【前端面试】手动实现一个bind函数?

前言

this关键词可以说在JavaScript有着举足轻重的地位了！我们对它简直是又爱又恨，它在给我们带来简便快乐的同时，还给我们带来了痛苦，因为它的多变性，让我们开发人员经常不知道this到底指向的哪里，当然这有一部分是太菜的原因，另外一部分原因是this指向还是比较复杂的。为了完全控制this指向，也就是我们常说的this指向的显示绑定，我们可以使用bind方法来显示绑定this。

1.bind函数用法

bind()方法用于创建一个新的函数，这个新函数接收的第一个参数代表的就是this，利用bind()函数我就就可以任意改变函数内部的this指向了。

官网的解释：

> bind() 方法创建一个新的函数，在 bind() 被调用时，这个新函数的 this 被指定为 bind() 的第一个参数，而其余参数将作为新函数的参数，供调用时使用。

官网解释得也比较通透明了，我们这儿为了让大家更加深刻理解bind的用法，利用代码来演示一下。

示例代码：

```
<script>
  let obj = {
    name: "小猪课堂",
    age: 20
  }
  // 声明一个函数
  function fn(a, b, c) {
    console.log("函数内部this指向:", this);
    console.log("参数列表:", a, b, c);
  }
  // 使用bind创建一个新函数
  let newFn = fn.bind(obj, 10, 20, 30);
  // 调用新函数
  newFn();
  // 调用旧函数
  fn(10, 20, 30);
```

```
</script>
```

输出结果：

```
函数内部this指向： ▶ {name: '小猪课堂', age: 20}
参数列表： 10 20 30
函数内部this指向：
▶ Window {window: Window, self: Window, document:
参数列表： 10 20 30
```

上段代码中我们声明了一个函数fn，并且在函数内部打印了this以及参数，然后我们利用bind()创建了一个新的函数，且第一个参数传入了obj，意味着新函数内部的this指向了obj。分别执行两个函数，两个函数内部的this指向一个指向了全局，一个指向了window。

2.bind函数的特点

如果我们想要手动实现一个bind函数，那么非常有必要了解bind函数的特点，所以知己知彼方能百战不殆。

从上一节中的代码我们大致总结出了bind函数的以下几个特点：

2.1 返回一个新函数

bind函数实际上是对原函数的一个拷贝，原函数认可以按照原逻辑处理。

示例代码：

```
<script>
  let obj = {
    name: "小猪课堂",
    age: 20
  }
  // 声明一个函数
  function fn(a, b, c) {
    console.log("函数内部this指向:", this);
    console.log("参数列表:", a, b, c);
  }
  // 使用bind创建一个新函数
  let newFn = fn.bind(obj, 10, 20, 30);
  console.log(typeof newFn); // 'function'
</script>
```

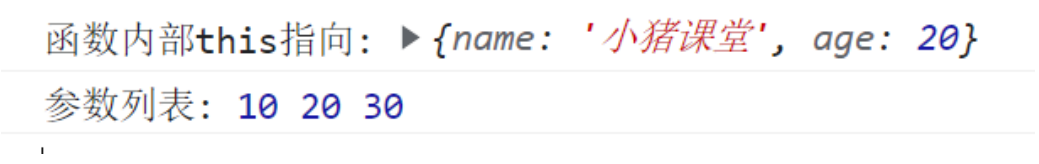
2.2 新函数仍可继续传参

bind函数创建的新函数是可以接收参数的，之前的列子中我们是在创建的时候就将参数传递了进去，实际上可以不必传。

示例代码：

```
<script>
  let obj = {
    name: "小猪课堂",
    age: 20
  }
  // 声明一个函数
  function fn(a, b, c) {
    console.log("函数内部this指向:", this);
    console.log("参数列表:", a, b, c);
  }
  let newFn = fn.bind(obj, 10);
  newFn(20, 30);
</script>
```

输出结果：



上面的输出结果和我们直接在创建的时候传递所有参数得出的结果一致，而且上段代码中我们的参数是分开传递的，也就是说使用bind创建新函数后，调用新函数时，函数接收的参数是调用传入的参数+创建时传入的参数。

2.3 新函数作为构造函数

如果我们将使用bind创建的新函数当作构造函数来执行，那么this的指向将和bind创建时绑定的无关，它会指向一个新的引用。

示例代码：

```
<script>
  let obj = {
    name: "小猪课堂",
    age: 20
  }
  function fn(name) {
    this.name = name;
    console.log("函数内部this指向:", this);
  }
  let newFn = fn.bind(obj);
  let obj2 = new newFn("构造函数");
</script>
```

输出结果：

```
函数内部this指向： ▶fn {name: '构造函数'}
>
```

上段代码中我们使用bind新创建了一个函数newFn，而且将这个函数的this指向了obj，但是我们后续使用的时候使用了new关键词来创建，这个时候函数内部的this指向不在指向obj了，而是指向了fn。

那么既然this指向了fn，那么我们在fn原型上添加属性或方法后，obj2是能访问到的。

3.实现bind函数

既然我们知道了bind的几个贴点，那么我们遵循它的即可特点就可以来实现它了。

1. 首先它是返回一个新函数，我们可以先把架子搭起来，**代码如下：**

```
Function.prototype.myBind = function () {
  // 返回新函数
  return function () {
    // 代码先省略
  }
}
```

上段代码只是一个基本的架子，我们在里面填充代码就好了。

2. 接下来我们需要将函数的this指向为传进来的第一个参数，并且使用bind创建的新函数可以继续接收参数，**代码如下：**

```
<script>
  let obj = {
    name: "小猪课堂",
    age: 20
  }
  // 手写bind函数
  Function.prototype.myBind = function (context) {
    const _this = this; // 当前函数
    let args = Array.from(arguments).slice(1); // 将参数列表转化为数组,出去第一
    // 返回新函数
    return function () {
      // context 是传进来的this
      _this.apply(context, args.concat(Array.from(arguments))); // 利用apply
    }
  }
  // 声明一个函数
  function fn(a, b, c) {
    console.log("函数内部this指向:", this);
    console.log("参数列表:", a, b, c);
  }
  let newFn = fn.myBind(obj, 10, 20);
```

```
  newFn(30);
</script>
```

上段代码中需要注意的有两点，第一点是利用apply函数将函数的this指向了传经来的context，第二点是将参数新传进来的参数与args拼接，因为我们调用newFn时，可能传进来新参数，所以需要将新老参数拼接上。

输出结果：

```
函数内部this指向: ▶ {name: '小猪课堂', age: 20}
参数列表: 10 20 30
>
```

上面的输出结果是和直接使用bind函数输出的结果是一样的。

3. 上面的代码还不够完善，如果我们将创建的新函数以构造函数的方式执行的话，this的执行和原生的bind不太一致。

代码如下：

```
<script>
  let obj = {
    name: "小猪课堂",
    age: 20
  }
  // 手写bind函数
  Function.prototype.myBind = function (context) {
    const _this = this; // 当前函数
    let args = Array.from(arguments).slice(1); // 将参数列表转化为数组,除去第一
    // 返回新函数
    return function () {
      // context 是传进来的this
      _this.apply(context, args.concat(Array.from(arguments))); // 利用apply
    }
  }
  // 声明一个函数
  function fn(a, b, c) {
    console.log("函数内部this指向:", this);
    console.log("参数列表:", a, b, c);
  }
  let newFn = fn.myBind(obj, 10, 20); // 调用封装的bind
  let newFn1 = fn.bind(obj, 10, 20); // 调用原生的bind

  new newFn("myBind构造函数");
  new newFn1("bind构造函数");
</script>
```

输出结果：

```
函数内部this指向: ▶{name: '小猪课堂', age: 20}
参数列表: 10 20 myBind构造函数
函数内部this指向: ▶fn {}
参数列表: 10 20 bind构造函数
```

上面的输出结果不一致，说明使用原生bind创建的新函数，如果使用构造函数的方式执行，那么函数内部的this执行会作为一个新的引用指向fn。

修改代码如下：

```
<script>
  let obj = {
    name: "小猪课堂",
    age: 20
  }
  // 手写bind函数
  Function.prototype.myBind = function (context) {
    const _this = this; // 当前函数
    let args = Array.from(arguments).slice(1); // 将参数列表转化为数组,除去第一
    // 返回新函数
    let fn = function () {
      // 如果被new调用，this应该是fn的实例
      return _this.apply(this instanceof fn ? this : (context || window), ar
    }
    // 维护fn的原型
    let temp = function () { }
    temp.prototype = _this.prototype;
    fn.prototype = new temp(); // new的过程继承temp原型
    return fn
  };
  // 声明一个函数
  function fn(a, b, c) {
    console.log("函数内部this指向:", this);
    console.log("参数列表:", a, b, c);
  }
  let newFn = fn.myBind(obj, 10, 20);
  let newFn1 = fn.bind(obj, 10, 20)

  new newFn("myBind构造函数");
  new newFn1("bind构造函数");
</script>
```

输出结果：

上段代码的输出结果是不是就和实际的bind函数输出结果一样了啊！想要理解上段代码，大家有必要去学习以下JS中new一个对象发生了什么，主要是下面几步：

- 创建一个新对象
- 将构造函数的this赋值给新对象
- 执行构造函数代码，给这个新的对象添加属性
- 返回新的对象

具体的new实现过程还需要大家自己去理解。

总结

想要实现bind函数，就必须要理解其中的原理，无非就是改变this指向的问题。其中唯一的难点就是如何实现构造函数执行的方式，也就是要明白js中new一个对象的时候发生了什么？