你真的了解 JavaScript 继承么

什么是继承

大多数人使用继承不外乎是为了获得这两点好处，***代码的抽象***和***代码的复用***。

代码的抽象就不用说了，交通工具和汽车这类的例子数不胜数，在传统的 OO 语言中(比如 Java)，代码的抽象更多的是使用接口(interface)来实现，而使用继承更多地是为了代码的复用(虽然现在强调使用组合而不是使用继承)。

怎么复用的？打个比方，class A 继承了 class B，class A 便拥有了 class B 的 public 和 protected 类型的变量和方法，用最简单的方法去想，便是 class B 将 这些属性和方法直接 copy 给 class A，这样便实现了继承。

因此我们可以这样说，继承实际上是一种**类与类**之间的 copy 行为。

JavaScript 中的继承

在 JavaScript 中没有类的概念，只有对象。虽然现在人们经常使用 class 关键字，这让 JavaScript 看起来似乎是拥有了”类”，可表面看到的不一定是本质，class 只是一块糖，嚼碎了才知道里面其实还是原型链那一套。因此，JavaScript 中的继承只是对象与对象之间的继承。反观继承的本质，继承便是让子类拥有父类的一些属性和方法，那么在 JavaScript 中便是让一个对象拥有另一个对象的属性和方法。

所以，这给我了我们一条十分清晰的思路，JavaScript 中如何实现继承？**只需让一个对象拥有另一个对象的属性和方法**，这就实现了。

利用 Mixin(mix in 的缩写，混合)

既然让一个对象拥有另一个对象的属性和方法，首先想到的便是利用 Mixin 的粗暴方式，直接将对象的属性和方法强制 copy 到另一个对象。

就像这样

```
function mixin(subObj, parentObj) {
    for (var prop in parentObj) {
        if (!(prop in subObj)) {
            subObj[prop] = parentObj[prop]
        }
    }
}
```

当然也可以用 ES6 中的更优雅的 Object.assign。

这段代码就实现了最简单的从一个对象复制属性和方法到另一个对象。然而这种方法有一个缺陷，如果父对象的属性是引用类型，比如一个对象或者数组，那么修改子对象的时候势必会对父对象也造成修改，这显然不可接受。一种想法是采用深度克隆，然而又可能会有循环引用的问题。

所以，这种继承方式，比较适合对简单对象的拓展，不太适合更复杂的继承。

利用原型链

首先来说一下什么是**原型**，原型在 JavaScript 中，其实就是某个对象的一个属性。只不过这个属性很特殊，对于外界一般是不可见(在 chrome 中可以通过__proto__获取),我们一般把它叫作[[Prototype]]。这里和函数的 prototype 属性很相似但却是两个东西，后面会提到。

那么什么是原型链呢，顾名思义就像这样：

obj1.[[Prototype]] ===> obj2.[[Prototype]] ===> obj3.[[Prototype]]…. ===> Object.prototype

某一对象的原型属性中保存着另一个对象，以此类推，好像链子一样串起来。
链的终点是 Object.prototype 对象，因此 Object.prototype 没有原型。当我们构建一个对象，这个对象的默认的原型就是 Object.prototype
在 chrome 中验证一下：

```
var a = {}
Object.prototype === a.__proto__    // true
```

那么我们如何用原型链实现继承呢？这要归功于 JavaScript 中的**委托**机制。
当我们获取一个对象的某个属性时，比如 a.b，会默认调用一个内置的[[Get]]方法，这个[[Get]]方法的算法就是：
在当前对象里查找，找不到则委托给当前对象的[[Prototype]],再找不到则委托给[[Prototype]]的[[Prototype]]，直到 Object.prototype 中也没找到，则返回 undefined。
因此，我们想让对象 a 拥有对象 b 的属性和方法，即对象 a 继承对象 b，只需要把 b 赋值给 a 的[[Prototype]]，利用属性查找的委托机制，实现了 a 也”拥有”了 b 的属性和方法，而且当 a 中有和 b 中的同名属性时，由于”屏蔽作用”，只有 a 中的属性会被优先获取到，实现了 override，看起来相当完美。

new 和 “构造函数”

前面提到，[[Prototype]]是个内置隐藏属性，虽然在 chrome 可以通过__proto__访问，但是其设计本意是不可被读取和修改的，那么我们如何利用原型链来建立继承关系？
JavaScript 提供了 new 关键字。
通常，在类似 Java 这样的 OO 语言中，new 被用来实例化一个类，然而在 JavaScript 中，new 仅仅是一个函数调用的方式！
JavaScript 中的函数也很奇怪，每一个函数都有一个默认的 prototype 属性，这个不同于对象的[[Prototype]]属性，函数的 prototype 是故意暴露出来的，而且这个属性还不为空，还有 prototype 还有另一个属性叫 constructor，这个 constructor 竟然又引用回来了这个函数本身！
于是我们看到的效果是这样的：

用 new 来调用函数有什么不同的呢？new 其实做了三件事：

1. 创建一个新对象
2. 将这个新对象的[[Prototype]]连接到调用函数的 prototype 上
3. 绑定调用函数的 this 并调用

用代码来表示就是：

```
function New(fn) {
    var tmp = {}
    tmp.__proto__ = fn.prototype
    fn.call(tmp)
    return tmp
}
```

可以看到，new 帮我们把对象的[[Prototype]]连接到了函数的 prototype 上。
到这儿，思路就清晰了，怎么让对象 a 和对象 b 的[[Prototype]]相连实现 a 继承 b?
只需把 a 的”构造函数”的[[Prototype]]连接到 b 就行了。
来实现一下：

```
function A() {
```

```
}
var b = {
    show: function() {
        console.log('这是来自 b 的方法')
    }
}
A.prototype = b
// 这里修复了原先的 constructor
A.prototype.constructor = A
var a = new A()
a.show()  // 这是来自 b 的方法
```

更简单的 Object.create

ES5 中提供的 Object.create 更简单粗暴，可以直接创建一个对象并将这个对象的[[Prototype]]指向传入的对象

```
var b = {c: 1}
var a = Object.create(b)
console.log(a.c) // 1
```

模拟类继承

在 JavaScript 中没有类的概念，虽然从 ES6 开始拥有了 class 关键字，但其背后仍然是原型链作支撑，所以这里还是用最本质的原型来模拟”类”的继承。这才是 JavaScript 的本来面目！

```
/**
 * 实现 A 继承 B
 */
function B(b) {
    this.b = b
}
function A(a, b) {
    // 调用 B 并绑定 this
    B.call(this, b)
    this.a = a
}
A.prototype = Object.assign({}, B.prototype)
A.prototype.constructor = A
var c = new A(1, 2)
console.log(c.a) // 1
// c 拥有了只有 B 的实例才拥有的 b 属性
console.log(c.b) // 2
```

总结

简单来说，继承即是 copy 和复用，JavaScript 的继承其实就是利用原型链的查找和委托来实现属性和方法的复用，new 关键字和”构造函数”只是连接原型链的工具，这样的工具还有 Object.create。