【前端代码】请手写一个apply和call方法?

前言

this 指向问题一直是一个老生常谈的问题了！我们对它可以说是又爱又恨，因为 this 指向常常没有按照我们的想法去指向谁，导致程序无缘出现许多 bug。所以我们常常直接强制改变程序中的 this 指向，我们常用的方法有 bind、apply 和 call，bind 与其它两个稍许不同，所以我们本篇文章专门讲解 call 和 apply 方法，并且手动模拟实现它们。

1.如何使用?

我们既然想要模拟实现 call 和 apply 两个方法，那么我们很有必要先了解它们的用法。它们两个的用法也比较简单，这里我就带大家简单复习一遍。

通过代码我们回顾一下。

代码如下：

```
<script>
  let obj = {
    name: '小猪课堂'
  }
  function say(age) {
    console.log("你好：", this.name, '我今年' + age + '岁了');
  }
  say(12);              // 你好：  我今年 12 岁了
  say.call(obj, 12);    // 你好： 小猪课堂 我今年 12 岁了
  say.apply(obj, [12]) // 你好： 小猪课堂 我今年 12 岁了
</script>
```

上段代码非常简单，我们直接调用 say 函数时，函数的 this 指向中是没有找到 name 属性的，我们通过 call 和 apply 方法调用时，将 this 指向了 obj，而 obj 内部有 name 属性，所以函数中，可以取到 name 属性的值。

看了代码之后我们再来看一下官网对这两个方法的解释，一下印象。

call 官网解释：

> call() 方法使用一个指定的 this 值和单独给出的一个或多个参数来调用一个函数。

apply 官网解释：

> apply() 方法调用一个具有给定 this 值的函数，以及以一个数组（或类数组对象）的形式提供的参数。

它们两个的基本上没什么区别，唯一的区别也只有一个，官网也给出了解释。

唯一的区别：

> call()方法的作用和 apply() 方法类似，区别就是 call()方法接受的是参数列表，而 apply()方法接受的是一个参数数组。

2.总结特点

知道了这两个方法怎么用之后，我们接下来总结它们的特点，为我们后面的模拟实现提供一个思路。

2.1 改变 this 指向

这一点毋庸置疑，我们使用这两个方法就是为了改变 this 指向的，这也是它们两个的共同特点。我们可以将函数原来的 this 指向以后改变后的 this 指向打印出来看看。

代码如下：

```
<script>
  let obj = {
    name: '小猪课堂'
  }
  function say() {
    console.log(this)
  }
  say();            // Window {window: Window, self: Window, document: docume
  say.call(obj);    // {name: '小猪课堂'}
  say.apply(obj)    // {name: '小猪课堂'}
</script>
```

输出结果：

```
▶ Window {window: Window, self: Window, document: document, name: '', location: Location, …}
▶ {name: '小猪课堂'}
▶ {name: '小猪课堂'}
```

很明显，直接调用函数 this 指向是指向的 window 全局，使用 call 和 apply 方法后 this 指向的是传入的对象 obj。

2.2 参数传递

call 和 apply 参数传递的第一个参数都是我们需要将 this 指向的对象，后面的参数则为函数正常应该接收的参数，只不过两个函数中这部分的写法不一致。

call 方法：

第一个参数为 this 指向对象，后面接收一个参数列表，为函数正常接收的参数，示例代码如下：

```
say.call(obj,12,32);
```

apply 方法：

apply 方法第一个参数也是接收的 this 指向的对象，只不过后面正常的参数采用数组的形式接收，示例代码如下：

```
say.apply(obj,[12,32]);
```

2.3 未指定 this 时

当我们调用 call 和 apply 两个方法时，未传入第一个参数或者传入的参数为 null 或者 undefind，这种情况下函数的 this 指向会指向哪里呢?

这种情况需要分为严格模式和非严格模式，js 严格模式下，如果未传入第一个参数，或者传入的参数为 null 或者 undefind，则函数 this 指向为 null 或者 undefind。在严格模式下则指向 window。

严格模式

代码如下：

```
say.call();           // undefined
say.call(null);       // undefined
say.call(undefined);  // undefined

say.apply()           // undefined
say.apply(null)       // null
say.apply(undefined)  // undefined
```

输出结果：

```
undefined
null
undefined
undefined
null
undefined
```

非严格模式

代码如下：

```
say.call();           // Window
say.call(null);       // Window
say.call(undefined);  // Window

say.apply()           // Window
```

```
say.apply(null)        // Window
say.apply(undefined)   // Window
```

输出结果：

```
▶ Window {window: Window, self: Window, document: document, name: '', location: Location, …}
▶ Window {window: Window, self: Window, document: document, name: '', location: Location, …}
▶ Window {window: Window, self: Window, document: document, name: '', location: Location, …}
▶ Window {window: Window, self: Window, document: document, name: '', location: Location, …}
▶ Window {window: Window, self: Window, document: document, name: '', location: Location, …}
▶ Window {window: Window, self: Window, document: document, name: '', location: Location, …}
```

2.4 自动封装对象

当我们 call 和 apply 方法第一个参数传输的不是对象类型时，那么 this 指向将会指向传入的值类型的包装对象，当然，除了 null 和 undefined 之外。

代码如下：

```
say.call(12);          // Number {12}
say.call('小猪课堂'); // String {'小猪课堂'}
say.call(true);        // Boolean {true}

say.apply(12);          // Number {12}
say.apply('小猪课堂'); // String {'小猪课堂'}
say.apply(true);        // Boolean {true}
```

输出结果：

```
▶ Number {12}
▶ String {'小猪课堂'}
▶ Boolean {true}
▶ Number {12}
▶ String {'小猪课堂'}
▶ Boolean {true}
>
```

3.实现 call 方法

我们知道了 call 方法和 apply 的用法以及它们有什么特点，那么接下来就需要针对这些特点一一想办法去实现它们，我们首先来实现 call 方法。

3.1 完成 this 指向

改变 this 指向是这两个方法的核心功能，只要我们搞明白了如何改变改变 this 指向，那就解决了一大半的问题。

回顾代码：

```
<script>
  let obj = {
    name: '小猪课堂'
  }
  function say(age) {
    console.log("你好：", this.name, '我今年' + age + '岁了');
  }
  say(12);              // 你好：  我今年 12 岁了
  say.call(obj, 12);    // 你好： 小猪课堂 我今年 12 岁了
  say.apply(obj, [12]) // 你好： 小猪课堂 我今年 12 岁了
</script>
```

解决思路：

我们的目标是将函数内部的 this 指向 obj。再学习 js 的时候，我们可能知道这么一句话，谁是调用者，this 指向就指向谁，那么我们是否可以试想一下，如果是 obj 调用函数 say 的话，那么函数内部 this 指向是不是就指向了 obj 呢?

思路代码：

```
<script>
  let obj = {
    name: '小猪课堂',
    say: function (age) {
      console.log("你好：", this.name, '我今年' + age + '岁了');
    }
  }
  obj.say(12); // 你好： 小猪课堂 我今年 12 岁了
</script>
```

上段代码中我们没有使用 call 或者 apply 方法，也将函数内部的 this 指向了 obj。

有了上面得思路后，我们就可以先实现一个简单得 call 方法了。

自定义 customCall 方法：

```
Function.prototype.customCall = function (context) {
  const fnKey = Symbol(); // 函数键名，使用 symbo 不会重复
  context[fnKey] = this; // 将函数赋值给对象中的 fnKey 属性
  const res = context[fnKey](); // 执行函数
  delete context[fnKey]; // 删除 context 对象中的该属性，避免越来越多
  return res // 返回结果
}
```

上段代码中有几点需要注意：

- 使用 symbol 是为了避免属性名重复，因为我们 obj 对象中可能已经有 say 属性名的存在了。
- 将函数赋值给属性的时候，我们直接使用了 this，因为这个时候的 this 本来就是原来的函数，比如我们调用 say.call()，这个时候 call 方法中的额 this 其实就是 say。
- 执行完函数后需要及时删除掉，因为下次我们调用 customCall 方法是还会生成新的属性。

3.2 this 指向全局

当 call 方法传入的第一个参数为 null 或者 undefind 时，我们需要将 this 指向到全局，修改一下代码。

代码如下：

```
Function.prototype.customCall = function (context) {
  if (context === null || context === undefined) {
    context = globalThis; // this 指向全局
  }
  const fnKey = Symbol(); // 函数键名，使用 symbo 不会重复
  context[fnKey] = this; // 将函数赋值给对象中的 fnKey 属性
  const res = context[fnKey](); // 执行函数
  delete context[fnKey]; // 删除 context 对象中的该属性，避免越来越多
  return res // 返回结果
}
```

3.3 传入参数

call 方法可以传入很多参数的，我们也需要实现接收参数，修改代码。

代码如下：

```
Function.prototype.customCall = function (context, ...args) {
  if (context === null || context === undefined) {
    context = globalThis; // this 指向全局
  }
  const fnKey = Symbol(); // 函数键名，使用 symbo 不会重复
  context[fnKey] = this; // 将函数赋值给对象中的 fnKey 属性
  const res = context[fnKey](...args); // 执行函数，...args 解构参数
  delete context[fnKey]; // 删除 context 对象中的该属性，避免越来越多
  return res // 返回结果
}
```

这里就添加了一个...args，...args 可以将我们的可变参数解构为一个数组。

3.4 值类型包装对象（最终版）

当传入的 context 是值类型时，我们需要将它改编为对应的包装对象，修改代码、

代码如下：

```
Function.prototype.customCall = function (context, ...args) {
  if (context === null || context === undefined) {
    context = globalThis; // this 指向全局
  }

  if (typeof context !== 'object') {
    context = new Object(context); // 值类型变为它的包装对象
  }
```

```
  const fnKey = Symbol(); // 函数键名，使用 symbo 不会重复
  context[fnKey] = this; // 将函数赋值给对象中的 fnKey 属性
  const res = context[fnKey](...args); // 执行函数，...args 解构参数
  delete context[fnKey]; // 删除 context 对象中的该属性，避免越来越多
  return res // 返回结果
}
```

4.实现 apply 方法

我们实现了 call 方法后，apply 方法那就手到擒来了，因为这两个方法就一个区别，接收的正餐参数的格式不一样而已，修改一个 customCall 方法代码即可。

代码如下：

```
Function.prototype.customApply = function (context, args) {
  if (context === null || context === undefined) {
    context = globalThis; // this 指向全局
  }

  if (typeof context !== 'object') {
    context = new Object(context); // 值类型变为它的包装对象
  }
  const fnKey = Symbol(); // 函数键名，使用 symbo 不会重复
  context[fnKey] = this; // 将函数赋值给对象中的 fnKey 属性
  const res = context[fnKey](...args); // 执行函数，...args 解构参数
  delete context[fnKey]; // 删除 context 对象中的该属性，避免越来越多
  return res // 返回结果
}
```

上段代码中我们只改了一个地方，就是把接收的...args 参数改为了 args。

5.测试

代码编写完了，接下来测试一下是否满足我们的需求。

代码如下：

```
<script>
  let obj = {
    name: '小猪课堂',
  }
  function say(age) {
    console.log("你好：", this.name, '我今年' + age + '岁了');
  }
  // 自定义 call 和 apply 方法
  Function.prototype.customCall = function (context, ...args) {
    if (context === null || context === undefined) {
      context = globalThis; // this 指向全局
    }

    if (typeof context !== 'object') {
```

```
      context = new Object(context); // 值类型变为它的包装对象
    }
    const fnKey = Symbol(); // 函数键名，使用 symbo 不会重复
    context[fnKey] = this; // 将函数赋值给对象中的 fnKey 属性
    const res = context[fnKey](...args); // 执行函数，...args 解构参数
    delete context[fnKey]; // 删除 context 对象中的该属性，避免越来越多
    return res // 返回结果
  }

  Function.prototype.customApply = function (context, args) {
    if (context === null || context === undefined) {
      context = globalThis; // this 指向全局
    }

    if (typeof context !== 'object') {
      context = new Object(context); // 值类型变为它的包装对象
    }
    const fnKey = Symbol(); // 函数键名，使用 symbo 不会重复
    context[fnKey] = this; // 将函数赋值给对象中的 fnKey 属性
    const res = context[fnKey](...args); // 执行函数，...args 解构参数
    delete context[fnKey]; // 删除 context 对象中的该属性，避免越来越多
    return res // 返回结果
  }
  say.customCall(obj, 32);     // 你好： 小猪课堂 我今年 32 岁了
  say.customCall();            // 你好：  我今年 undefined 岁了

  say.customApply(obj, [32]); // 你好： 小猪课堂 我今年 32 岁了
  say.customApply();           // 你好：  我今年 undefined 岁了
</script>
```

测试出来这两个方法基本是满足我们要求的。

总结

实现 call 和 apply 方法实际很简单，总也也就 10 行代码左右，只要我们知道了其中的原理，这段代码就是信手拈来。