目标

1. Javascript高级篇之深入剖析闭包
2. Javascript高级篇之深入全面讲解this各种应用场景
3. Javascript高级篇之深入剖析面向对象编程

深入剖析闭包及其作用

什么是闭包

MDN 上对于闭包的定义是这样的：

一个函数和对其周围状态（**lexical environment，词法环境**）的引用捆绑在一起（或者说函数被引用包围），这样的组合就是**闭包**（**closure**）。也就是说，闭包让你可以在一个内层函数中访问到其外层函数的作用域。在 JavaScript 中，每当创建一个函数，闭包就会在函数创建的同时被创建出来。

关键在下面两点：

1. 是一个函数
2. 可以访问到另外一个函数作用域中的变量

对于闭包有以下三个特性：

1. 闭包可以访问当前函数以外的变量

```
function init() {
    var name = '名字'; // name 是一个被 init 创建的局部变量
    function displayName() {
        // displayName() 是内部函数，一个闭包
        alert(name); // 使用了父函数中声明的变量
    }
    return displayName();
}
init();
```

2. 即使外部函数已经返回，闭包仍能访问外部函数定义的变量

```
function init() {
    var name = '名字'; // name 是一个被 init 创建的局部变量
    function displayName() {
        // displayName() 是内部函数，一个闭包
        alert(name); // 使用了父函数中声明的变量
    }
    return displayName();
}
init();
init();
```

3. 闭包可以更新外部函数的值

```
function updateName() {
    var name = 'TT';
    function getName(value) {
        name = value;
        console.log(name);
    }
    return getName; //外部函数返回
}
var sayName = updateName();
sayName('Thomas'); //Thomas
sayName('Jack'); //Jack
```

从作用域链理解闭包

首先我们先来分析一个例子

```
var scope = '全局作用域';
function localScope() {
    var scope = '本地作用域';
    function f() {
        return scope;
    }
    return f;
}

var foo = localScope(); // foo指向函数f
foo(); // 调用函数f()
```

执行上下文

执行上下文是当前 JavaScript 代码被解析和执行时所在环境的抽象概念。

执行上下文总共有三种类型

- **全局执行上下文**：只有一个，浏览器中的全局对象就是 window 对象，`this` 指向这个全局对象。
- **函数执行上下文**：存在无数个，只有在函数被调用的时候才会被创建，每次调用函数都会创建一个新的执行上下文。
- **Eval 函数执行上下文**： 指的是运行在 `eval` 函数中的代码，很少用而且不建议使用。

因为JS引擎创建了很多的执行上下文，所以JS引擎创建了执行上下文**栈**（Execution context stack，ECS）来**管理**执行上下文。当 JavaScript 初始化的时候会向执行上下文栈压入一个**全局**执行上下文，我们用 globalContext 表示它，并且只有当整个应用程序结束的时候，执行栈才会被清空，所以程序结束之前， 执行栈最底部永远有个 globalContext。

```
ECStack = [     // 使用数组模拟栈
    globalContext
];
```

执行过程分析

1. 进入全局代码，创建全局执行上下文，全局执行上下文**压入执行上下文栈**
2. 全局执行**上下文初始化**
3. 执行 localScope函数，创建 localScope函数执行上下文，localScope 执行上下文被压入执行上下文栈

4. localScope执行**上下文初始化**，创建变量对象、作用域链、this等
5. localScope函数执行完毕，localScope执行上下文从执行上下文栈中弹出
6. 执行 f 函数，创建 f 函数执行上下文，f 执行上下文被压入执行上下文栈
7. f 执行**上下文初始化**，创建变量对象、作用域链、this等
8. f 函数执行完毕，f 函数上下文从执行上下文栈中弹出

那么**问题**来了， 函数f 执行的时候，localScope函数上下文已经被销毁了，那函数f是如何获取到scope变量的呢?

其实函数f 执行上下文维护了一个作用域链，会指向指向`localScope`作用域，作用域链是一个数组，结构如下。

```
fContext = {
    Scope: [AO, localScopeContext.AO, globalContext.VO],
}
```

在函数上下文中，用活动对象(activation object, **AO**)来表示变量对象。

- 1、变量对象（**VO**）是规范上或者是JS引擎上实现的，并不能在JS环境中直接访问。
- 2、当进入到一个执行上下文后，这个变量对象才会被**激活**，所以叫活动对象（**AO**），这时候活动对象上的各种属性才能被访问。

例如，下面代码

```
function foo(a) {
  var b = 2;
  function c() {}
  var d = function() {};

  b = 3;
}

foo(1);
```

这个时候的AO是

```
AO = {
    arguments: {
        0: 1,
        length: 1
    },
    a: 1,
    b: undefined,
    c: reference to function c(){},
    d: undefined
}
```

所以指向关系是当前作用域 --> `localScope`作用域--> 全局作用域，即使 localscopeContext 被销毁了，但是 JavaScript 依然会让 localScopeContext.AO（活动对象） 活在内存中，f 函数依然可以通过 f 函数的作用域链找到它，这就是闭包实现的**关键**。

闭包的作用和问题

闭包的作用

1. 隐藏变量，避免全局污染
2. 可以读取函数内部的变量

用闭包模拟私有方法的例子

```
var Counter = (function () {
    var privateCounter = 0;
    function changeBy(val) {
        privateCounter += val;
    }
    return {
        increment: function () {
            changeBy(1);
        },
        decrement: function () {
            changeBy(-1);
        },
        value: function () {
            return privateCounter;
        },
    };
})();

console.log(Counter.value()); /* logs 0 */
Counter.increment();
Counter.increment();
console.log(Counter.value()); /* logs 2 */
Counter.decrement();
console.log(Counter.value()); /* logs 1 */
```

闭包循环的例子

没有使用闭包之前

```
var data = [];

for (var i = 0; i < 3; i++) {
  data[i] = function () {
    console.log(i);
  };
}

data[0]();  // 3
data[1]();  // 3
data[2]();  // 3
```

使用闭包之后

```
for (var i = 0; i < 3; i++) {
    (function(num) {
        setTimeout(function() {
            console.log(num);
        }, 1000);
    })(i);
}
// 0
// 1
// 2
```

闭包的问题

这里简单说一下，为什么使用闭包时变量不会被垃圾回收机制收销毁呢，这里需要了解一下JS垃圾回收机制；

JS规定在一个函数作用域内，程序执行完以后变量就会被销毁，这样可节省内存；使用闭包时，按照作用域链的特点，闭包（函数）外面的变量不会被销毁，因为函数会一直被调用，所以一直存在，如果闭包使用过多会造成内存销毁

例子

```
function getData() {
    var buf = new Array(1000).join('*');
    var index = 0;
    return function () {
        index++;
        if (index < buf.length) {
            return buf[index - 1];
        } else {
            buf = null; // 不再使用buf, 手动清除引用。
            return '';
        }
    };
}
var data = getData();
var next = data();
while (next !== '') {
    // process data()
    next = data();
    console.log(next);
}
```

getData()返回一个函数，就是说返回了一个闭包。闭包引用了getData()函数中的局部变量buf。var data = getData()执行完后，getData()的局部变量buf不会被释放。这是因为data变量引用了getData()返回的闭包，而该闭包又引用了变量buf，所以javascript引擎不会回收buf内存。

全面讲解this各种应用场景

与其他语言相比，**函数的 `this` 关键字**在 JavaScript 中的表现略有不同，此外，在严格模式和非严格模式之间也会有一些差别。在绝大多数情况下，函数的调用方式决定了 `this` 的值（运行时绑定）。`this` 不能在执行期间被赋值，并且在每次函数被调用时 `this` 的值也可能会不同。ES5 引入了 bind 方法来设置函数的 `this` 值，而不用考虑函数如何被调用的。ES2015 引入了箭头函数，箭头函数不提供自身的 this 绑定（`this` 的值将保持为闭合词法上下文的值）。

1. 全局上下文

无论是否在严格模式下，在全局执行环境中（在任何函数体外部）`this` 都指向全局对象。

```
// 在浏览器中，window 对象同时也是全局对象：
console.log(this === window); // true

a = 37;
console.log(window.a); // 37

this.b = 'ABC';
console.log(window.b); // "ABC"
console.log(b); // "ABC"
```

2. 函数上下文

在函数内部，`this`的值取决于函数被调用的方式。

不在严格模式下，且 `this` 的值不是由该调用设置的，所以 `this` 的值默认指向全局对象，浏览器中就是 `window`。

```
function f1(){
  return this;
}
//在浏览器中：
f1() === window;   //在浏览器中，全局对象是window

//在Node中：
f1() === globalThis;
```

然而，在严格模式下，如果进入执行环境时没有设置 `this` 的值，`this` 会保持为 `undefined`，如下

```
function f2(){
  "use strict"; // 这里是严格模式
  return this;
}

f2() === undefined; // true
```

3. 类上下文

`this` 在 类 中的表现与在函数中类似，因为类本质上也是函数，但也有一些区别和注意事项。

在类的构造函数中，`this` 是一个常规对象。类中所有非静态的方法都会被添加到 `this` 的原型中：

```
class Example {
  constructor() {
    const proto = Object.getPrototypeOf(this);
    console.log(Object.getOwnPropertyNames(proto));
  }
  first(){}
  second(){}
  static third(){}
}

new Example(); // ['constructor', 'first', 'second']
```

4.箭头函数中的this

在全局代码中，它将被设置为全局对象：

```
var globalObject = this;
var foo = (() => this);
console.log(foo() === globalObject); // true
```

这同样适用于在其他函数内创建的箭头函数：这些箭头函数的 `this` 被设置为封闭的词法环境的。

```
// 创建一个含有bar方法的obj对象，
// bar返回一个函数，
// 这个函数返回this，
// 这个返回的函数是以箭头函数创建的，
// 所以它的this被永久绑定到了它外层函数的this。
// bar的值可以在调用中设置，这反过来又设置了返回函数的值。
var obj = {
  bar: function() {
    var x = (() => this);
    return x;
  }
};

// 作为obj对象的一个方法来调用bar，把它的this绑定到obj。
// 将返回的函数的引用赋值给fn。
var fn = obj.bar();

// 直接调用fn而不设置this，
// 通常(即不使用箭头函数的情况)默认为全局对象
// 若在严格模式则为undefined
console.log(fn() === obj); // true

// 但是注意，如果你只是引用obj的方法，
// 而没有调用它
var fn2 = obj.bar;
// 那么调用箭头函数后，this指向window，因为它从 bar 继承了this。
console.log(fn2()() == window); // true
```

5. 作为对象的方法

当函数作为对象里的方法被调用时，`this` 被设置为调用该函数的对象。

下面的例子中，当 `o.f()` 被调用时，函数内的 `this` 将绑定到 `o` 对象。

```
var o = {
  prop: 37,
  f: function() {
    return this.prop;
  }
};

console.log(o.f()); // 37
```

6. 作为一个DOM事件处理函数

当函数被用作事件处理函数时，它的 `this` 指向触发事件的元素（一些浏览器在使用非 `addEventListener` 的函数动态地添加监听函数时不遵守这个约定）。

```
// 被调用时，将关联的元素变成蓝色
function bluify(e){
  console.log(this === e.currentTarget); // 总是 true

  // 当 currentTarget 和 target 是同一个对象时为 true
  console.log(this === e.target);
  this.style.backgroundColor = '#A5D9F3';
}

// 获取文档中的所有元素的列表
var elements = document.getElementsByTagName('*');

// 将bluify作为元素的点击监听函数，当元素被点击时，就会变成蓝色
for(var i=0 ; i<elements.length ; i++){
  elements[i].addEventListener('click', bluify, false);
}
```

7. 作为一个内联事件处理函数

当代码被内联 on-event 处理函数 (en-US) 调用时，它的 `this` 指向监听器所在的DOM元素：

```
<button onclick="alert(this.tagName.toLowerCase());">
  Show this
</button>
```

上面的 alert 会显示 `button`。注意只有外层代码中的 `this` 是这样设置的：

```
<button onclick="alert((function(){return this})());">
  Show inner this
</button>
```

在这种情况下，没有设置内部函数的 `this`，所以它指向 global/window 对象（即非严格模式下调用的函数未设置 `this` 时指向的默认对象）。

8.类中的this

和其他普通函数一样，方法中的 `this` 值取决于它们如何被调用。有时，改写这个行为，让类中的 `this` 值总是指向这个类实例会很有用。为了做到这一点，可在构造函数中绑定类方法：

```
class Car {
  constructor() {
    // Bind sayBye but not sayHi to show the difference
    this.sayBye = this.sayBye.bind(this);
  }
  sayHi() {
    console.log(`Hello from ${this.name}`);
  }
  sayBye() {
    console.log(`Bye from ${this.name}`);
  }
  get name() {
    return 'Ferrari';
  }
}

class Bird {
  get name() {
    return 'Tweety';
  }
}

const car = new Car();
const bird = new Bird();

// The value of 'this' in methods depends on their caller
car.sayHi(); // Hello from Ferrari
bird.sayHi = car.sayHi;
bird.sayHi(); // Hello from Tweety

// For bound methods, 'this' doesn't depend on the caller
bird.sayBye = car.sayBye;
bird.sayBye();  // Bye from Ferrari
```

> **注意：**类内部总是严格模式。调用一个 `this` 值为 undefined 的方法会抛出错误。

面向对象编程介绍

什么是对象

一切皆对象

对象到底是什么，我们可以从两个方面来理解

(1) 对象是事物的抽象。

一个人、一架飞机、一个人都可以是对象，一张网页、一个西瓜、一个次数据库连接连接也可以是对象。当实际的物体被抽象成对象，实物之间的关系就变成了对象之间的关系，从而我们就可以模拟对象之间的关系，针对对象进行编程。

(2) 对象是一个实体，里面封装了属性（property）和方法（method）。

属性是对象的状态，方法是对象的行为。例如，我们可以把人抽象为Person对象，使用“属性”记录具体人的种类，使用“方法”表示人的某种行为（说话，唱歌，跑步等等）。

在实际开发中，对象是一个抽象的概念，可以将其简单理解为：**数据集或功能集。**

ECMAScript-262 把对象定义为：**无序属性的集合，其属性可以包含基本值、对象或者函数。** 严格来讲，这就相当于说对象是一组没有特定顺序的值。对象的每个属性或方法都有一个名字，而每个名字都映射到一个值。

什么是面向对象编程

面向对象编程 —— Object Oriented Programming，简称 OOP ，是一种编程开发思想。 它将真实世界各种复杂的关系，抽象为一个个对象，然后由对象之间的分工与合作，完成对真实世界的模拟。

在面向对象程序开发思想中，每一个对象都是功能中心，具有明确分工，可以完成接受信息、处理数据、发出信息等任务。 因此，面向对象编程具有灵活、代码可复用、高度模块化等特点，容易维护和开发，比起由一系列函数或指令组成的传统的过程式编程（procedural programming），更适合多人合作的大型软件项目。

面向对象的特性：

- 封装性
- 继承性
- 多态性

JS 中创建对象的方式

在 JavaScript 中，所有数据类型都可以视为对象，当然也可以自定义对象。 自定义的对象数据类型就是面向对象中的类（ Class ）的概念。

1. new Object 或者对象字面量

new Object

```
      // new Object() 或者对象字面量
var person = new Object();
person.name = 'Tom';
person.age = 18;

person.sayName = function () {
    console.log(this.name);
};
console.log(person);
```

对象字面量方式

```
var person1 = {
  name: 'Jack',
  age: 18,
  sayName: function () {
    console.log(this.name)
  }
}
```

通过上面的两种方式可以看出，这样写的代码太过冗余，重复性太高。

2. 通过工厂函数方式进行改进

```
function createPerson(name, age) {
    return {
        name: name,
        age: age,
        sayHello: function () {
            console.log('hello');
        },
    };
}
let person = createPerson('Thomas', 18);
console.log(person);
```

3. 更优雅的工厂函数：构造函数

如下所示：

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.sayHello = function () {
        console.log('hello');
    };
}

var p1 = new Person('Thomas', 18); // 返回了一个新的对象
p1.sayHello(); // => Thomas

var p2 = new Person('Jack', 23);
p2.sayHello(); // => Jack
console.log(p1, p2);
```

构造函数和实例之间的关系

```
console.log(p1.constructor === Person) // => true
console.log(p2.constructor === Person) // => true
console.log(p1.constructor === p2.constructor) // => true
```

另外，通过 instanceof 操作符也可以检查对象的类型

```
console.log(p1 instanceof Person);
```

综上所述，构造函数是根据具体的事物抽象出来的抽象模板，实例对象是根据抽象的构造函数模板得到的具体实例对象， 每一个实例对象都具有一个 `constructor` 属性，指向创建该实例的构造函数 。

构造函数的问题

使用构造函数可以大大简化创建对象的方式，但是其本身也存在一个浪费内存的问题：

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.sayHello = function () {
        console.log('hello');
    };
}
```

```
var p1 = new Person('Thomas', 18);
p1.sayHello(); // => Thomas

var p2 = new Person('Jack', 23);
p2.sayHello(); // => Jack
console.log(p1.sayHello === p2.sayHello); // p1 和 p2 的函数不一样
```

从上面的例子可以看出， 每一次生成一个实例，sayHello 这个函数都是不一样的，如果实例对象很多，会造成极大的内存浪费。

解决这个问题我们可以把要共享的函数定义到构造函数外部，通过命名空间来解决可能的名字冲突的问题

```
let fn = {
    sayHello: function () {
        console.log('hello');
    },
};
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.sayHello = fn.sayHello;
}

var p1 = new Person('Thomas', 18);
p1.sayHello(); // => Thomas

var p2 = new Person('Jack', 23);
p2.sayHello(); // => Jack
console.log(p1.sayHello === p2.sayHello); // p1 和 p2 的函数一样
```

基本上解决了构造函数的内存浪费问题,但是代码看起来还没有那么优雅。

4. 更好的解决方式： prototype

Javascript 规定，每一个构造函数都有一个 `prototype` 属性，指向另一个对象。 这个对象的所有属性和方法，都会被构造函数的实例继承。

这也就意味着，我们可以把所有对象实例需要共享的属性和方法直接定义在 `prototype` 对象上。

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype.sayHello = function () {
    console.log('hello');
};

var p1 = new Person('Thomas', 18);
p1.sayHello(); // => Thomas

var p2 = new Person('Jack', 23);
p2.sayHello(); // => Jack
console.log(p1.sayHello === p2.sayHello); // p1 和 p2 的函数一样
```

这个时候的sayHello,指向的都是同一个地址，减少内存空间的使用。

从新认识constructor 、实例和prototype 三者之间的关系

 1641112904233

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype.sayHello = function () {
    console.log('hello');
};

var p1 = new Person('Thomas', 18);
p1.sayHello(); // => Thomas
console.log(Object.getPrototypeOf(p1) === Person.prototype); // true
console.log(Person.prototype.constructor === Person); // true
console.log(Object.getPrototypeOf(p1));
```

任何函数都具有一个 `prototype` 属性，该属性是一个对象。

```
function Person () {}
console.log(Person.prototype) // => 对象
```

构造函数的 `prototype` 对象默认都有一个 `constructor` 属性，指向 `prototype` 对象所在函数。

```
console.log(Person.prototype.constructor === Person); // true
```

通过Object.getPrototypeOf 可以获取 指向构造函数的 `prototype` 对象。

```
var instance = new Person()
console.log(Object.getPrototypeOf(instance) === Person.prototype) // => true
```

```
警告： 当Object.prototype.__proto__ 已被大多数浏览器厂商所支持的今天，其存在和确切行为仅在ECMAScript 2015规范中被标准化为传统功能，以确保Web浏览器的兼容性。为了更好的支持，建议只使用Object.getPrototypeOf()。https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/proto
```

理解原型链

 1641113820131

```
function Foo() {}
Foo.prototype.abc = 123;
let f = new Foo();
console.log(f.__proto__.__proto__);
console.log(f.__proto__.__proto__.__proto__);
console.log(f.toString());
console.log(f.abc);
```

接下来我们来解释一下为什么实例对象可以访问原型对象中的成员。

每当代码读取某个对象的某个属性时，都会执行一次搜索，目标是具有给定名字的属性

1. 首先从搜索对象实例本身开始，如果在实例中找到了具有给定名字的属性，则返回该属性的值
2. 如果没有找到，则继续搜索指针指向的原型对象，在原型对象中查找具有给定名字的属性
3. 如果在原型对象中找到了这个属性，则返回该属性的值

也就是说，在我们调用 `f.toString()` 的时候，会先后执行两次搜索：

- 首先，会先看一下实例 f 有没有 toString 属性吗
- 如果没有，就会继续找f的原型，看有没有 toString 属性
- 如果还是没有找到，就会继续找f原型，将会重现相同的搜索过程，直到null为止。

而这正是多个对象实例共享原型所保存的属性和方法的基本原理。

继承与原型链

继承案例

```
// 实现继承的核心函数
function inheritPrototype(subType, superType) {
    function F() {}
    //F()的原型指向的是superType
    F.prototype = superType.prototype;
    //subType的原型指向的是F()
    subType.prototype = new F();
    // 重新将构造函数指向自己，修正构造函数
    subType.prototype.constructor = subType;
}
// 设置父类
function Parent(name) {
    this.name = name;
    Parent.prototype.sayName = function () {
        console.log(this.name);
    };
}
// 设置子类
function Child(name, age) {
    //构造函数式继承--子类构造函数中执行父类构造函数
    Parent.call(this, name);
    this.age = age;
}
// 核心：因为是对父类原型的复制，所以不包含父类的构造函数，也就不会调用两次父类的构造函数造成浪费
inheritPrototype(Child, Parent);
// 添加子类私有方法
Child.prototype.sayAge = function () {
    console.log(this.age);
};
var instance = new Child('Thomas', 18);
console.dir(instance);
instance.sayName();
```

拓展：[javascript 中常用的八种继承方案]