11 原型与面向对象真题解析

更新时间：2020-07-06 17:48:48

> 既然我已经踏上这条道路，那么，任何东西都不应妨碍我沿着这条路走下去。——康德

命题思路整体把握

原型侧知识偶见单独命题，但更多的是和其它 JS 核心知识结合起来命题。

这样做，可以从整体上拔高题目的区分度、同时 “一箭多雕” 实现对候选人基本功的全面考察。

处理这样的题目，大家首先要保持冷静的头脑，甄别出题目中所涉及的知识点、对其分门别类、快速在脑海中做映射；作答过程中，主要**抓手是梳理出一条清晰正确的原型链** —— 大部分的题目乍一看非常复杂，但如果你因为复杂想撤退，就恰恰中了命题人的计了！很多时候，只要你能静下心来把原型链抓出来，你就会发现自己的畏难情绪少了一大半，整个作答的脉络也随之清晰起来。

命题点一：原型基础 + 构造函数基础

```
var A = function() {};
A.prototype.n = 1;
var b = new A();
A.prototype = {
  n: 2,
  m: 3
}
var c = new A();

console.log(b.n);
console.log(b.m);

console.log(c.n);
console.log(c.m);
```

易错点拨：

这里我知道非常多的同学可能会给出这个答案：

```
2
3
2
3
```

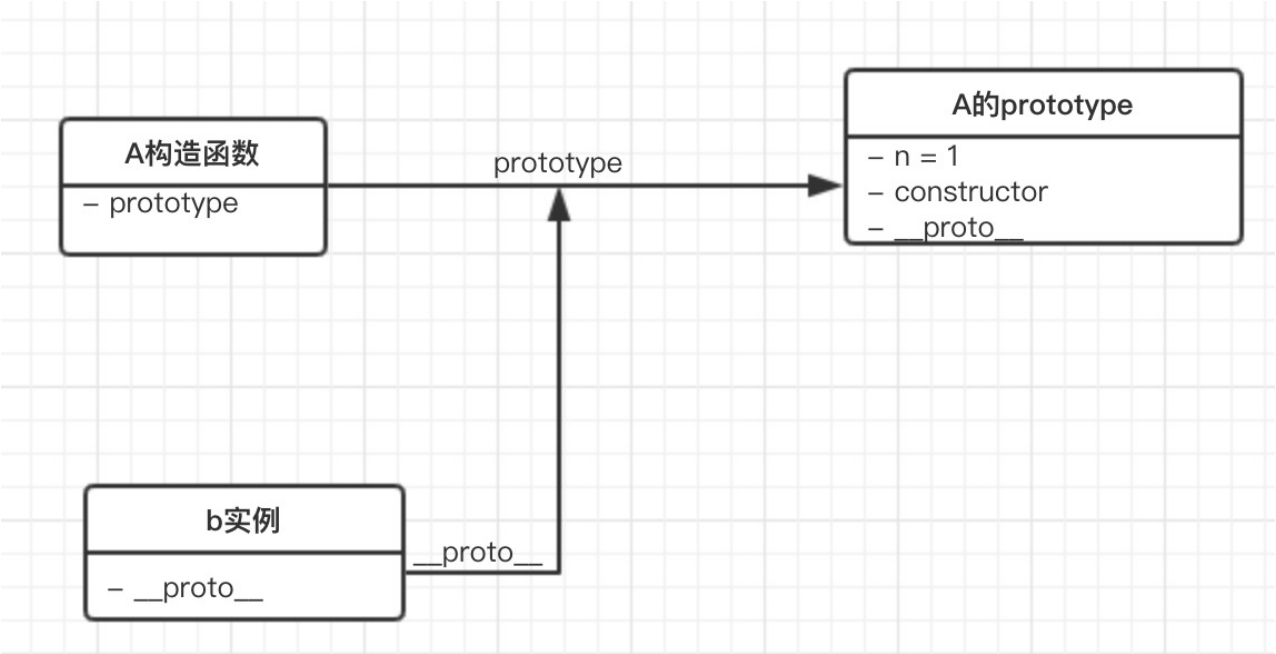
但如果你把它丢进控制台，你会发现答案是：

```
1                    VM5291:10
undefined            VM5291:11
2                    VM5291:13
3                    VM5291:14
```

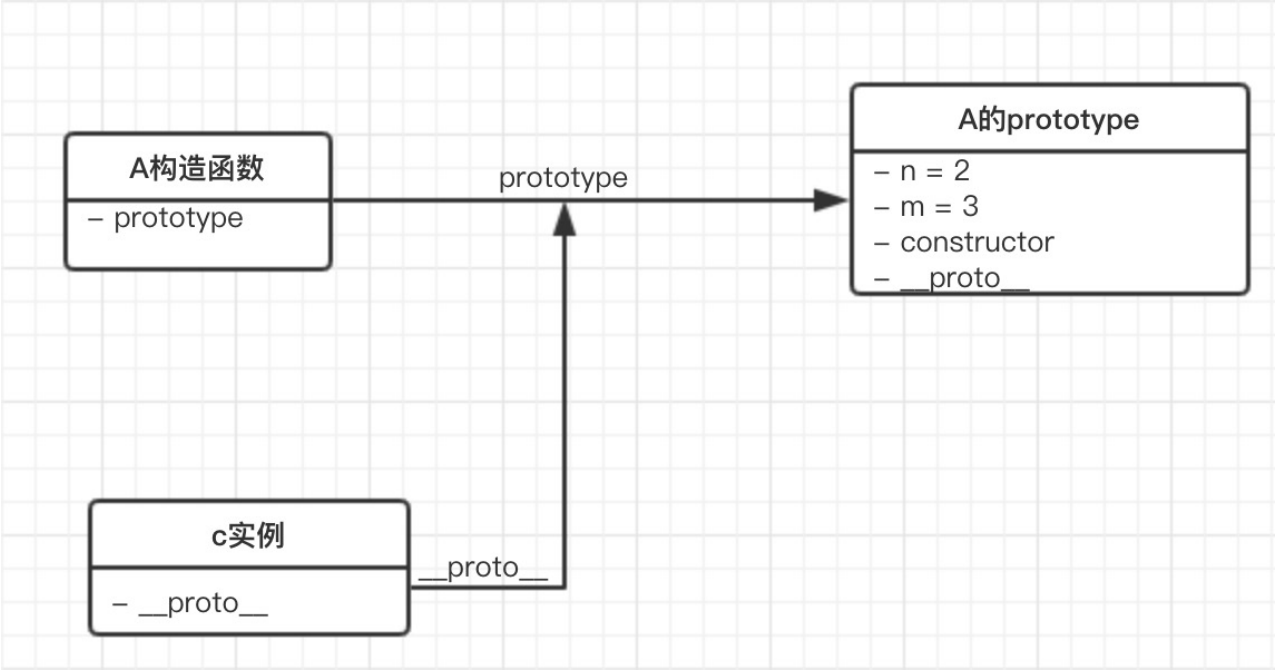
为什么会这样呢？我们一起来看一下这个例子中几个对象间的关系：

Step1：明确原型关系：

b 实例与 A 之间的关系：



c 实例与 A 之间的关系：



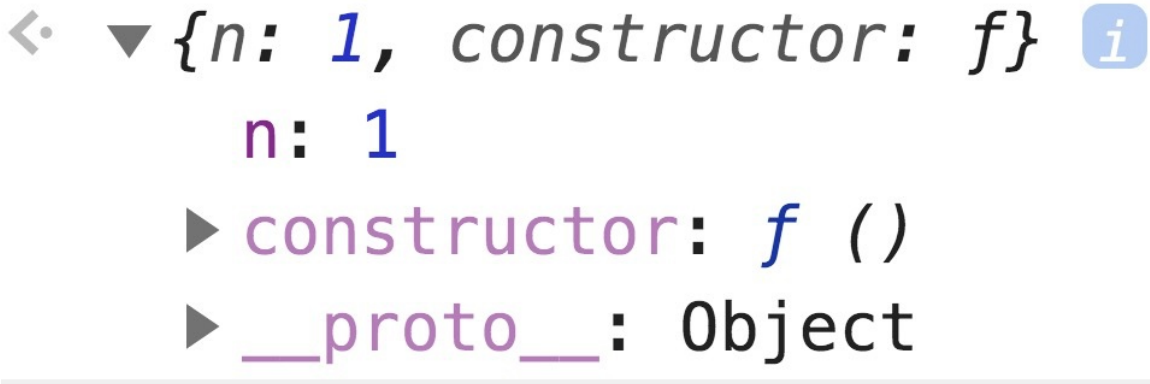
Step2：关键思路解析 - 构造函数的工作机理

相信很多同学对 c 实例没有疑问，更多是疑惑 b 实例为什么明明和 c 实例继承自一个原型，却有着不同的表现。

这里需要大家注意一个知识点：当我们用 new 去创建一个实例时，new 做了什么？它做了这四件事：

- 为这个新的对象开辟一块属于它的内存空间
- **把函数体内的 this 指到 1 中开辟的内存空间去**
- **将新对象的 _ proto_ 这个属性指向对应构造函数的 prototype 属性，把实例和原型对象关联起来**
- 执行函数体内的逻辑，最后即便你没有手动 return，构造函数也会帮你把创建的这个新对象 return 出来

注意第二步哦，第二步执行完之后，实例对象的原型就把构造函数的 prototype 的引用给存下来了。那么在 b 实例创建的时候，构造函数的 prototype 是啥呢？就是这么个对象：

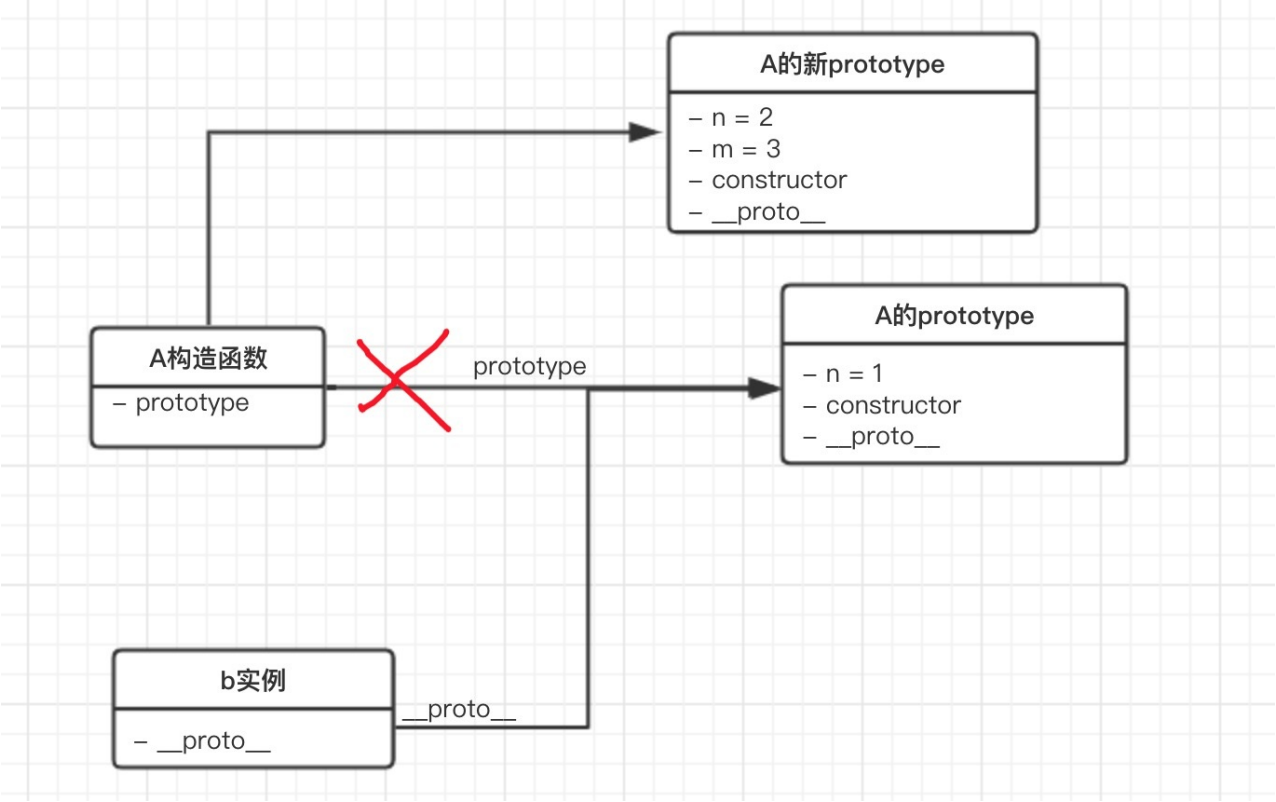


所以 b 实例输出的 n 就是 1；同时由于它没有 m 属性，直接输出 undefined。

有同学会说了，可是后面我们还对 A 的 prototype 做了修改啊！b 如果存的是引用，它应该感知到我这个修改啊！注意你修改 A 的 prototype 的形式：

```
A.prototype = {
  n: 2,
  m: 3
}
```

这严格意义上来说不算修改，而是一个重新赋值操作。这个动作的本质是把 A 的 prototype 指向了一个全新的 js 对象：



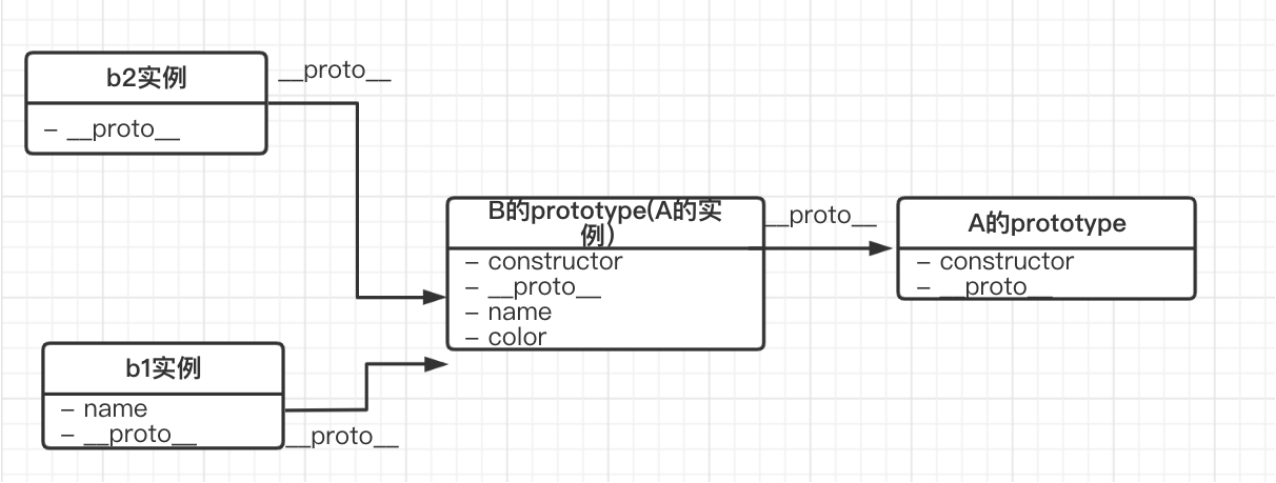
从图中我们可以看出，A 单方面切断了和旧 prototype 的关系，而 b 却仍然保留着旧 prototype 的引用。这就是造成 b 实例和 c 实例之间表现差异的原因。

命题点二：自有属性与原型继承属性

```
function A() {
  this.name = 'a'
  this.color = ['green', 'yellow']
}
function B() {

}
B.prototype = new A()
var b1 = new B()
var b2 = new B()

b1.name = 'change'
b1.color.push('black')

console.log(b2.name) // 'a'
console.log(b2.color) // ["green", "yellow", "black"]
```

Step1：画出原型链图

这道题有一个迷惑你的地方，就是没有直接用 A 去 new 对象，而是找了一个“中间人” B，这样就达到了把原型链复杂化的目的。不过问题不大，咱们图照画：



Step2：读操作与写操作的区别

这里呢，大家看到 b1 和 b2 之间的一个区别就是 b1 有自有的 name 属性。有的同学可能会迷惑，这一行代码：

```
b1.name = 'change'
```

在查找 b1 的 name 属性时，难道不应该沿着原型链去找，然后定位并修改原型链上的 name 吗？

实际上，这个“逆流而上”的变量定位过程，当且仅当我们在进行“读”操作时会发生。

楼上这行代码，是一个赋值动作，是一个“写”操作。在写属性的过程中，如果发现 name 这个属性在 b1 上还没有，那么就会原地为 b1 创建这个新属性，而不会去打扰原型链了。

那么 color 这个属性，看上去也像是一个“写”操作，为什么它没有给 b2 新增属性、而是去修改了原型链上的 color 呢？首先，这样的写法：

```
b1.color.push('black')
```

包括这样的写法（修改对象的属性值）：

```
b1.color.attribute = 'xxx'
```

它实际上并没有改变对象的引用，而仅仅是在原有对象的基础上修改了它的内容而已。像这种不触发引用指向改变的操作，它走的就是 **原型链 查询 + 修改** 的流程，而非原地创建新属性的流程。

如何把它变成写操作呢？直接赋值：

```
b1.color = ['newColor']
```

这样一来，color 就会变成 b1 的一个自有属性了。 因为 ['newColor'] 是一个全新的数组，它对应着一个全新的引用。对 js 来说，这才是真正地在向 b1 “写入” 一个新的属性。

易错点拨

有同学读到这里可能会小看这样的考法：不就是注意一下引用类型么？太 normal 了吧？

实际上，上面这道题在实际考察中区分度非常高。很多的候选人可能就和此时此刻的你一样，觉得 “这题看上去基础”，于是张口就来。这里我要提醒大家，看似再 normal 的题，也需要你调度自己的综合能力去 fix 它。这个 “综合能力” 不仅仅是说你要把自己知道的知识点综合起来，还包括你做题的细心程度和谨慎程度 —— 切记，面试无难题，难在人心。

命题点三：构造函数综合考察

```
function A() {}
function B(a) {
    this.a = a;
}
function C(a) {
    if (a) {
        this.a = a;
    }
}
A.prototype.a = 1;
B.prototype.a = 1;
C.prototype.a = 1;

console.log(new A().a); // 1
console.log(new B().a); // undefined
console.log(new C(2).a); // 2
```

Step1 画出原型链图

这道题所涉及的原型关系比较简单，整体虽具有综合性、却不是一道难题，非常适合用来练手，验证各位对命题点一的理解是否到位。这里建议大家动手做，模仿我前面给出的两张原型关系、画一张自己的原型图。

Step2 构造函数的工作机理

结合我们前面对构造函数的分析，当我们像这样通过 new + 构造函数创建新对象的时候：

```
function C(a) {
    if (a) {
        this.a = a;
    }
}

var c = new C(2)
```

实际上发生了四件事情：

1. 为 c 实例开辟一块属于它的内存空间
2. **把函数体内的 this 指到 1 中开辟的内存空间去**
3. **将实例 c 的 _ proto_ 这个属性指向构造函数 C 的 prototype 属性**
4. 执行函数体内的逻辑，最后构造函数会帮你把创建的这个 c 实例 return 出来

我们基于这个结论来看 console 中的三次调用：

- **new A ().a**：构造函数逻辑为空，返回的实例对象 _ *proto*_ 中包含了 a = 1 这个属性。new A ().a 时，发现实例对象本身没有 a，于是沿着原型链找到了原型中的 a，输出其值为 1。
- **new B ().a**：构造函数中会无条件为实例对象创建一个自有属性 a，这个 a 的值以入参为准。这里我们的入参是 undefined，所以 a 值也是 undefined。
- **new C (2).a**：构造函数中会有条件地为实例对象创建一个自有属性 a—— 若确实存在一个布尔判定不为 false 的入参 a，那么为实例对象创建对应的 a 值；否则，不做任何事情。这里我们传入了 2，因此实例输出的 a 值就是 2。

小结

原型面试题乍一看挺唬人，实际拆开来看非常简单。真实面试中，刁钻的原型题目比较少见，更多还是考察你对最最基本的那些原理的理解，因此不建议大家花费大量的时间去钻难题怪题。

这里尤其要注意的是对构造函数的理解和运用。做题时，做对的关键在于你能否明确出题中所涉及的原型关系。在明确原型关系的过程中，挑战的是你的耐心和细心。做原型题目不能求快，对已经确认的原型关系，应反复梳理、二次甚至三次确认后再报答案。

}

10 原型编程范式与面向对象

12 异步编程模型与异步解决方案