前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏

查看详情 >

我们不背诵 API，只实现 API

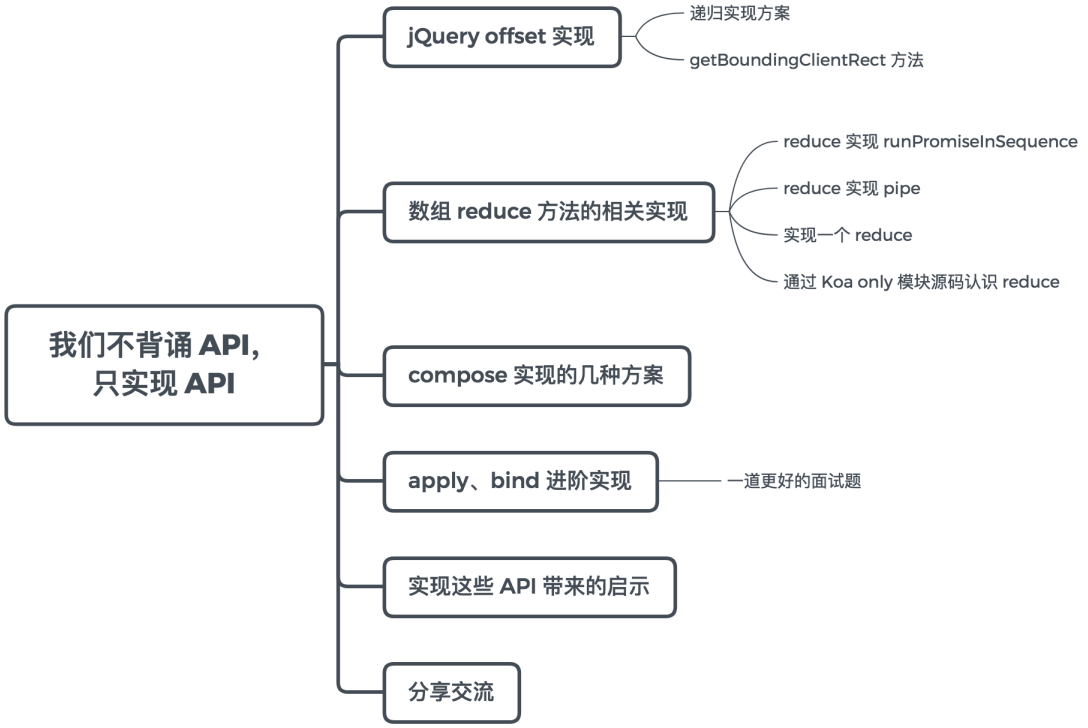
有不少刚入行的同学跟我说：「JavaScript 很多 API 记不清楚怎么办？数组的这方法、那方法总是傻傻分不清楚，该如何是好？操作 DOM 的方式今天记、明天忘，真让人奔溃！」甚至有的开发者在讨论面试时，总向我抱怨：「面试官总爱纠结 API 的使用，甚至 jQuery 某些方法的参数顺序都需要让我说清楚！」

我认为，对于反复使用的方法，所有人都要做到「机械记忆」，能够反手写出。一些貌似永远记不清的 API 只是因为用得不够多而已。

在做面试官时，我从来不强求开发者准确无误地「背诵」 API。相反，我喜欢从另外一个角度来考察面试者：「**既然记不清使用方法，那么我告诉你它的使用方法，你来实现一个吧！**」实现一个 API，除了可以考察面试者对这个 API 的理解，更能体现开发者的编程思维和代码能力。对于积极上进的前端工程师，模仿并实现一些经典方法，应该是「家常便饭」，这是比较基本的要求。

本小节，我根据了解的面试题目和作为面试官的经历，挑了几个典型的 API，通过对其不同程度，不同方式的实现，来覆盖 JavaScript 中的部分知识点和编程要领。通过学习本节内容，期待你不仅能领会代码奥义，更应该学习举一反三的方法。

API 主题的相关知识点如下：



jQuery offset 实现

这个话题演变自今日头条某部门面试题。当时面试官提问：「如何获取文档中任意一个元素距离文档 `document` 顶部的距离？」

熟悉 jQuery 的读者应该对 `offset` 方法并不陌生，它返回或设置匹配元素相对于文档的偏移（位置）。这个方法返回的对象包含两个整型属性：`top` 和 `left`，以像素计。如果可以使用 jQuery， 我们可以直接调取该 API 获得结果。但是，**如果用原生 JavaScript 实现，也就是说手动实现 jQuery `offset` 方法，该如何着手呢?**

主要有两种思路：

通过递归实现

通过 `getBoundingClientRect` API 实现

递归实现方案

我们通过遍历目标元素、目标元素的父节点、父节点的父节点……依次溯源，并累加这些遍历过的节点相对于其最近祖先节点（且 position 属性非 static）的偏移量，向上直到 document，累加即可得到结果。

其中，我们需要使用 JavaScript 的 offsetTop 来访问一个 DOM 节点上边框相对离其本身最近、且 position 值为非 static 的祖先元素的垂直偏移量。具体实现为：

```
const offset = ele => {
   let result = {
       top: 0,
       left: 0
   }

const getOffset = (node, init) => {
       if (node.nodeType !== 1) {
           return
       }

       position = window.getComputedStyle(node)
['position']

       if (typeof(init) === 'undefined' && position ===
'static') {
           getOffset(node.parentNode)
           return
       }

       result.top = node.offsetTop + result.top -
node.scrollTop
       result.left = node.offsetLeft + result.left -
node.scrollLeft

       if (position === 'fixed') {
           return
       }
```

```
        getOffset(node.parentNode)
    }

    // 当前 DOM 节点的 display === 'none' 时，直接返回 {top:
0, left: 0}
    if (window.getComputedStyle(ele)['display'] === 'none')
{
        return result
    }

    let position

    getOffset(ele, true)

    return result

}
```

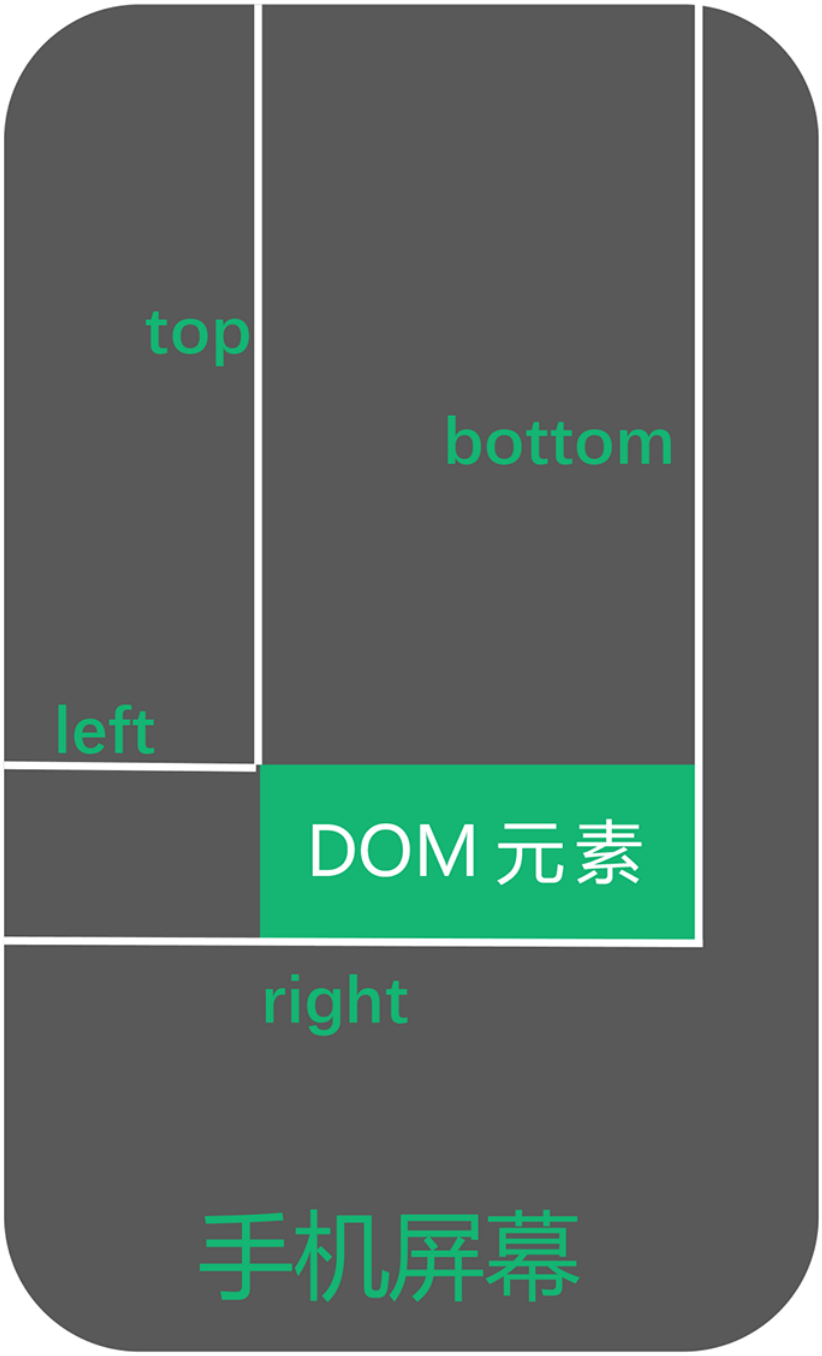
上述代码并不难理解，使用递归实现。如果节点 `node.nodeType` 类型不是 `Element(1)`，则跳出；如果相关节点的 `position` 属性为 `static`，则不计入计算，进入下一个节点（其父节点）的递归。如果相关属性的 `display` 属性为 `none`，则应该直接返回 0 作为结果。

这个实现很好地考察了开发者对于递归的初级应用、以及对 JavaScript 方法的掌握程度。

接下来，我们换一种思路，用一个相对较新的 API：`getBoundingClientRect` 来实现 jQuery `offset` 方法。

getBoundingClientRect 方法

`getBoundingClientRect` 方法用来描述一个元素的具体位置，该位置的下面四个属性都是相对于视口左上角的位置而言的。对某一节点执行该方法，它的返回值是一个 DOMRect 类型的对象。这个对象表示一个矩形盒子，它含有：`left`、`top`、`right` 和 `bottom` 等只读属性。



请参考实现代码：

```
const offset = ele => {
    let result = {
        top: 0,
        left: 0
```

```
    }
    // 当前为 IE11 以下，直接返回 {top: 0, left: 0}
    if (!ele.getClientRects().length) {
        return result
    }

    // 当前 DOM 节点的 display === 'none' 时，直接返回 {top:
0, left: 0}
    if (window.getComputedStyle(ele)['display'] === 'none')
{
        return result
    }

    result = ele.getBoundingClientRect()
    var docElement = ele.ownerDocument.documentElement

    return {
        top: result.top + window.pageYOffset -
docElement.clientTop,
        left: result.left + window.pageXOffset -
docElement.clientLeft
    }
}
```

需要注意的细节有：

node.ownerDocument.documentElement 的用法可能大家比较陌生，ownerDocument 是 DOM 节点的一个属性，它返回当前节点的顶层的 document 对象。ownerDocument 是文档，documentElement 是根节点。事实上，ownerDocument 下含 2 个节点：

documentElement

`docElement.clientTop`，`clientTop` 是一个元素顶部边框的宽度，不包括顶部外边距或内边距。

> 除此之外，该方法实现就是简单的几何运算，边界 case 和兼容性处理，也并不难理解。

从这道题目看出，相比考察「死记硬背」 API，这样的实现更有意义。站在面试官的角度，我往往会给面试者（开发者）提供相关的方法提示，以引导其给出最后的方案实现。

数组 `reduce` 方法的相关实现

数组方法非常重要：**因为数组就是数据，数据就是状态，状态反应着视图**。对数组的操作我们不能陌生，其中 `reduce` 方法更要做到驾轻就熟。我认为这个方法很好地体现了「函数式」理念，也是当前非常热门的考察点之一。

我们知道 `reduce` 方法是 ES5 引入的，reduce 英文解释翻译过来为「减少，缩小，使还原，使变弱」，MDN 对该方法直述为：

The reduce method applies a function against an accumulator and each value of the array (from left-to-right) to reduce it to a single value.

它的使用语法：

```
arr.reduce(callback[, initialValue])
```

这里我们简要介绍一下。

> `reduce` 第一个参数 `callback` 是核心，它对数组的每一项进行「叠加加工」，其最后一次返回值将作为 `reduce`方法的最终返回值。 它包含 4 个参数：
>
> `previousValue` 表示「上一次」 `callback` 函数的返回值
>
> `currentValue` 数组遍历中正在处理的元素

`currentIndex` 可选，表示 `currentValue` 在数组中对应的索引。如果提供了 `initialValue`，则起始索引号为 0，否则为 1

`array` 可选，调用 `reduce()` 的数组

`initialValue` 可选，作为第一次调用 `callback` 时的第一个参数。如果没有提供 `initialValue`，那么数组中的第一个元素将作为 `callback` 的第一个参数。

reduce 实现 runPromiseInSequence

我们看它的**一个典型应用**，按顺序运行 Promise：

```
const runPromiseInSequence = (array, value) =>
array.reduce(
   (promiseChain, currentFunction) =>
promiseChain.then(currentFunction),
   Promise.resolve(value)
)
```

runPromiseInSequence 方法将会被一个每一项都返回一个 Promise 的数组调用，并且依次执行数组中的每一个 Promise，请读者仔细体会。如果觉得晦涩，可以参考示例：

```
const f1 = () => new Promise((resolve, reject) => {
   setTimeout(() => {
       console.log('p1 running')
       resolve(1)
   }, 1000)
})

const f2 = () => new Promise((resolve, reject) => {
   setTimeout(() => {
       console.log('p2 running')
       resolve(2)
   }, 1000)
})
```

```
const array = [f1, f2]

const runPromiseInSequence = (array, value) =>
array.reduce(
   (promiseChain, currentFunction) =>
promiseChain.then(currentFunction),
   Promise.resolve(value)
)

runPromiseInSequence(array, 'init')
```

执行结果如下图：

reduce 实现 pipe

reduce 的另外一个**典型应用**可以参考函数式方法 pipe 的实现：pipe(f, g, h) 是一个 curry 化函数，它返回一个新的函数，这个新的函数将会完成 (...args) => h(g(f(...args))) 的调用。即 pipe 方法返回的函数会接收一个参数，这个参数传递给 pipe 方法第一个参数，以供其调用。

```
const pipe = (...functions) => input => functions.reduce(
   (acc, fn) => fn(acc),
   input
)
```

仔细体会 runPromiseInSequence 和 pipe 这两个方法，它们都是 reduce 应用的典型场景。

实现一个 reduce

那么我们该如何实现一个 reduce 呢？参考来自 MDN 的 polyfill：

```
if (!Array.prototype.reduce) {
 Object.defineProperty(Array.prototype, 'reduce', {
   value: function(callback /*, initialValue*/) {
     if (this === null) {
       throw new TypeError( 'Array.prototype.reduce ' +
         'called on null or undefined' )
     }
     if (typeof callback !== 'function') {
       throw new TypeError( callback +
         ' is not a function')
     }

     var o = Object(this)

     var len = o.length >>> 0

     var k = 0
     var value

     if (arguments.length >= 2) {
       value = arguments[1]
     } else {
       while (k < len && !(k in o)) {
         k++
       }
```

```
        if (k >= len) {
          throw new TypeError( 'Reduce of empty array ' +
            'with no initial value' )
        }
        value = o[k++]
      }

      while (k < len) {
        if (k in o) {
          value = callback(value, o[k], k, o)
        }

        k++
      }

      return value
    }
  })
}
```

上述代码中使用了 value 作为初始值，并通过 while 循环，依次累加计算出 value 结果并输出。但是相比 MDN 上述实现，我个人更喜欢的实现方案是：

```
Array.prototype.reduce = Array.prototype.reduce ||
function(func, initialValue) {
    var arr = this
    var base = typeof initialValue === 'undefined' ? arr[0]
: initialValue
    var startPoint = typeof initialValue === 'undefined' ?
1 : 0
    arr.slice(startPoint)
        .forEach(function(val, index) {
            base = func(base, val, index + startPoint, arr)
        })
    return base
}
```

核心原理就是使用 `forEach` 来代替 `while` 实现结果的累加，它们本质上是相同的。

我也同样看了下 ES5-shim 里的 pollyfill，跟上述思路完全一致。唯一的区别在于：我用了 `forEach` 迭代而 ES5-shim 使用的是简单的 `for` 循环。实际上，如果「杠精」一些，我们会指出数组的 `forEach` 方法也是 ES5 新增的。因此，用 ES5 的一个 API（`forEach`），去实现另外一个 ES5 的 API（`reduce`），这并没什么实际意义——这里的 pollyfill 就是在不兼容 ES5 的情况下，模拟的降级方案。此处不多做追究，因为根本目的还是希望读者对 `reduce` 有一个全面透彻的了解。

通过 Koa only 模块源码认识 `reduce`

通过了解并实现 `reduce` 方法，我们对它已经有了比较深入的认识。最后，再来看一个 `reduce` 使用示例——通过 Koa 源码的 only 模块，加深印象：

```
var o = {
   a: 'a',
   b: 'b',
   c: 'c'
}
only(o, ['a','b'])   // {a: 'a',  b: 'b'}
```

该方法返回一个经过指定筛选属性的新对象。
only 模块实现：

```
var only = function(obj, keys){
   obj = obj || {}
   if ('string' == typeof keys) keys = keys.split(/ +/)
   return keys.reduce(function(ret, key) {
       if (null == obj[key]) return ret
       ret[key] = obj[key]
       return ret
   }, {})
}
```

小小的 reduce 及其衍生场景有很多值得我们玩味、探究的地方。举一反三，活学活用是技术进阶的关键。

compose 实现的几种方案

函数式理念——这一古老的概念如今在前端领域「遍地开花」。函数式很多思想都值得借鉴，其中一个细节：compose 因为其巧妙的设计而被广泛运用。对于它的实现，从面向过程式到函数式实现，风格迥异，值得我们探究。在面试当中，也经常有面试官要求实现 compose 方法，我们先看什么是 compose。

compose 其实和前面提到的 pipe 一样，就是执行一连串不定长度的任务（方法），比如：

```
let funcs = [fn1, fn2, fn3, fn4]
let composeFunc = compose(...funcs)
```

执行：

```
composeFunc(args)
```

就相当于：

```
fn1(fn2(fn3(fn4(args))))
```

总结一下 compose 方法的关键点：

> compose 的参数是函数数组，返回的也是一个函数
>
> compose 的参数是任意长度的，所有的参数都是函数，执行方向是自右向左的，因此初始函数一定放到参数的最右面
>
> compose 执行后返回的函数可以接收参数，这个参数将作为初始函数的参数，所以初始函数的参数是多元的，初始函数的返回结果将作为下一个函数的参数，以此类推。因此除了初始函数之外，其他函数的接收值是一元的

我们发现，实际上，compose 和 pipe 的差别只在于调用顺序的不同：

```
// compose
fn1(fn2(fn3(fn4(args))))

// pipe
fn4(fn3(fn2(fn1(args))))
```

即然跟我们先前实现的 pipe 方法如出一辙，那么还有什么好深入分析的呢？请继续阅读，看看还能玩出什么花儿来。

compose 最简单的实现是面向过程的：

```
const compose = function(...args) {
   let length = args.length
   let count = length - 1
   let result
   return function f1 (...arg1) {
       result = args[count].apply(this, arg1)
       if (count <= 0) {
           count = length - 1
           return result
       }
       count--
       return f1.call(null, result)
   }
}
```

这里的关键是用到了**闭包**，使用闭包变量储存结果 result 和函数数组长度以及遍历索引，并利用递归思想，进行结果的累加计算。整体实现符合正常的面向过程思维，不难理解。

聪明的读者可能也会意识到，利用上文所讲的 reduce 方法，应该能更**函数式**地解决问题：

```
const reduceFunc = (f, g) => (...arg) => g.call(this,
f.apply(this, arg))
```

```
const compose = (...args) =>
args.reverse().reduce(reduceFunc, args.shift())
```

通过前面的学习，结合 call、apply 方法，这样的实现并不难理解。

我们继续开拓思路，「既然涉及串联和流程控制」，那么还可以使用 Promise 实现：

```
const compose = (...args) => {
   let init = args.pop()
   return (...arg) =>
   args.reverse().reduce((sequence, func) =>
     sequence.then(result => func.call(null, result))
   , Promise.resolve(init.apply(null, arg)))
}
```

这种实现利用了 Promise 特性：首先通过 `Promise.resolve(init.apply(null, arg))` 启动逻辑，启动一个 resolve 值为最后一个函数接收参数后的返回值，依次执行函数。因为 `promise.then()` 仍然返回一个 Promise 类型值，所以 reduce 完全可以按照 Promise 实例执行下去。

既然能够使用 Promise 实现，那么 **generator** 当然应该也可以实现。这里给大家留一个思考题，感兴趣的读者可以尝试，欢迎在评论区或读者群讨论。

最后，我们再看下社区上著名的 lodash 和 Redux 的实现。

lodash 版本

```
// lodash 版本
var compose = function(funcs) {
   var length = funcs.length
   var index = length
   while (index--) {
       if (typeof funcs[index] !== 'function') {
           throw new TypeError('Expected a function');
```

```
        }
    }
    return function(...args) {
        var index = 0
        var result = length ? funcs.reverse()
[index].apply(this, args) : args[0]
        while (++index < length) {
            result = funcs[index].call(this, result)
        }
        return result
    }
}
```

lodash 版本更像我们的第一种实现方式，理解起来也更容易。

Redux 版本

```
// Redux 版本
function compose(...funcs) {
    if (funcs.length === 0) {
        return arg => arg
    }

    if (funcs.length === 1) {
        return funcs[0]
    }

    return funcs.reduce((a, b) => (...args) =>
a(b(...args)))
}
```

总之，还是充分利用了数组的 reduce 方法。

函数式概念确实有些抽象，需要开发者仔细琢磨，并动手调试。一旦顿悟，必然会感受到其中的优雅和简洁。

apply、bind 进阶实现

面试中关于 this 绑定的相关话题如今已经「泛滥」，同时对 bind 方法的实现，社区上也有相关讨论。但是很多内容尚不系统，且存在一些瑕疵。这里简单摘录我 2017 年年初写的文章 从一道面试题，到「我可能看了假源码」 来递进讨论。在《一网打尽 this》一课中，我们介绍过对 bind 的实现，这里进一步展开讲解。

此处不再赘述 bind 函数的使用，尚不清楚的读者可以自行补充一下基础知识。我们先来看一个初级实现版本：

```
Function.prototype.bind = Function.prototype.bind ||
function (context) {
    var me = this;
    var argsArray = Array.prototype.slice.call(arguments);
    return function () {
        return me.apply(context, argsArray.slice(1))
    }
}
```

这是一般合格开发者提供的答案，如果面试者能写到这里，给他 60 分。

先简要解读一下：

基本原理是使用 apply 进行模拟 bind。函数体内的 this 就是需要绑定 this 的函数，或者说是原函数。最后使用 apply 来进行参数（context）绑定，并返回。

与此同时，将第一个参数（context）以外的其他参数，作为提供给原函数的预设参数，这也是基本的「 curry 化」基础。

上述实现方式，我们返回的参数列表里包含：argsArray.slice(1)，**它的问题在于存在预置参数功能丢失的现象。**

想象我们返回的绑定函数中，如果想实现预设传参（就像 bind 所实现的那样），就面临尴尬的局面。真正实现「 curry 化」的「完美方式」是：

```
Function.prototype.bind = Function.prototype.bind ||
function (context) {
    var me = this;
    var args = Array.prototype.slice.call(arguments, 1);
    return function () {
        var innerArgs =
Array.prototype.slice.call(arguments);
        var finalArgs = args.concat(innerArgs);
        return me.apply(context, finalArgs);
    }
}
```

但继续探究，我们注意 bind 方法中：bind 返回的函数如果作为构造函数，搭配 new 关键字出现的话，我们的绑定 this 就需要「被忽略」，this 要绑定在实例上。也就是说，new 的操作符要高于 bind 绑定，兼容这种情况的实现：

```
Function.prototype.bind = Function.prototype.bind ||
function (context) {
    var me = this;
    var args = Array.prototype.slice.call(arguments, 1);
    var F = function () {};
    F.prototype = this.prototype;
    var bound = function () {
        var innerArgs =
Array.prototype.slice.call(arguments);
        var finalArgs = args.concat(innerArgs);
        return me.apply(this instanceof F ? this : context
|| this, finalArgs);
    }
    bound.prototype = new F();
    return bound;
}
```

如果你认为这样就完了，其实我会告诉你说，高潮才刚要上演。曾经的我也认为上述方法已经比较完美了，直到我看了 es5-shim 源码（已适当删减）：

```
function bind(that) {
   var target = this;
   if (!isCallable(target)) {
       throw new TypeError('Function.prototype.bind called
on incompatible ' + target);
   }
   var args = array_slice.call(arguments, 1);
   var bound;
   var binder = function () {
       if (this instanceof bound) {
           var result = target.apply(
               this,
               array_concat.call(args,
array_slice.call(arguments))
           );
           if ($Object(result) === result) {
               return result;
           }
           return this;
       } else {
           return target.apply(
               that,
               array_concat.call(args,
array_slice.call(arguments))
           );
       }
   };
   var boundLength = max(0, target.length - args.length);
   var boundArgs = [];
   for (var i = 0; i < boundLength; i++) {
       array_push.call(boundArgs, '$' + i);
   }
   bound = Function('binder', 'return function (' +
boundArgs.join(',') + '){ return binder.apply(this,
arguments); }')(binder);

   if (target.prototype) {
```

```
        Empty.prototype = target.prototype;
        bound.prototype = new Empty();
        Empty.prototype = null;
    }
    return bound;
}
```

es5-shim 的实现到底在「搞什么鬼」呢？你可能不知道，其实每个函数都有 length 属性。对，就像数组和字符串那样。函数的 length 属性，用于表示函数的形参个数。更重要的是函数的 length 属性值是不可重写的。我写了个测试代码来证明：

```
function test (){}
test.length  // 输出 0
test.hasOwnProperty('length')  // 输出 true
Object.getOwnPropertyDescriptor('test', 'length')
// 输出：
// configurable: false,
// enumerable: false,
// value: 4,
// writable: false
```

说到这里，那就好解释了：**es5-shim 是为了最大限度地进行兼容，包括对返回函数 length 属性的还原**。而如果按照我们之前实现的那种方式，length 值始终为零。因此，既然不能修改 length 的属性值，那么在初始化时赋值总可以吧！于是我们可通过 eval 和 new Function 的方式动态定义函数。但是出于安全考虑，在某些浏览器中使用 eval 或者 Function() 构造函数都会抛出异常。然而巧合的是，这些无法兼容的浏览器基本上都实现了 bind 函数，这些异常又不会被触发。在上述代码里，重设绑定函数的 length 属性：

```
var boundLength = max(0, target.length - args.length)
```

构造函数调用情况，在 binder 中也有效兼容：

```
if (this instanceof bound) {
    ... // 构造函数调用情况
```

```
} else {
    ... // 正常方式调用
}

if (target.prototype) {
    Empty.prototype = target.prototype;
    bound.prototype = new Empty();
    // 进行垃圾回收清理
    Empty.prototype = null;
}
```

对比过几版的 polyfill 实现，对于 bind 应该有了比较深刻的认识。这一系列实现有效地考察了很重要的知识点：比如 this 的指向、JavaScript 闭包、原型与原型链，设计程序上的边界 case 和兼容性考虑经验等硬素质。

一道更好的面试题

最后，现如今在很多面试中，面试官都会以「实现 bind」作为题目。**如果是我，现在可能会规避这个很容易「应试」的题目，而是别出心裁，让面试者实现一个 「call/apply」**。我们往往用 call/apply 模拟实现 bind，而直接实现 call/apply 也算简单：

```
Function.prototype.applyFn = function (targetObject,
argsArray) {
    if(typeof argsArray === 'undefined' || argsArray ===
null) {
        argsArray = []
    }

    if(typeof targetObject === 'undefined' || targetObject
=== null){
        targetObject = window
    }

    targetObject = new Object(targetObject)

    const targetFnKey = 'targetFnKey'
```

```
    targetObject[targetFnKey] = this

    const result = targetObject[targetFnKey](...argsArray)
    delete targetObject[targetFnKey]
    return result
}
```

这样的代码不难理解，函数体内的 this 指向了调用 applyFn 的函数。为了将该函数体内的 this 绑定在 targetObject 上，我们采用了隐式绑定的方法：targetObject[targetFnKey](...argsArray)。

细心的读者会发现，这里存在一个问题：如果 targetObject 对象本身就存在 targetFnKey 这样的属性，那么在使用 applyFn 函数时，原有的 targetFnKey 属性值就会被覆盖，之后被删除。解决方案可以使用 ES6 Sybmol() 来保证键的唯一性；另一种解决方案是用 Math.random() 实现独一无二的 key，这里我们不再赘述。

实现这些 API 带来的启示

这些 API 的实现并不算复杂，却能恰如其分地考验开发者的 JavaScript 基础。基础是地基，是探究更深入内容的钥匙，是进阶之路上最重要的一环，需要每个开发者重视。在前端技术快速发展迭代的今天，在「前端市场是否饱和」，「前端求职火爆异常」，「前端入门简单，钱多人傻」等众说纷纭的浮躁环境下，对基础内功的修炼就显得尤为重要。这也是你在前端路上能走多远、走多久的关键。

从面试的角度看，面试题归根结底是对基础的考察，只有对基础烂熟于胸，才能具备突破面试的基本条件。

课程代码仓库请单击这里查看。

点击查看下一节

JavaScript 知识图谱和高频考点梳理