this指向

1. 常见的this指向
2. call作用
3. apply作用
4. bind作用
5. 模拟call bind apply

this指向

```
//eg:1
var length = 10;
function fn() {
    console.log(this.length);
}
var obj = {
    length: 5,
    method: function () {
      fn();
      arguments[0]();
    }
}
obj.method(fn, 1);
```

```
//eg:2
var b = 1;
function outer() {
    var b = 2
    function inner() {
        b++;
        var b = 3;
        console.log(b)
    }
    inner();
}
outer();
```

Bind 方法

我们用 Bind() 来实现在指明函数内部 this 指向的情况下去调用该函数, 换句话说, bind() 允许我们非常简单的在函数或者方法被调用时绑定 this 到指定对象上.

当我们在一个方法中用到了 this, 而这个方法调用于一个接收器对象, 我们会需要使用到 bind() 方法; 在这种情况下, 由于 this 不一定完全如我们所期待的绑定在目标对象上, 程序有时便会出错;

Bind 允许我们明确指定方法中的 this 指向

当以下按钮被点击的时候, 文本输入框会被随机填入一个名字.

```
// <button>Get Random Person</button>
// <input type="text">

var user = {
  data:[
    {name:"T. Woods", age:37},
    {name:"P. Mickelson", age:43}
  ],
  clickHandler:function(event) {
    var randomNum = ((Math.random () * 2 | 0) + 1) - 1; // random number
between 0 and 1

    // 从 data 数组中随机选取一个名字填入 input 框内
```

```
    $("input").val(this.data[randomNum].name + " " +
this.data[randomNum].age);
  }
}

// 给点击事件添加一个事件处理器
$("button").click(user.clickHandler);
```

当你点击按钮时, 会发现一个报错信息: 因为 `clickHandler()` 方法中的 `this` 绑定的是按钮 `HTML` 内容的上下文, 因为这才是 `clickHandler` 方法的执行时的调用对象.

在 `JavaScript` 中这种问题比较常见, `JavaScript` 框架中例如 `Backbone.js`, `jQuery` 都自动为我们做好了绑定的工作, 所以在使用时 `this` 总是可以绑定到我们所期望的那个对象上.

为了解决之前例子中存在的问题, 我们利用 `bind()` 方法将 `$("button").click(user.clickHandler);` 换成以下形式:

```
$("button").click(user.clickHandler.bind(user));
```

再考虑另一个方法来修复 `this` 的值: 你可以给 `click()` 方法传递一个匿名回调函数, `jQuery` 会将匿名函数的 this 绑定到按钮对象上.

> bind() 函数在 ECMA-262 第五版才被加入；它可能无法在所有浏览器上运行。你可以部份地在脚本开头加入以下代码，就能使它运作，让不支持的浏览器也能使用 bind() 功能。 - MDN

```
if (!Function.prototype.bind) {
  Function.prototype.bind = function(oThis) {
    if (typeof this !== "function") {
      // closest thing possible to the ECMAScript 5
      // internal IsCallable function
      throw new TypeError("Function.prototype.bind - what is trying to be
bound is not callable");
    }

    var aArgs = Array.prototype.slice.call(arguments, 1),
        fToBind = this, // 此处的 this 指向目标函数
        fNOP = function() {},
        fBound = function() {
          return fToBind.apply(this instanceof fNOP
            ? this // 此处 this 为 调用 new obj() 时所生成的 obj 本身
            : oThis || this, // 若 oThis 无效则将 fBound 绑定到 this
```

```
            // 将通过 bind 传递的参数和调用时传递的参数进行合并，并作为最终的参数传递
            aArgs.concat(Array.prototype.slice.call(arguments)));
        };

    // 将目标函数的原型对象拷贝到新函数中，因为目标函数有可能被当作构造函数使用
    fNOP.prototype = this.prototype;
    fBound.prototype = new fNOP();

    return fBound;
  };
}
```

继续之前的例子，如果我们将包含 this 的方法赋值给一个变量，那么 this 的指向也会绑定到另一个对象上，如下所示：

```
// 全局变量 data
var data = [
    {name:"Samantha", age:12},
    {name:"Alexis", age:14}
]

var user = {
    // 局部变量 data
    data:[
        {name:"T. Woods", age:37},
        {name:"P. Mickelson", age:43}
    ],
    showData:function(event) {
        var randomNum = ((Math.random () * 2 | 0) + 1) - 1; // random
number between 0 and 1

        console.log(this.data[randomNum].name + " " +
this.data[randomNum].age);
    }

}

// 将 user 对象的 showData 方法赋值给一个变量
var showDataVar = user.showData;

showDataVar(); // Samantha 12 (来自全局变量数组而非局部变量数组)
```

当我们执行 showDataVar() 函数时, 输出到 console 的数值来自全局 data 数组, 而不是 user 对象. 这是因为 showDataVar() 函数是被当做一个全局函数执行的, 所以在函数内部 this 被绑定位全局对象, 即浏览器中的 window 对象.

来, 我们用 bind 方法来修复这个 bug.

```
// Bind the showData method to the user object
var showDataVar = user.showData.bind(user);
```

Bind 方法允许我们实现函数借用

在 JavaScript 中, 我们可以传递函数, 返回函数, 借用他们等等, 而 bind() 方法使函数借用变得极其简单. 以下为一个函数借用的例子:

```
 // cars 对象
var cars = {
    data:[
        {name:"Honda Accord", age:14},
        {name:"Tesla Model S", age:2}
    ]

}

// 我们从之前定义的 user 对象借用 showData 方法
// 这里我们将 user.showData 方法绑定到刚刚新建的 cars 对象上
cars.showData = user.showData.bind(cars);
cars.showData(); // Honda Accord 14
```

这里存在一个问题, 当我们在 cars 对象上添加一个新方法(showData)时我们可能不想只是简单的借用一个函数那样, 因为 cars 本身可能已经有一个方法或者属性叫做 showData 了, 我们不想意外的将这个方法覆盖了. 正如在之后的 *Apply 和 Call 方法* 章节我们会介绍, 借用函数的最佳实践应该是使用 Apply 或者 Call 方法.

Bind 方法允许我们柯里化一个函数

> 柯里化的概念很简单, 只传递给函数一部分参数来调用它, 让它返回一个函数去处理剩下的参数. 你可以一次性地调用 curry 函数, 也可以每次只传一个参数分多次调用, 以下为一个简单的示例. - **JS 函数是编程指南 第 4 章: 柯里化（curry）**

```
var add = function(x) {
  return function(y) {
    return x + y;
  };
};

var increment = add(1);
var addTen = add(10);

increment(2);
// 3

addTen(2);
// 12
```

现在, 我们使用 bind() 方法来实现函数的柯里化. 我们首先定义一个接收三个参数的 greet() 函数:

```
function greet(gender, age, name) {
    // if a male, use Mr., else use Ms.
    var salutation = gender === "male" ? "Mr. " : "Ms. ";

    if (age > 25) {
        return "Hello, " + salutation + name + ".";
    }
    else {
        return "Hey, " + name + ".";
    }
}
```

接着我们使用 bind() 方法柯里化 greet() 方法. bind() 接收的第一个参数指定了 this 的值:

```
 // 在 greet 函数中我们可以传递 null, 因为函数中并未使用到 this 关键字
var greetAnAdultMale = greet.bind (null, "male", 45);

greetAnAdultMale("John Hartlove"); // "Hello, Mr. John Hartlove."

var greetAYoungster = greet.bind(null, "", 16);
greetAYoungster("Alex"); // "Hey, Alex."
greetAYoungster("Emma Waterloo"); // "Hey, Emma Waterloo."
```

当我们用 bind() 实现柯里化时, greet() 函数参数中除了最后一个参数都被预定义好了, 所以当我们调用柯里化后的新函数时只需要指定最后一位参数.

所以小结一下, bind() 方法允许我们明确指定对象方法中的 this 指向, 我们可以借用, 复制一个方法或者将方法赋值为一个可作为函数执行的变量. 我们以可以借用 bind 实现函数柯里化.

JavaScript 中的 Apply 和 Call 方法

作为 JavaScript 中最常用的两个函数方法, apply 和 call 允许我们借用函数以及在函数调用中指定 this 指向. 除此外, apply 函数允许我们在执行函数时传入一个参数数组, 以此使函数在执行可变参数的函数时可以将每个参数单独的传入函数并得到处理.

使用 apply 或者 call 设置 this

当我们使用 apply 或者 call 时, 传入的第一个参数为目标函数中 this 指向的对象, 以下为一个简单的例子:

```
// 全局变量
var avgScore = "global avgScore";

// 全局函数
function avg(arrayOfScores) {
    // 分数相加并返回结果
    var sumOfScores = arrayOfScores.reduce(function(prev, cur, index,
array) {
        return prev + cur;
    });

    // 这里的 "this" 会被绑定到全局对象上, 除非使用 Call 或者 Apply 明确指定 this
的指向
    this.avgScore = sumOfScores / arrayOfScores.length;
}

var gameController = {
    scores  :[20, 34, 55, 46, 77],
    avgScore:null
}

// 调用 avg 函数, this 指向 window 对象
avg(gameController.scores);
// 证明 avgScore 已经被设置为 window 对象的属性
```

```
console.log(window.avgScore); // 46.4
console.log(gameController.avgScore); // null

// 重置全局变量
avgScore = "global avgScore";

// 使用 call() 方法明确将 "this" 绑定到 gameController 对象
avg.call(gameController, gameController.scores);

console.log(window.avgScore); // 全局变量 avgScore 的值
console.log(gameController.avgScore); // 46.4
```

以上例子中 call() 中的第一个参数明确了 this 的指向, 第二参数被传递给了 avg() 函数.

apply 和 call 的用法几乎相同, 唯一的差别在于当函数需要传递多个变量时, apply 可以接受一个数组作为参数输入, call 则是接受一系列的单独变量.

在灰调函数中用 call 或者 apply 设置 this

以下为一个例子, 这种做法允许我们在执行 callback 函数时能够明确 其内部的 this 指向

```
// 定义一个方法
var clientData = {
    id: 094545,
    fullName: "Not Set",
    // clientData 对象中的一个方法
    setUserName: function (firstName, lastName)  {
        this.fullName = firstName + " " + lastName;
    }
}

function getUserInput(firstName, lastName, callback, callbackObj) {
    // 使用 apply 方法将 "this" 绑定到 callbackObj 对象
    callback.apply(callbackObj, [firstName, lastName]);
}
```

如下样例中传递给 callback 函数 中的参数将会在 clientData 对象中被设置/更新.

```
getUserInput("Barack", "Obama", clientData.setUserName, clientData);
console.log(clientData.fullName); // Barack Obama
```

使用 Apply 或者 Call 借用函数(必备知识)

相比 bind 方法, 我们使用 apply 或者 call 方法实现函数借用能够有很大的施展空间. 接下来我们考虑从 Array 中借用方法的问题, 让我们定义一个**类数组对象(array-like object)**然后从数组中借用方法来处理我们定义的这个对象, 不过在这之前请记住我们要操作的是一个对象, 而不是数组;

```
// An array-like object: note the non-negative integers used as keys
var anArrayLikeObj = {0:"Martin", 1:78, 2:67, 3:["Letta", "Marieta",
"Pauline"], length:4 };
```

接下来我们可以这样使用数组的原生方法:

```
 // Make a quick copy and save the results in a real array:
// First parameter sets the "this" value
var newArray = Array.prototype.slice.call(anArrayLikeObj, 0);

console.log(newArray); // ["Martin", 78, 67, Array[3]]

// Search for "Martin" in the array-like object
console.log(Array.prototype.indexOf.call(anArrayLikeObj, "Martin") === -1
? false : true); // true

// Try using an Array method without the call () or apply ()
console.log(anArrayLikeObj.indexOf("Martin") === -1 ? false : true); //
Error: Object has no method 'indexOf'

// Reverse the object:
console.log(Array.prototype.reverse.call(anArrayLikeObj));
// {0: Array[3], 1: 67, 2: 78, 3: "Martin", length: 4}

// Sweet. We can pop too:
console.log(Array.prototype.pop.call(anArrayLikeObj));
console.log(anArrayLikeObj); // {0: Array[3], 1: 67, 2: 78, length: 3}

// What about push?
console.log(Array.prototype.push.call(anArrayLikeObj, "Jackie"));
console.log(anArrayLikeObj); // {0: Array[3], 1: 67, 2: 78, 3: "Jackie",
length: 4}
```

这样的操作使得我们定义的对象既保留有所有对象的属性, 同时也能够在对象上使用数组方法.

arguments 对象是所有 JavaScript 函数中的一个类数组对象, 因此 call() 和 apply() 的一个最常用的用法是从 arguments 中提取参数并将其传递给一个函数.

以下为 Ember.js 源码中的一部分, 加上了我的一些注释:

```
function transitionTo(name) {
    // 因为 arguments 是一个类数组对象, 所以我们可以使用 slice()来处理它
    // 参数 "1" 表示我们返回一个从下标为1到结尾元素的数组
    var args = Array.prototype.slice.call(arguments, 1);

    // 添加该行代码用于查看 args 的值
    console.log(args);

    // 注释本例不需要使用到的代码
    //doTransition(this, name, this.updateURL, args);
}

// 使用案例
transitionTo("contact", "Today", "20"); // ["Today", "20"]
```

以上例子中, args 变量是一个真正的数组. 从以上案例中我们可以写一个得到快速得到传递给函数的所有参数(以数组形式)的函数:

```
function doSomething() {
    var args = Array.prototype.slice.call(arguments);
    console.log(args);
}

doSomething("Water", "Salt", "Glue"); // ["Water", "Salt", "Glue"]
```

考虑到字符串是不可变的, 如果使用 apply 或者 call 方法借用字符串的方法, 不可变的数组操作对他们来说才是有效的, 所以你不能使用类似 reverse 或者 pop 等等这类的方法. 除此外, 我们也可以用他们借用我们自定义的方法.

```
var gameController = {
    scores  :[20, 34, 55, 46, 77],
    avgScore:null,
    players :[
        {name:"Tommy", playerID:987, age:23},
        {name:"Pau", playerID:87, age:33}
    ]
```

```
}

var appController = {
    scores  :[900, 845, 809, 950],
    avgScore:null,
    avg     :function() {
        var sumOfScores = this.scores.reduce(function(prev, cur, index,
array) {
            return prev + cur;
        });
        this.avgScore = sumOfScores / this.scores.length;
    }
}

// Note that we are using the apply() method, so the 2nd argument has to
be an array
appController.avg.apply(gameController);
console.log(gameController.avgScore); // 46.4

// appController.avgScore is still null; it was not updated, only
gameController.avgScore was updated
console.log(appController.avgScore); // null
```

这个例子非常简单, 我们定义的 gameController 对象借用了 appController 对象的 avg() 方法. 你也许会想, 如果我们借用的函数定义发生了变化, 那么我们的代码会发生什么变化. 借用(复制后)的函数也会变化么, 还是说他在完整复制后已经和原始的方法切断了联系? 让我们用下面这个小例子来说明这个问题:

```
appController.maxNum = function() {
    this.avgScore = Math.max.apply(null, this.scores);
}

appController.maxNum.apply(gameController, gameController.scores);
console.log(gameController.avgScore); // 77
```

正如我们所期望的那样, 如果我们修改原始的方法, 这样的变化会在借用实例的方法上体现出来. 我们总是希望如此, 因为我们从来不希望完整的复制一个方法, 我们只是想简单的借用一下.

使用 apply() 执行参数可变的函数

关于 Apply, Call 和 Bind 方法的多功能性和实用性, 我们将讨论一下Apply方法的一个很简单的功能: 使用参数数组执行函数.

Math.max() 方法是 JavaScript 中一个常见的参数可变函数:

```
console.log(Math.max(23, 11, 34, 56)); // 56
```

但如果我们有一个数组要传递给 Math.max(), 是不能这样做的:

```
var allNumbers = [23, 11, 34, 56];
console.log(Math.max(allNumbers)); // NaN
```

使用 apply 我们可以像下面这样传递数组:

```
var allNumbers = [23, 11, 34, 56];
console.log(Math.max.apply(null, allNumbers)); // 56
```

正如之前讨论, apply() 的第一个参数用于设置 this 的指向, 但是 Math.max() 并未使用到 this, 所以我们传递 null 给他.

为了更进一步解释 apply() 在 参数可变函数上的能力, 我们自定义了一个参数可变函数:

```
var students = ["Peter Alexander", "Michael Woodruff", "Judy Archer",
"Malcolm Khan"];

// 不定义参数, 因为我们可以传递任意多个参数进入该函数
function welcomeStudents() {
    var args = Array.prototype.slice.call(arguments);

    var lastItem = args.pop();
    console.log("Welcome " + args.join (", ") + ", and " + lastItem +
".");
}

welcomeStudents.apply(null, students);
// Welcome Peter Alexander, Michael Woodruff, Judy Archer, and Malcolm
Khan.
```

区别与注意事项

三个函数存在的区别, 用一句话来说的话就是: bind是返回对应函数, 便于稍后调用; apply, call则是立即调用. 除此外, 在 ES6 的箭头函数下, call 和 apply 的失效, 对于箭头函数来说:

- 函数体内的 this 对象, 就是定义时所在的对象, 而不是使用时所在的对象;
- 不可以当作构造函数, 也就是说不可以使用 new 命令, 否则会抛出一个错误;
- 不可以使用 arguments 对象, 该对象在函数体内不存在. 如果要用, 可以用 Rest 参数代替;
- 不可以使用 yield 命令, 因此箭头函数不能用作 Generator 函数;

更多关于箭头函数的介绍在这里就不做过多介绍了, 详情可以查看 Arrow functions.

es5实现call

```
/*封装call*/
Function.prototype.selfCall = function selfCall(context, ...args) {
  context || (context = window);

  if (typeof this !=='function') throw new TypeError('selfCall is called
must be a function');
  const fnc = this;
  const caller = Symbol('caller');
  context[caller] = fnc;
  let res = context[caller](...args);
  delete context[caller];
  return res;
};


/*
* 看看call怎么用
* 1. call是Function原型上面的一个方法
* 2. call 函数参数 xxx.call(上下文,参数(可以有多个参数))   //xxx是一个函数
* 3. 相当于执行xxx函数 只不过 传递了上下文对象 this
* */

function fn1 (cook){
  console.log(this,cook);//window / node对象
}

// fn1.call({obj:{}},'爆炒鱿鱼');
fn1.selfCall({obj:'lala'});
```

es5实现bind

```
/*
* 实现bind函数
* */

const selfBind = function (bindTarget, ...args1) {
  /*不是函数调用抛出错误*/
  /*封装函数 如果没有绑定this则绑定 否则直接返回该函数*/
  /*绑定原型*/
  /*原型链绑定*/
  /*返回该函数*/
  /*2:*/
  const func = this;
  const boundFn = function (...args2) {
    const args = [...args1, ...args2];
    if (new.target) {
      /*判断this必须是对象或者方法 */
      let res = func.apply(this, args);
      //new之后点出方法或者属性
      if ((typeof res === 'function' || typeof res === 'object') && res
!== null) return res;
      return this;
    } else {
      func.apply(bindTarget, args);
    }
    return func;
  };
  /*原型赋值*/
  this.prototype && (boundFn.prototype = Object.create(this.prototype));
  /*原型链赋值*/
  /*取出this中的所有原型链对象属性*/
  let des = Object.getOwnPropertyDescriptors(func);
  console.log('des.length',des.length);
  Object.defineProperties(boundFn,{
    length: des.length,
    name:Object.assign(des,{
      value:`bound${des.name.value}`
    })
  });
  return boundFn;
};
```

```
/*
* bind函数的使用
* */

let obj = {
  getName() {
    console.info('Obj name')
  },
  selfName: 'Obj'
};

let obj2 = {
  getAge() {
    console.info('Obj2 age')
  },
  selfName: 'Obj2'
};

function aaa(...args) {
  console.log(args);
  // console.log(this, index, aa);
  console.log(this);
}

aaa.prototype.name = 'zhangsan';
Object.prototype.selfBind = selfBind;

let newAaa = aaa.selfBind(obj2, '张三', '李四', 123);
console.log(new newAaa(1, 'aa').name);
console.log(newAaa.prototype.name);
//获取该对象或者方法自身属性的描述
console.log(Object.getOwnPropertyDescriptors(newAaa));
// newAaa();
// new newAaa();
// let aa = new newAaa('王五');
// console.log(aa);
// aa.selfBind(obj2)();


//
// let newAaa = aaa.bind(obj, 'name');
// newAaa('111').getName();
// console.log(aaa.__proto__ === newAaa.__proto__);

/*
```

```
* 1:改变this指向
* 2:是一个'函数'调用bind
* 3:返回一个新的函数 但不执行 只有调用的时候才执行
* 4:调用bind传入的参数  和真正调用函数的参数 有一个参数融合
* 5:new 调用和普通调用
* 6:原型属性是一样的
* 7:原型链属性一致
* */
console.log(Function);
```

总结

经过上面的叙述, Call, Apply 和 Bind 在设置 this 指向, 声称与执行参数可变函数以及函数借用方面的强大之处已经非常明显. 作为一名 JavaScript 开发者, 你一定会经常见到这种用法, 或者在开发中尝试使用他. 请确保你已经很好的了解了如上所述的概念与用法.