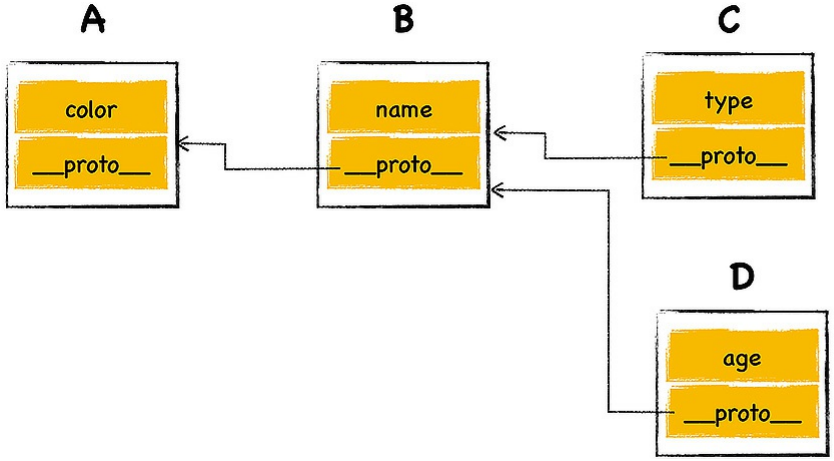
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过*_proto_*来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

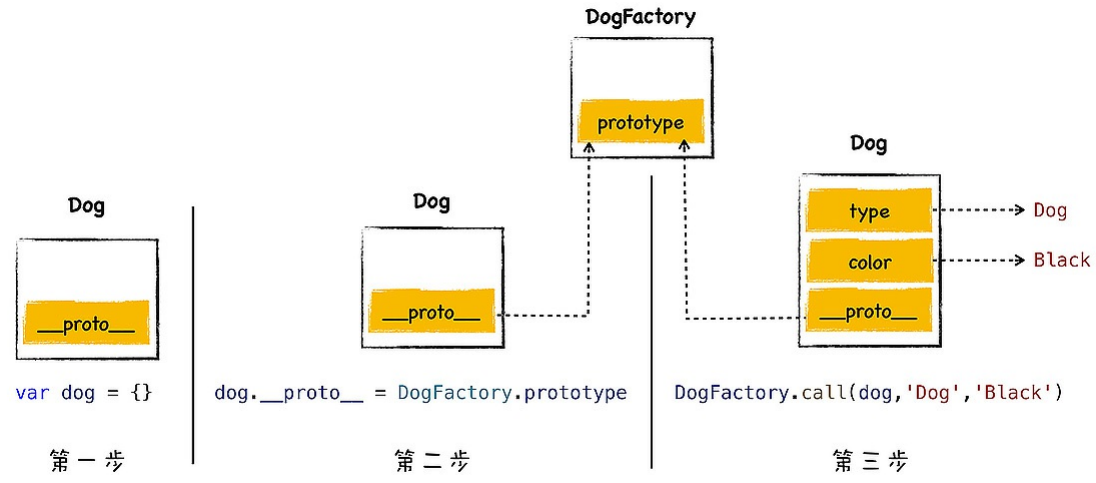
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

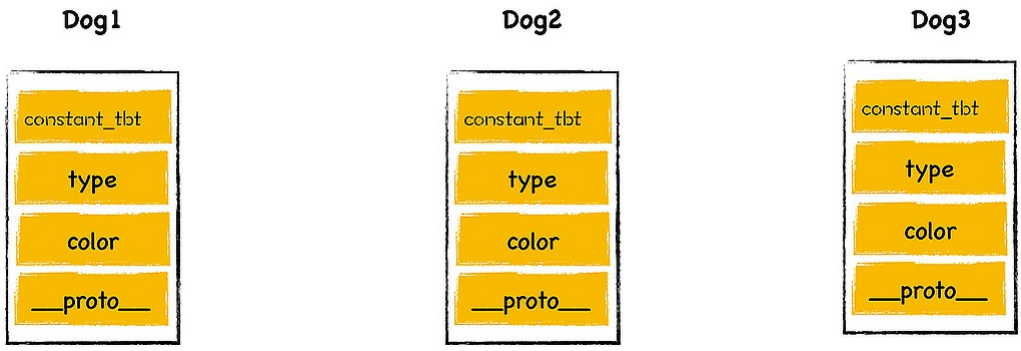
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

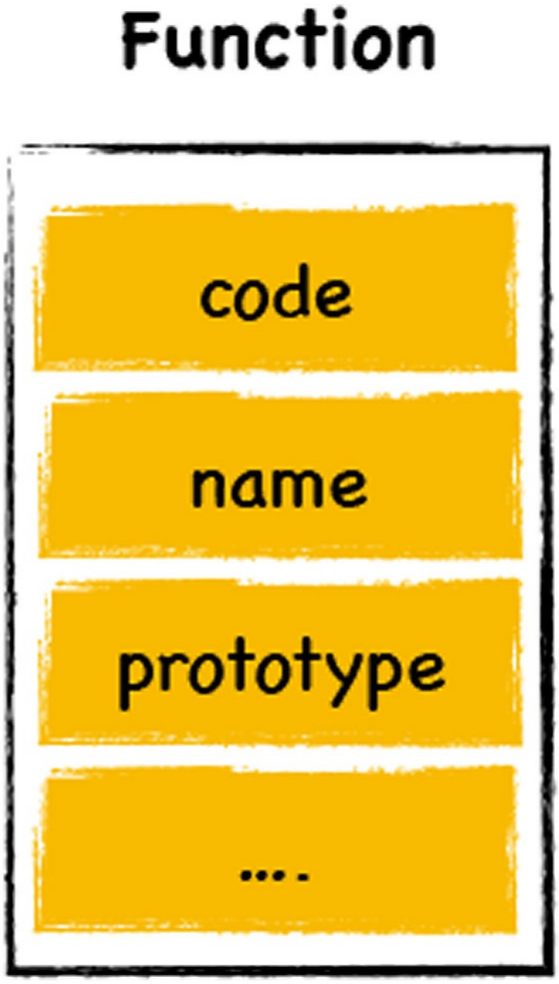
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

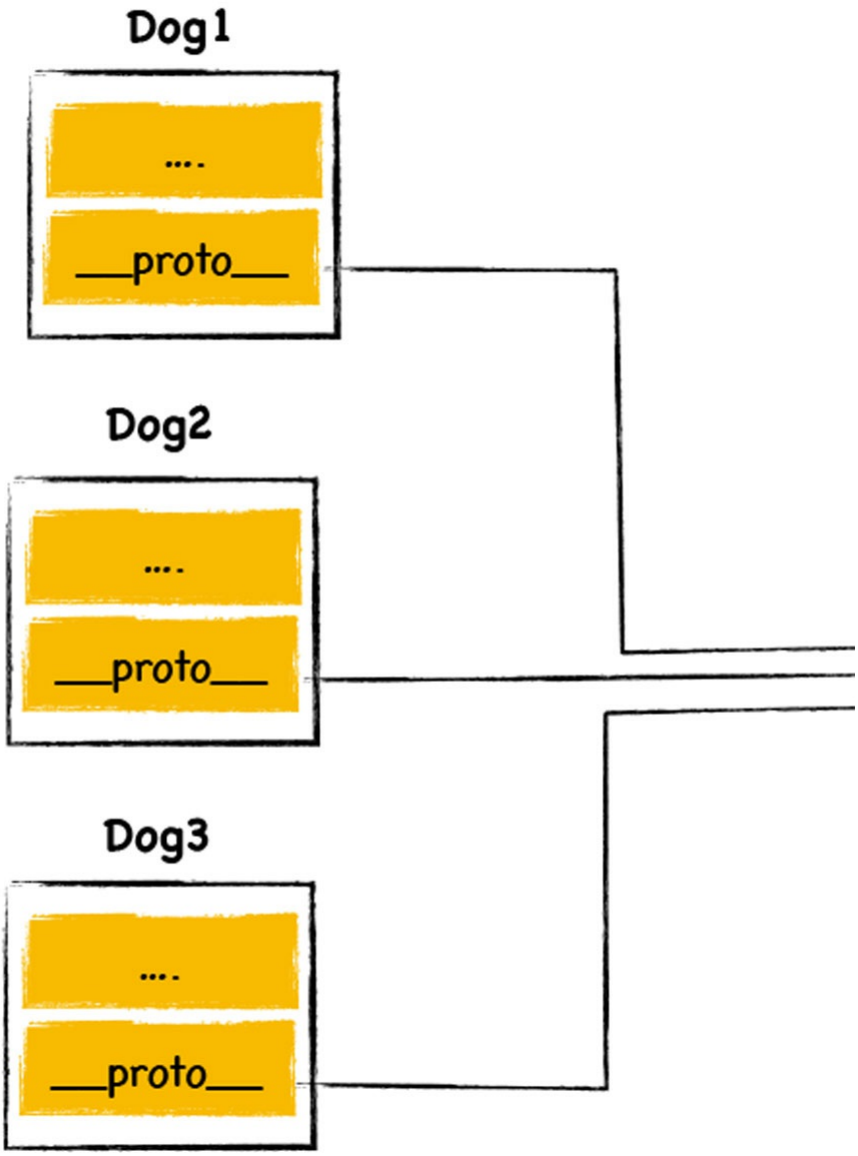
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

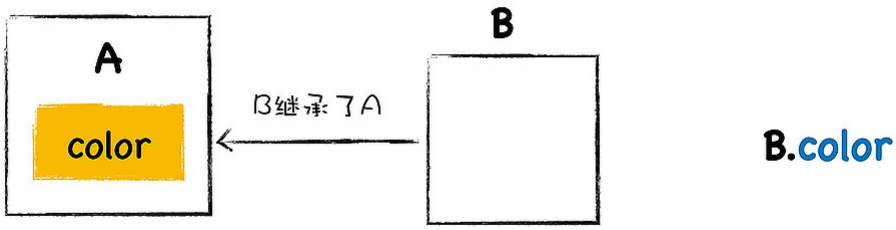
我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*__proto__*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

在前面两节中，我们分析了什么是JavaScript中的对象，以及V8内部是怎么存储对象的，本节我们继续深入学习对象，一起来聊聊V8是如何实现JavaScript中对象继承的。

简单地理解，**继承就是一个对象可以访问另外一个对象中的属性和方法**，比如我有一个B对象，该对象继承了A对象，那么B对象便可以直接访问A对象中的属性和方法，你可以参考下图：



观察上图，因为B继承了A，那么B可以直接使用A中的color属性，就像这个属性是B自带的一样。

不同的语言实现继承的方式是不同的，其中最典型的两种方式是**基于类的设计**和**基于原型继承的设计**。

C++、Java、C#这些语言都是基于经典的类继承的设计模式，这种模式最大的特点就是提供了非常复杂的规则，并提供了非常多的关键字，诸如class、friend、protected、private、interface等，通过组合使用这些关键字，就可以实现继承。

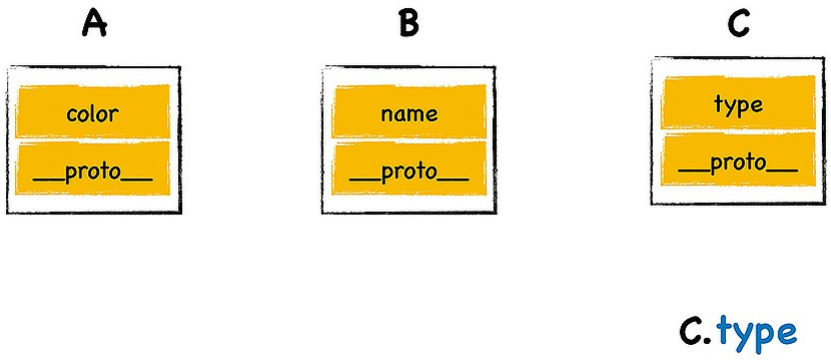
使用基于类的继承时，如果业务复杂，那么你需要创建大量的对象，然后需要维护非常复杂的继承关系，这会导致代码过度复杂和臃肿，另外引入了这么多关键字也给设计带来了更大的复杂度。

而JavaScript的继承方式和其他面向对象的继承方式有着很大差别，JavaScript本身不提供一个class 实现。虽然标准委员会在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 的继承依然和基于类的继承没有一点关系。所以当你看到JavaScript出现了class关键字时，不要以为JavaScript也是面向对象语言了。

JavaScript仅仅在对象中引入了一个原型的属性，就实现了语言的继承机制，基于原型的继承省去了很多基于类继承时的繁文缛节，简洁而优美。

原型继承是如何实现的？

那么，基于原型继承是如何实现的呢？我们参看下图：

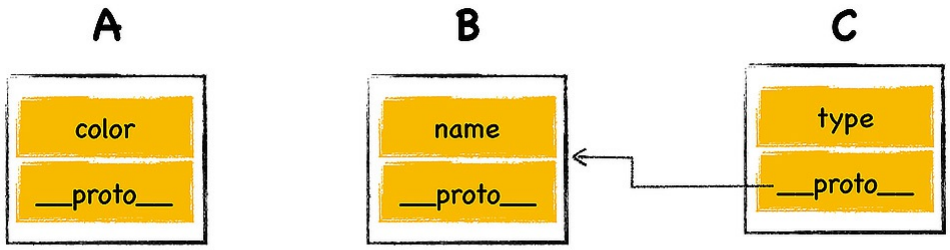


有一个对象C，它包含了一个属性“type”，那么对象C是可以直接访问它自己的属性type的，这点毫无疑问。

怎样让C对象像访问自己的属性一样，访问B对象呢？

上节我们从V8的内存快照看到，JavaScript的每个对象都包含了一个隐藏属性__proto__，我们就把该隐藏属性__proto__称之为该**对象的原型(prototype)**，__proto__指向了内存中的另外一个对象，我们就把__proto__指向的对象称为该对象的**原型对象**，那么该对象就可以直接访问其原型对象的方法或者属性。

比如我让C对象的原型指向B对象，那么便可以利用C对象来直接访问B对象中的属性或者方法了，最终的效果如下图所示：

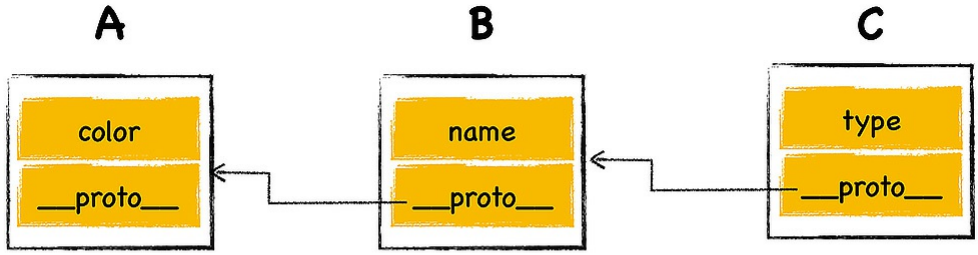


C.type

C.name

观察上图，当C对象将它的__proto__属性指向了B对象后，那么通过对象C来访问对象B中的name属性时，V8会先从对象C中查找，但是并没有查找到，接下来V8继续在其原型对象B中查找，因为对象B中包含了name属性，那么V8就直接返回对象B中的name属性值，虽然C和B是两个不同的对象，但是使用的时候，B的属性看上去就像是C的属性一样。

同样的方式，B也是一个对象，它也有自己的__proto__属性，比如它的属性指向了内存中另外一块对象A，如下图所示：



C.type

C.name

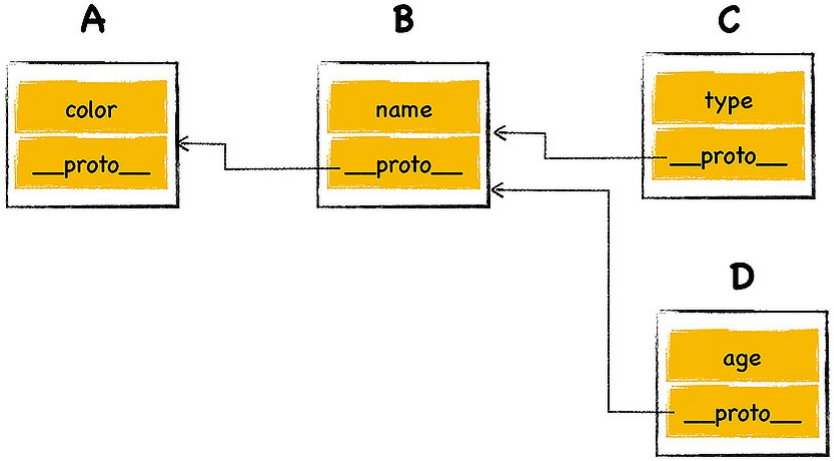
C.color

从图中可以看到，对象A有个属性是color，那么通过C.color访问color属性时，V8会先在C对象内部查找，但是没有查找到，接着继续在C对象的原型对象B中查找，但是依然没有查找到，那么继续去对象B的原型对象A中查找，因为color在对象A中，那么V8就返回该属性值。

我们看到使用C.name和C.color时，给人的感觉属性name和color都是对象C本身的属性，但实际上这些属性都是位于原型对象上，我们把这个查找属性的路径称为**原型链**，它像一个链条一样，将几个原型链接了起来。

在这里还要注意一点，不要将原型链接和作用域链搞混淆了，作用域链是沿着函数的作用域一级一级来查找变量的，而原型链是沿着对象的原型一级一级来查找属性的，虽然它们的实现方式是类似的，但是它们的用途是不同的，关于作用域链，我会在《06 | 作用域链：V8是如何查找变量的？》这节课来介绍。

关于继承，还有一种情况，如果我有另外一个对象D，它可以和C共同拥有同一个原型对象B，如下图所示：



C.type

C.name

C.color

D.age

D.name

D.color

因为对象C和对象D的原型都指向了对象B，所以它们共同拥有同一个原型对象，当我通过D去访问name属性或者color属性时，返回的值和使用对象C访问name属性和color属性是一样的，因为它们是同一个数据。

我们再来回顾下继承的概念：**继承就是一个对象可以访问另外一个对象中的属性和方法，在JavaScript中，我们通过原型和原型链的方式来实现了继承特性。**

通过上面的分析，你可以看到在JavaScript中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。所以对于JavaScript中的原型继承，你不需要把它想得过度复杂。

实践：利用__proto__实现继承

了解了JavaScript中的原型和原型链继承之后，下面我们就可以通过一个例子，看看原型是怎么应用在JavaScript中的，你可以先看下面这段代码：

```
var animal = {
    type: "Default",
    color: "Default",
    getInfo: function () {
        return `Type is: ${this.type}, color is ${this.color}.`
    }
}
var dog = {
    type: "Dog",
    color: "Black",
}
```

在这段代码中，我创建了两个对象animal和dog，我想让dog对象继承于animal对象，那么最直接的方式就是将dog的原型指向对象animal，应该怎么操作呢？

我们可以通过设置dog对象中的__proto__属性，将其指向animal，代码是这样的：

```
dog.__proto__ = animal
```

设置之后，我们就可以使用dog来调用animal中的getInfo方法了。

```
dog.getInfo()
```

你可以尝试调用下，看看输出的内容。在这里留给你一个关于“this”的小思考题：调用dog.getInfo()时，getInfo函数中的this.type和this.color都是什么值？为什么？

还有一点我们要注意，通常隐藏属性是不能使用JavaScript来直接与之交互的。虽然现代浏览器都开了一个口子，让JavaScript可以访问隐藏属性 *_proto_*，但是在实际项目中，我们不应该直接通过*_proto_*来访问或者修改该属性，其主要原因有两个：

- 首先，这是隐藏属性，并不是标准定义的；
- 其次，使用该属性会造成严重的性能问题。

我们之所以在课程中使用 *_proto_* 属性，主要是为了方便教学，将其他的一些复杂的概念先抛到一边，这样有利于你循序渐进地掌握我们的课程内容，但是我并不推荐你这么做。那应该怎么去正确地设置对象的原型对象呢？

答案是使用构造函数来创建对象，下面我们就来详细解释这个过程。

构造函数是怎么创建对象的？

比如我们要创建一个dog对象，我可以先创建一个DogFactory的函数，属性通过参数进行传递，在函数体内，通过this设置属性值。代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
}
```

然后再结合关键字“new”就可以创建对象了，创建对象的代码如下所示：

```
var dog = new DogFactory('Dog','Black')
```

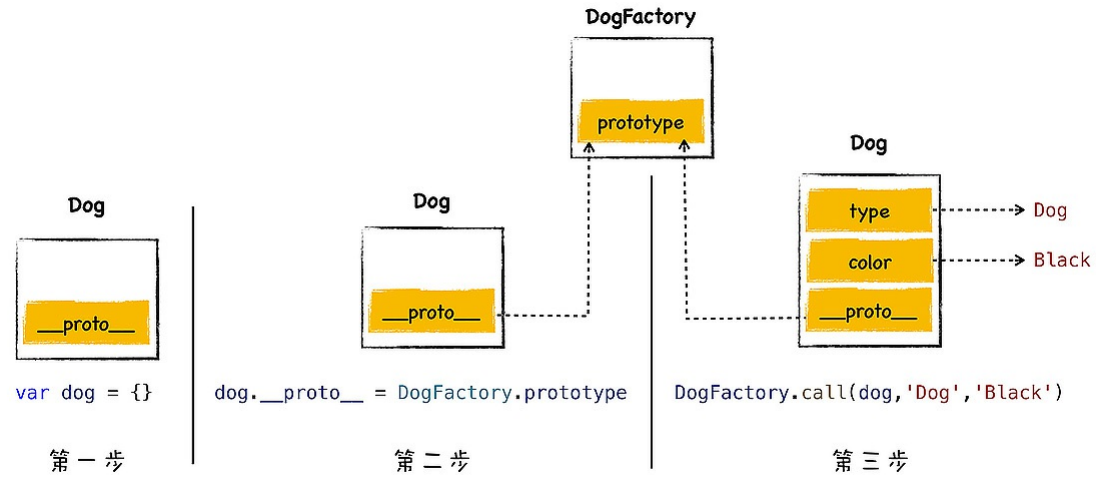
通过这种方式，我们就把后面的函数称为构造函数，因为通过执行new配合一个函数，JavaScript虚拟机便会返回一个对象。如果你没有详细研究过这个问题，很可能对这种操作感到迷惑，为什么通过new关键字配合一个函数，就会返回一个对象呢？

关于JavaScript为什么要采用这种怪异的写法，我们文章最后再来介绍，先来看看这段代码的深层含义。

其实当V8执行上面这段代码时，V8会在背后悄悄地做了以下几件事情，模拟代码如下所示：

```
var dog = {}
dog.__proto__ = DogFactory.prototype
DogFactory.call(dog,'Dog','Black')
```

为了加深你的理解，我画了上面这段代码的执行流程图：



观察上图，我们可以看到执行流程分为三步：

- 首先，创建了一个空白对象dog；
- 然后，将DogFactory的prototype属性设置为dog的原型对象，这就是给dog对象设置原型对象的关键一步，我们后面来介绍；
- 最后，再使用dog来调用DogFactory，这时候DogFactory函数中的this就指向了对象dog，然后在DogFactory函数中，利用this对对象dog执行属性填充操作，最终就创建了对象dog。

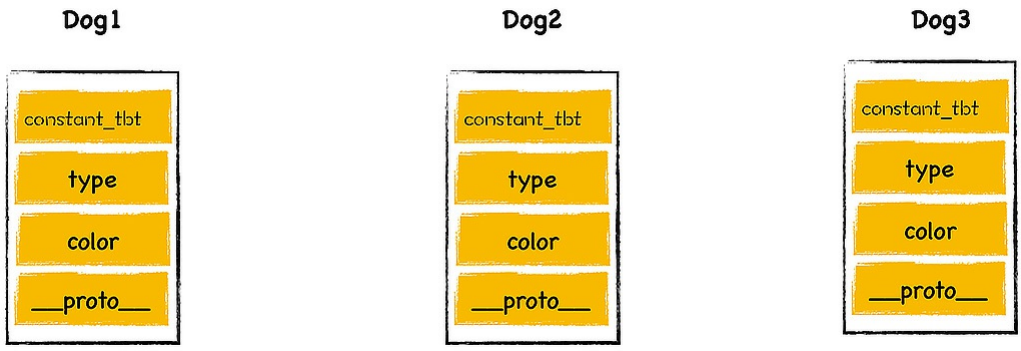
构造函数怎么实现继承？

好了，现在我们可以通过构造函数来创建对象了，接下来我们就看看构造函数是如何实现继承的？你可以先看下面这段代码：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
```

```
    //恒温
    this.constant_temperature = 1
}
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

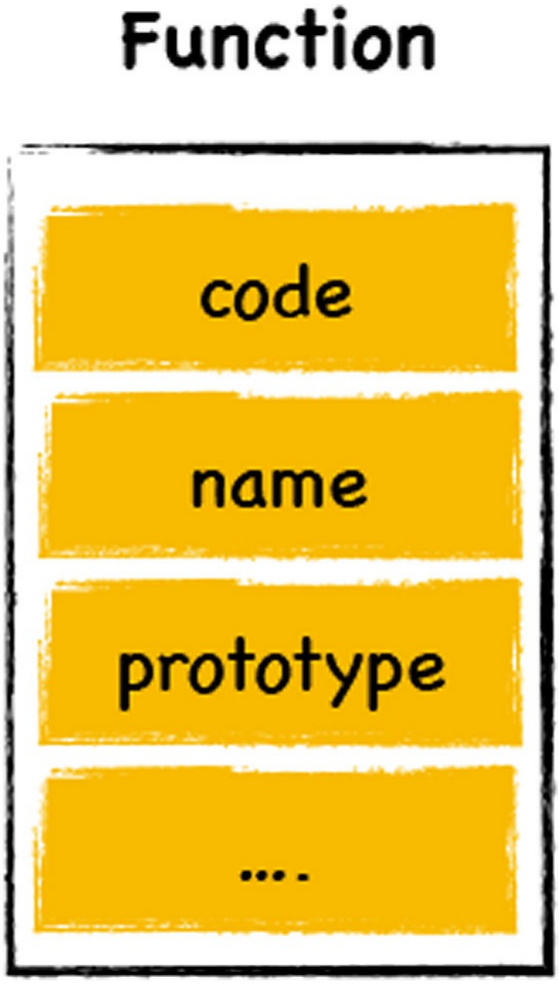
我利用上面这段代码创建了三个dog对象，每个对象都占用了一块空间，占用空间示意图如下所示：



从图中可以看出来，对象dog1到dog3中的constant_temperature属性都占用了一块空间，但是这是一个通用的属性，表示所有的dog对象都是恒温动物，所以没有必要在每个对象中都为该属性分配一块空间，我们可以将该属性设置公用的。

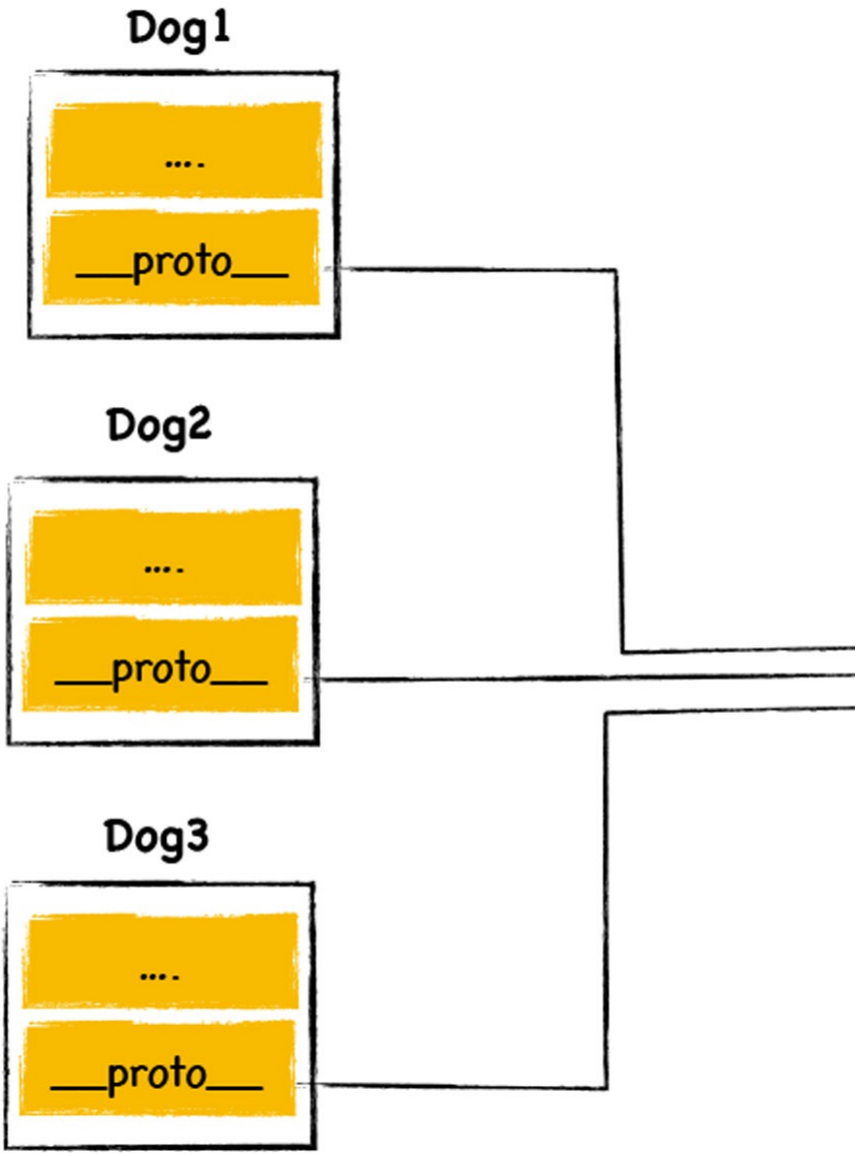
怎么设置呢？

还记得我们介绍函数时提到关于函数有两个隐藏属性吗？这两个隐藏属性就是name和code，其实函数还有另外一个隐藏属性，那就是prototype，刚才介绍构造函数时我们也提到过。一个函数有以下几个隐藏属性：



每个函数对象中都有一个公开的prototype属性，当你将这个函数作为构造函数来创建一个新的对象时，新创建对象的原型对象就指向了该函数的prototype属性。当然了，如果你只是正常调用该函数，那么prototype属性将不起作用。

现在我们知道了新对象的原型对象指向了构造函数的prototype属性，当你通过一个构造函数创建多个对象的时候，这几个对象的原型都指向了该函数的prototype属性，如下图所示：



这时候我们可以将constant_temperature属性添加到DogFactory的prototype属性上，代码如下所示：

```
function DogFactory(type,color){
    this.type = type
    this.color = color
    //Mammalia
}
DogFactory. prototype.constant_temperature = 1
var dog1 = new DogFactory('Dog','Black')
var dog2 = new DogFactory('Dog','Black')
var dog3 = new DogFactory('Dog','Black')
```

这样我们三个dog对象的原型对象都指向了prototype，而prototype又包含了constant_temperature属性，这就是我们实现继承的正确方式。

一段关于new的历史

现在我们知道new关键字结合构造函数，就能生成一个对象，不过这种方式很怪异，为什么要这样呢？要了解这背后的原因，我们需要了解一段关于关于JavaScript的历史。

JavaScript是Brendan Eich发明的，那是个“战乱”的时代，各种大公司相互争霸，有Sun、微软、网景、甲骨文等公司，它们都有推出自己的语言，其中最炙手可热的编程语言是Sun的Java，而JavaScript就是这个时候诞生的。当时创造JavaScript的目的仅仅是为了让浏览器页面可以动起来，所以尽可能采用简化的方式来设计JavaScript，所以本质上来说，Java和JavaScript的关系就像雷锋和雷峰塔的关系。

那么之所以叫JavaScript是出于市场原因考量的，因为一门新的语言需要吸引新的开发者，而当时最大的开发者群体就是Java，于是JavaScript就蹭了Java的热度，事后，这一招被证明的确有效果。

虽然叫JavaScript，但是其编程方式和Java比起来，依然存在着非常大的差异，其中Java中使用最频繁的代码就是创建一个对象，如下所示：

```
CreateInstance instance = new CreateInstance();
```

当时JavaScript并没有使用这种方式来创建对象，因为JavaScript中的对象和Java中的对象是完全不一样的，因此，完全没有必要使用关键字new来创建一个新对象的，但是为了进一步吸引Java程序员，依然需要在语法层面去蹭Java热点，所以JavaScript中就被硬生生地强制加入了非常不协调的关键字new，然后使用new来创造对象就变成这样了：

```
var bar = new Foo()
```

Java程序员看到这段代码时，当然会感到倍感亲切，觉得Java和JavaScript非常相似，那么使用JavaScript也就天经地义了。不过代码形式只是表象，其背后原理是完全不同的。

了解了这段历史之后，我们知道JavaScript的new关键字设计并不合理，但是站在市场角度来说，它的出现又是非常成功的，成功地推广了JavaScript。

总结

好了，今天的主要内容就介绍到这里，下面我们来回顾下。

今天我们的主要目的是介绍清楚JavaScript中的继承机制，这涉及到了原型继承机制，虽然基于原型的继承机制本身比较简单，但是在JavaScript中，这是通过关键字new加上构造函数来体现的。这种方式非常绕，且不符合人的直觉，如果直接上来就介绍new加构造函数是怎么工作的，可能会把你给绕晕了。

于是我先通过每个对象中都有的隐含属性__proto__，来介绍了什么是原型和原型链。V8为每个对象都设置了一个__proto__属性，该属性直接指向了该对象的原型对象，原型对象也有自己的__proto__属性，这些属性串连在一起就成了原型链。

不过在JavaScript中，并不建议直接使用__proto__属性，主要有两个原因。

- 一，这是隐藏属性，并不是标准定义的；
- 二，使用该属性会造成严重的性能问题。

所以，在JavaScript中，是使用new加上构造函数的这种组合来创建对象和实现对象的继承。不过使用这种方式隐含的语义过于隐晦，所以理解起来有点难度。

为什么JavaScript中要使用这种怪异的方式来创建对象？为了理解这个问题，我们回顾了一段JavaScript的历史。由于当前的Java非常流行，基于市场推广的考虑，JavaScript采取了蹭Java热度的策略，在语言命名上使用了Java字样，在语法形式上也模仿了Java。事实上通过这些策略，确实为JavaScript带来了市场上的成功。不过你依然要记住，JavaScript和Java是完全两种不同的语言。

思考题

我们知道函数也是一个对象，所以函数也有自己的__proto__属性，那么今天留给你的思考题是：DogFactory是一个函数，那么“DogFactory.prototype”和“DogFactory.*_proto_*”这两个属性之间有关联吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。