

原生 JS 灵魂之问，请问你能接得住几个? (上)

前端技匠 5 天前

编者荐语：

掘金大佬，低调又内涵的男人

以下文章来源于前端三元同学，作者神三元



前端三元同学

冲鸭！大前端

笔者最近在对原生 JS 的知识做系统梳理，因为我觉得 JS 作为前端工程师的根本技术，学再多遍都不为过。打算来做一个系列，一共分三次发，以一系列的问题为驱动，当然也会有追问和扩展，内容系统且完整，对初中级选手会有很好的提升，高级选手也会得到复习和巩固。敬请关注！

第一篇：JS 数据类型之问 —— 概念篇

1. JS 原始数据类型有哪些？引用数据类型有哪些？

在 JS 中，存在着 7 种原始值，分别是：

- boolean
- null
- undefined
- number
- string
- symbol
- bigint

引用数据类型：

对象 Object（包含普通对象 - Object，数组对象 - Array，正则对象 - RegExp，日期对象 - Date，数学函数 - Math，函数对象 - Function）

2. 说出下面运行的结果，解释原因。

```
function test(person) {
```

```
person.age = 26
person = {
  name: 'hzj',
  age: 18
}
return person
}
const p1 = {
  name: 'fyq',
  age: 19
}
const p2 = test(p1)
console.log(p1) // -> ?
console.log(p2) // -> ?
```

结果:

```
p1: {name: "fyq", age: 26}
p2: {name: "hzj", age: 18}
```

原因：在函数传参的时候传递的是对象在堆中的内存地址值，test 函数中的实参 person 是 p1 对象的内存地址，通过调用 person.age = 26 确实改变了 p1 的值，但随后 person 变成了另一块内存空间的地址，并且在最后将这另外一份内存空间的地址返回，赋给了 p2。

3.null 是对象吗？为什么？

结论: null 不是对象。

解释：虽然 typeof null 会输出 object，但是这只是 JS 存在的一个悠久 Bug。在 JS 的最初版本中使用的是 32 位系统，为了性能考虑使用低位存储变量的类型信息，000 开头代表是对象，然而 null 表示为全零，所以将它错误的判断为 object。

4.'1'.toString () 为什么可以调用？

其实在这个语句运行的过程中做了这样几件事情：

```
var s = new Object('1');
s.toString();
s = null;
```

第一步：创建 Object 类实例。注意为什么不是 String？由于 Symbol 和 BigInt 的出现，对它们调用 new 都会报错，目前 ES6 规范也不建议用 new 来创建基本类型的包装类。

第二步：调用实例方法。

第三步：执行完方法立即销毁这个实例。

整个过程体现了 基本包装类型 的性质，而基本包装类型恰恰属于基本数据类型，包括 Boolean, Number 和 String。

参考：《JavaScript 高级程序设计 (第三版)》P118

5.0.1+0.2 为什么不等于 0.3？

0.1 和 0.2 在转换成二进制后会无限循环，由于标准位数的限制后面多余的位数会被截掉，此时就已经出现了精度的损失，相加后因浮点数小数位的限制而截断的二进制数字在转换为十进制就会变成 0.30000000000000004。

6. 如何理解 BigInt？

什么是 BigInt？

BigInt 是一种新的数据类型，用于当整数值大于 Number 数据类型支持的范围时。这种数据类型允许我们安全地对 大整数 执行算术操作，表示高分辨率的时间戳，使用大整数 id，等等，而不需要使用库。

为什么需要 BigInt？

在 JS 中，所有的数字都以双精度 64 位浮点格式表示，那这会带来什么问题呢？

这导致 JS 中的 Number 无法精确表示非常大的整数，它会将非常大的整数四舍五入，确切地说，JS 中的 Number 类型只能安全地表示 -9007199254740991 ($-(2^{53}-1)$) 和 9007199254740991 ($(2^{53}-1)$)，任何超出此范围的整数值都可能失去精度。

```
console.log(9999999999999999); // => 10000000000000000
```

同时也会有一定的安全性问题：

```
9007199254740992 === 9007199254740993; // → true 居然是true!
```

如何创建并使用 BigInt?

要创建 BigInt，只需要在数字末尾追加 n 即可。

```
console.log( 9007199254740995n );    // → 9007199254740995n
console.log( 9007199254740995 );     // → 9007199254740996
```

另一种创建 BigInt 的方法是用 BigInt () 构造函数、

```
BigInt("9007199254740995");    // → 9007199254740995n
```

简单使用如下:

```
10n + 20n;    // → 30n
10n - 20n;    // → -10n
+10n;         // → TypeError: Cannot convert a BigInt value to a number
-10n;         // → -10n
10n * 20n;    // → 200n
20n / 10n;    // → 2n
23n % 10n;    // → 3n
10n ** 3n;    // → 1000n

const x = 10n;
++x;         // → 11n
--x;         // → 9n
console.log(typeof x);    //"bigint"
```

值得警惕的点

1. BigInt 不支持一元加号运算符，这可能是某些程序可能依赖于 + 始终生成 Number 的不变量，或者抛出异常。另外，更改 + 的行为也会破坏 asm.js 代码。
2. 因为隐式类型转换可能丢失信息，所以不允许在 bigint 和 Number 之间进行混合操作。当混合使用大整数和浮点数时，结果值可能无法由 BigInt 或 Number 精确表示。

```
10 + 10n;    // → TypeError
```

1. 不能将 BigInt 传递给 Web api 和内置的 JS 函数，这些函数需要一个 Number 类型的数字。尝试这样做会报 TypeError 错误。

```
Math.max(2n, 4n, 6n);    // → TypeError
```

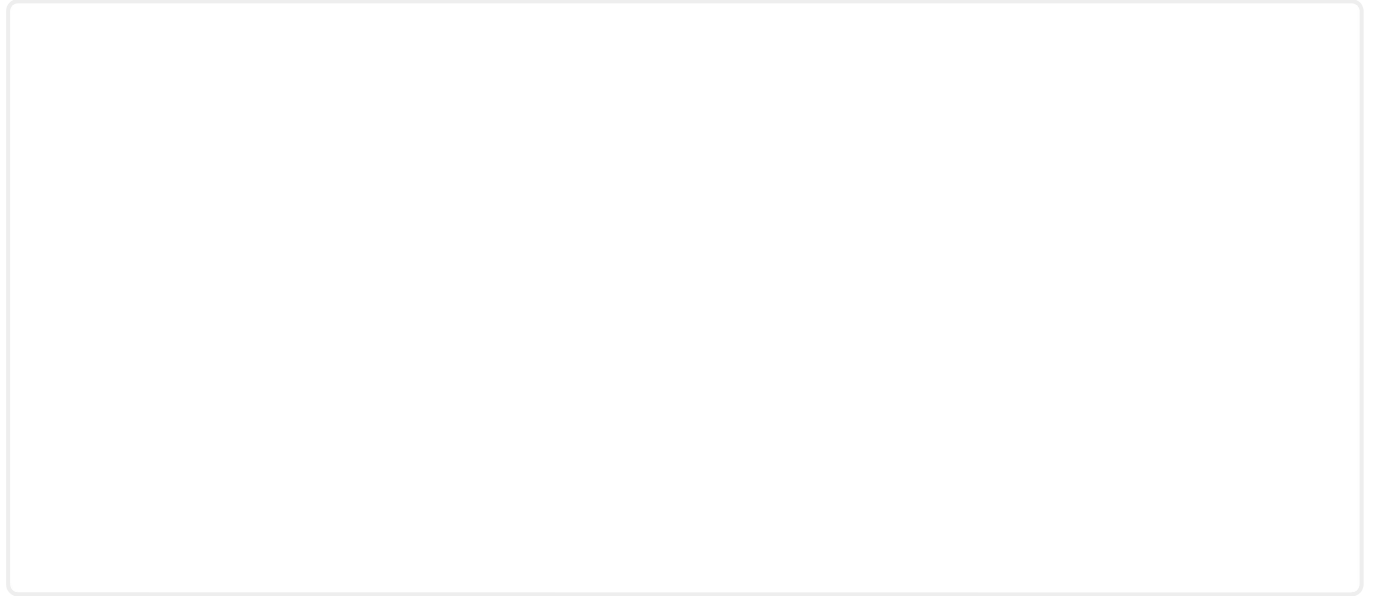
1. 当 Boolean 类型与 BigInt 类型相遇时，BigInt 的处理方式与 Number 类似，换句话说，只要不是 0n，BigInt 就被视为 truthy 的值。

```
if(0n){//条件判断为false
}
if(3n){//条件为true
}
```

1. 元素都为 BigInt 的数组可以进行 sort。
2. BigInt 可以正常地进行位运算，如 |、&、<<、>> 和 ^

浏览器兼容性

caniuse 的结果：



其实现在的兼容性并不怎么好，只有 chrome67、firefox、Opera 这些主流实现，要正式成为规范，其实还有很长的路要走。

我们期待 BigInt 的光明前途！

第二篇：JS 数据类型之问 —— 检测篇

1. typeof 是否能正确判断类型？

对于原始类型来说，除了 null 都可以调用 typeof 显示正确的类型。

```
typeof 1 // 'number'
typeof '1' // 'string'
typeof undefined // 'undefined'
typeof true // 'boolean'
typeof Symbol() // 'symbol'
```

但对于引用数据类型，除了函数之外，都会显示 "object"。

```
typeof [] // 'object'
typeof {} // 'object'
typeof console.log // 'function'
```

因此采用 `typeof` 判断对象数据类型是不合适的，采用 `instanceof` 会更好，`instanceof` 的原理是基于原型链的查询，只要处于原型链中，判断永远为 `true`

```
const Person = function() {}
const p1 = new Person()
p1 instanceof Person // true

var str1 = 'hello world'
str1 instanceof String // false

var str2 = new String('hello world')
str2 instanceof String // true
```

2. instanceof 能否判断基本数据类型？

能。比如下面这种方式：

```
class PrimitiveNumber {
  static [Symbol.hasInstance](x) {
    return typeof x === 'number'
  }
}

console.log(111 instanceof PrimitiveNumber) // true
```

如果你不知道 `Symbol`，可以看看 MDN 上关于 `hasInstance` 的解释。

其实就是自定义 `instanceof` 行为的一种方式，这里将原有的 `instanceof` 方法重定义，换成了 `typeof`，因此能够判断基本数据类型。

3. 能不能手动实现一下 instanceof 的功能？

核心：原型链的向上查找。

```
function myInstanceOf(left, right) {
  //基本数据类型直接返回false
  if(typeof left !== 'object' || left === null) return false;
  //getPrototypeOf是Object对象自带的一个方法，能够拿到参数的原型对象
  let proto = Object.getPrototypeOf(left);
  while(true) {
    //查找到尽头，还没找到
    if(proto == null) return false;
    //找到相同的原型对象
    if(proto == right.prototype) return true;
    proto = Object.getPrototypeOf(proto);
  }
}
```

测试：

```
console.log(myInstanceof("111", String)); //false
console.log(myInstanceof(new String("111"), String)); //true
```

4. Object.is 和 === 的区别？

Object 在严格等于的基础上修复了一些特殊情况下的失误，具体来说就是 +0 和 -0，NaN 和 NaN。源码如下：

```
function is(x, y) {
  if (x === y) {
    //运行到1/x === 1/y的时候x和y都为0，但是1/+0 = +Infinity， 1/-0 = -Infinity，是不一样的
    return x !== 0 || y !== 0 || 1 / x === 1 / y;
  } else {
    //NaN===NaN是false,这是不对的，我们在这里做一个拦截，x !== x，那么一定是 NaN，y 同理
    //两个都是NaN的时候返回true
    return x !== x && y !== y;
  }
}
```

第三篇：JS 数据类型之问 —— 转换篇

1. [] == ![] 结果是什么？为什么？

解析：

== 中，左右两边都需要转换为数字然后进行比较。

[] 转换为数字为 0。

![] 首先是转换为布尔值，由于 [] 作为一个引用类型转换为布尔值为 true，

因此! [] 为 false，进而在转换成数字，变为 0。

0 == 0， 结果为 true

2. JS 中类型转换有哪几种？

JS 中，类型转换只有三种：

- 转换成数字
- 转换成布尔值

■ 转换成字符串

转换具体规则如下：

注意 "Boolean 转字符串" 这行结果指的是 true 转字符串的例子

3. == 和 === 有什么区别？

=== 叫做严格相等，是指：左右两边不仅值要相等，类型也要相等，例如 `'1'===1` 的结果是 `false`，因为一边

`==` 不像 `===` 那样严格，对于一般情况，只要值相等，就返回 `true`，但 `==` 还涉及一些类型转换，它的转换规则如下：

- 两边的类型是否相同，相同的话就比较值的大小，例如 `1==2`，返回 `false`
- 判断的是否是 `null` 和 `undefined`，是的话就返回 `true`
- 判断的类型是否是 `String` 和 `Number`，是的话，把 `String` 类型转换成 `Number`，再进行比较
- 判断其中一方是否是 `Boolean`，是的话就把 `Boolean` 转换成 `Number`，再进行比较
- 如果其中一方为 `Object`，且另一方为 `String`、`Number` 或者 `Symbol`，会将 `Object` 转换成字符串，再进行比较


```
console.log({a: 1} == true); // false
console.log({a: 1} == "[object Object]"); // true
```

4. 对象转原始类型是根据什么流程运行的？

对象转原始类型，会调用内置的 [ToPrimitive] 函数，对于该函数而言，其逻辑如下：

1. 如果 Symbol.toPrimitive () 方法，优先调用再返回
2. 调用 valueOf ()，如果转换为原始类型，则返回
3. 调用 toString ()，如果转换为原始类型，则返回
4. 如果都没有返回原始类型，会报错

```
var obj = {
  value: 3,
  valueOf() {
    return 4;
  },
  toString() {
    return '5'
  },
  [Symbol.toPrimitive]() {
    return 6
  }
}
console.log(obj + 1); // 输出 7
```

5. 如何让 if (a == 1 && a == 2) 条件成立？

其实就是上一个问题的应用。

```
var a = {
  value: 0,
  valueOf: function() {
    this.value++;
    return this.value;
  }
};
console.log(a == 1 && a == 2); // true
```

第四篇：谈谈你对闭包的理解

什么是闭包？

红宝书(p178)上对于闭包的定义：闭包是指有权访问另外一个函数作用域中的变量的函数，

MDN 对闭包的定义为：闭包是指那些能够访问自由变量的函数。

(其中自由变量，指在函数中使用的，但既不是函数参数arguments也不是函数的局部变量的变量，其实就

闭包产生的原因？

首先要明白作用域链的概念，其实很简单，在 ES5 中只存在两种作用域 —— 全局作用域和函数作用域，

当访问一个变量时，解释器会首先在当前作用域查找标示符，如果没有找到，就去父作用域找，直到找到该变量的标示符或者不在父作用域中，这就是作用域链

，值得注意的是，每一个子函数都会拷贝上级的作用域，形成一个作用域的链条。比如：

```
var a = 1;
function f1() {
  var a = 2
  function f2() {
    var a = 3;
    console.log(a); // 3
  }
}
```

在这段代码中，f1 的作用域指向有全局作用域 (window) 和它本身，而 f2 的作用域指向全局作用域 (window)、f1 和它本身。而且作用域是从最底层向上找，直到找到全局作用域 window 为止，如果全局还没有的话就会报错。就这么简单一件事情！

闭包产生的本质就是，当前环境中存在指向父级作用域的引用。还是举上面的例子：

```
function f1() {
  var a = 2
  function f2() {
    console.log(a); // 2
  }
  return f2;
}
var x = f1();
x();
```

这里 x 会拿到父级作用域中的变量，输出 2。因为在当前环境中，含有对 f2 的引用，f2 恰恰引用了 window、f1 和 f2 的作用域。因此 f2 可以访问到 f1 的作用域的变量。

那是不是只有返回函数才算是产生了闭包呢？、

回到闭包的本质，我们只需要让父级作用域的引用存在即可，因此我们还可以这么做：

```
var f3;
function f1() {
```

```
var a = 2;
f3 = function() {
  console.log(a);
}
f1();
f3();
```

让 f1 执行，给 f3 赋值后，等于说现在 f3 拥有了 window、f1 和 f3 本身这几个作用域的访问权限，还是自底向上查找，最近是在 f1 中找到了 a，因此输出 2。

在这里是外面的变量 f3 存在着父级作用域的引用，因此产生了闭包，形式变了，本质没有改变。

闭包有哪些表现形式？

明白了本质之后，我们就来看看，在真实的场景中，究竟在哪些地方能体现闭包的存在？

1. 返回一个函数。刚刚已经举例。
2. 作为函数参数传递

```
var a = 1;
function foo(){
  var a = 2;
  function baz(){
    console.log(a);
  }
  bar(baz);
}
function bar(fn){
  // 这就是闭包
  fn();
}
// 输出2，而不是1
foo();
```

1. 在定时器、事件监听、Ajax 请求、跨窗口通信、Web Workers 或者任何异步中，只要使用了回调函数，实际上就是在使用闭包。

以下的闭包保存的仅仅是 window 和当前作用域。

```
// 定时器
setTimeout(function timeHandler(){
  console.log('111');
}, 100)

// 事件监听
$('#app').click(function(){
  console.log('DOM Listener');
```

```
})
```

1. IIFE (立即执行函数表达式) 创建闭包，保存了 全局作用域 window 和 当前函数的作用域，因此可以全局的变量。

```
var a = 2;  
(function IIFE(){  
  // 输出2  
  console.log(a);  
})();
```

如何解决下面的循环输出问题？

```
for(var i = 1; i <= 5; i++){  
  setTimeout(function timer(){  
    console.log(i)  
  }, 0)  
}
```

为什么会全部输出 6？如何改进，让它输出 1, 2, 3, 4, 5? (方法越多越好)

因为 setTimeout 为宏任务，由于 JS 中单线程 eventLoop 机制，在多线程同步任务执行完后才去执行宏任务，因此循环结束后 setTimeout 中的回调才依次执行，但输出 i 的时候当前作用域没有，往上一级再找，发现了 i，此时循环已经结束，i 变成了 6。因此会全部输出 6。

解决方法：

- 1、利用 IIFE (立即执行函数表达式) 当每次 for 循环时，把此时的 i 变量传递到定时器中

```
for(var i = 1; i <= 5; i++){  
  (function(j){  
    setTimeout(function timer(){  
      console.log(j)  
    }, 0)  
  })(i)  
}
```

- 2、给定时器传入第三个参数，作为 timer 函数的第一个函数参数

```
for(var i=1;i<=5;i++){  
  setTimeout(function timer(j){  
    console.log(j)  
  }, 0, i)  
}
```

- 3、使用 ES6 中的 let

```
for(let i = 1; i <= 5; i++){
  setTimeout(function timer(){
    console.log(i)
  },0)
}
```

let 使 JS 发生革命性的变化，让 JS 有函数作用域变为了块级作用域，用 let 后作用域链不复存在。代码的作用域以块级为单位，以上面代码为例：

```
// i = 1
{
  setTimeout(function timer(){
    console.log(1)
  },0)
}
// i = 2
{
  setTimeout(function timer(){
    console.log(2)
  },0)
}
// i = 3
...
```

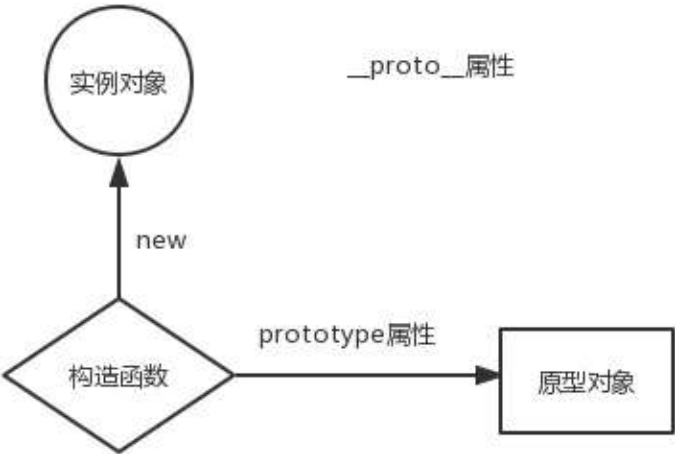
因此能输出正确的结果。

第五篇：谈谈你对原型链的理解

1. 原型对象和构造函数有何关系？

在 JavaScript 中，每当定义一个函数数据类型（普通函数、类）时候，都会天生自带一个 prototype 属性，这个属性指向函数的原型对象。

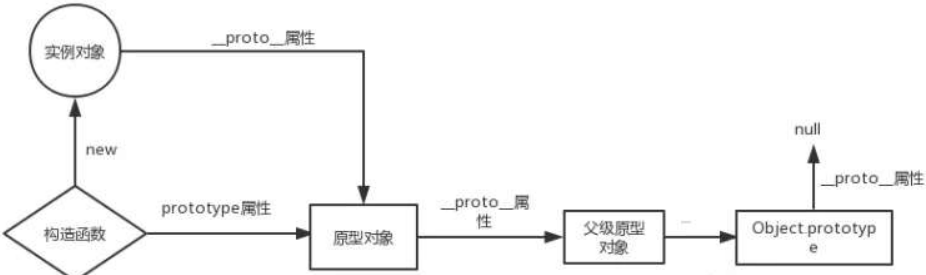
当函数经过 new 调用时，这个函数就成为了构造函数，返回一个全新的实例对象，这个实例对象有一个 **proto** 属性，指向构造函数的原型对象。



前端三元同学

2. 能不能描述一下原型链？

JavaScript 对象通过 `prototype` 指向父类对象，直到指向 `Object` 对象为止，这样就形成了一个原型指向的链条，即原型链。



前端三元同学

- 对象的 `hasOwnProperty ()` 来检查对象自身中是否含有该属性
- 使用 `in` 检查对象中是否含有某个属性时，如果对象中没有但是原型链中有，也会返回 `true`

第六篇：JS 如何实现继承？

第一种：借助 `call`

```
function Parent1(){
```

```
    this.name = 'parent1';
  }
  function Child1(){
    Parent1.call(this);
    this.type = 'child1'
  }
  console.log(new Child1);
```

这样写的时候子类虽然能够拿到父类的属性值，但是问题是父类原型对象中一旦存在方法那么子类无法继承。那么引出下面的方法。

第二种：借助原型链


```
function Parent2() {
  this.name = 'parent2';
  this.play = [1, 2, 3]
}
function Child2() {
  this.type = 'child2';
}
Child2.prototype = new Parent2();

console.log(new Child2());
```

看似没有问题，父类的方法和属性都能够访问，但实际上有一个潜在的不足。举个例子：

```
var s1 = new Child2();
var s2 = new Child2();
s1.play.push(4);
console.log(s1.play, s2.play);
```

可以看到控制台：



明明我只改变了 s1 的 play 属性，为什么 s2 也跟着变了呢？很简单，因为两个实例使用的是同一个原型对象。

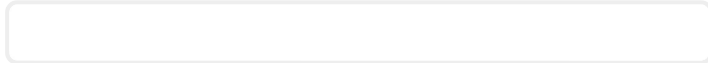
那么还有更好的方式么？

第三种：将前两种组合

```
function Parent3 () {
  this.name = 'parent3';
  this.play = [1, 2, 3];
}
```

```
function Child3() {
  Parent3.call(this);
  this.type = 'child3';
}
Child3.prototype = new Parent3();
var s3 = new Child3();
var s4 = new Child3();
s3.play.push(4);
console.log(s3.play, s4.play);
```

可以看到控制台：



之前的问题都得以解决。但是这里又徒增了一个新问题，那就是 Parent3 的构造函数会多执行了一次（Child3.prototype = new Parent3 ();）。这是我们不愿看到的。那么如何解决这个问题？

第四种：组合继承的优化 1

```
function Parent4 () {
  this.name = 'parent4';
  this.play = [1, 2, 3];
}
function Child4() {
  Parent4.call(this);
  this.type = 'child4';
}
Child4.prototype = Parent4.prototype;
```

这里让将父类原型对象直接给到子类，父类构造函数只执行一次，而且父类属性和方法均能访问，但是我们来测试一下：

```
var s3 = new Child4();
var s4 = new Child4();
console.log(s3)
```

```
▼ Child4 {name: "parent4", play: Array(3), type: "child4"}
  name: "parent4"
  ► play: (3) [1, 2, 3]
  type: "child4"
  ▼ __proto__:
    ► constructor: f Parent4()
    ► __proto__: Object
```


子类实例的构造函数是 Parent4，显然这是不对的，应该是 Child4。

第五种（最推荐使用）：组合继承的优化 1

```
function Parent5 () {
  this.name = 'parent5';
  this.play = [1, 2, 3];
}
function Child5() {
  Parent5.call(this);
  this.type = 'child5';
}
Child5.prototype = Object.create(Parent5.prototype);
Child5.prototype.constructor = Child5;
```

这是最推荐的一种方式，接近完美的继承，它的名字也叫做寄生组合继承。

ES6 的 extends 被编译后的 JavaScript 代码

ES6 的代码最后都是要在浏览器上能够跑起来的，这中间就利用了 babel 这个编译工具，将 ES6 的代码编译成 ES5 让一些不支持新语法的浏览器也能运行。

那最后编译成了什么样子呢？

```
function _possibleConstructorReturn (self, call) {
  // ...
  return call && (typeof call === 'object' || typeof call === 'function') ? call :
}

function _inherits (subClass, superClass) {
  // ...
  //看到没有
  subClass.prototype = Object.create(superClass && superClass.prototype, {
    constructor: {
      value: subClass,
      enumerable: false,
      writable: true,
      configurable: true
    }
  });
  if (superClass) Object.setPrototypeOf ? Object.setPrototypeOf(subClass, superClass) :
}

var Parent = function Parent () {
  // 验证是否是 Parent 构造出来的 this
  _classCallCheck(this, Parent);
};
```

```
var Child = (function (_Parent) {
  _inherits(Child, _Parent);

  function Child () {
    _classCallCheck(this, Child);

    return _possibleConstructorReturn(this, (Child.__proto__ || Object.getPr
  }

  return Child;
})(Parent));
```

核心是_inherits 函数，可以看到它采用的依然也是第五种方式 —— 寄生组合继承方式，同时证明了这种方式的成功。不过这里加了一个 Object.setPrototypeOf (subClass, superClass)，这是用来干啥的呢？

答案是用来继承父类的静态方法。这也是原来的继承方式疏忽掉的地方。

追问：面向对象的设计一定是好的设计吗？

不一定。从继承的角度说，这一设计是存在巨大隐患的。

从设计思想上谈谈继承本身的问题

假如现在有不同品牌的车，每辆车都有 drive、music、addOil 这三个方法。

```
class Car{
  constructor(id) {
    this.id = id;
  }
  drive(){
    console.log("wuwuwu!");
  }
  music(){
    console.log("lalala!")
  }
  addOil(){
    console.log("哦哟！ ")
  }
}
class otherCar extends Car{}
```

现在可以实现车的功能，并且以此去扩展不同的车。

但是问题来了，新能源汽车也是车，但是它并不需要 addOil (加油)。

如果让新能源汽车的类继承 Car 的话，也是有问题的，俗称 "大猩猩和香蕉" 的问题。大猩猩手里有香蕉，但是我现在明明只需要香蕉，却拿到了一只大猩猩。也就是说加油这个方法，我现在是不需要的，但是由于继承的原因，也给到子类了。

继承的最大问题在于：无法决定继承哪些属性，所有属性都得继承。

当然你可能会说，可以再创建一个父类啊，把加油的方法给去掉，但是这也是有问题的，一方面父类是无法描述所有子类的细节情况的，为了不同的子类特性去增加不同的父类，代码势必会大量重复，另一方面一旦子类有所变动，父类也要进行相应的更新，代码的耦合性太高，维护性不好。

那如何解决继承的诸多问题呢？

用组合，这也是当今编程语法发展的趋势，比如 go lang 完全采用的是面向组合的设计方式。

顾名思义，面向组合就是先设计一系列零件，然后将这些零件进行拼装，来形成不同的实例或者类。

```
function drive(){
  console.log("wuwuwu!");
}
function music(){
  console.log("lalala!")
}
function addOil(){
  console.log("哦哟！")
}

let car = compose(drive, music, addOil);
let newEnergyCar = compose(drive, music);
```

代码干净，复用性也很好。这就是面向组合的设计方式。