DAY22 闭包和继承

2023年5月18日 18:13

函数的两个阶段

1.定义阶段

1.1.开辟一个存储空间

1.2.把函数体内的代码一模一样的放在这个空间内（不解析变量）

1.3.把存储空间的地址给函数名

```
const fn = function(){ .... }
```

2.调用阶段

2.1.按照函数名的地址找到函数的存储空间

2.2.形参赋值

2.3.预解析

2.4.在内存中开辟一个执行空间

2.5.将函数存储空间中的代码拿出来在刚刚开辟的执行空间中执行

2.6.执行完毕后，内存中开辟的执行空间销毁

注意：存储空间　　执行空间

3.函数执行空间

但是每一次调用都会生成一个完全不一样的执行空间

并且执行空间会在函数执行完毕后就销毁了，但是存储空间不会

```
function fn(n1, n2) {
  console.log(n1 + n2);
}
fn(1, 2);//开启执行空间，执行完成空间销毁
fn(3, 4);//再次开启执行空间，执行完成销毁
```

梳理作用域

全局作用域：全局变量拥有的作用域

局部作用域：函数内部的变量拥有的作用域

变量的生命周期：

全局变量的生命周期是永久的，不会销毁，开发中不建议使用全局变量。

局部变量随着函数调用的结束，里面的变量就会销毁，开发中建议使用局部变量

```
var num = 10;//全局变量，一直存在
function sum() {
  var a = 1;
  var b = 2;
  console.log(a + b);
}
```

```
sum()//a,b被销毁了 执行空间会在函数执行完毕后就销毁了
console.log(a);//a is not defined
```

浏览器垃圾回收机制

浏览器的 Javascript 具有自动垃圾回收机制，垃圾收集器会定期（周期性）找出那些不再继续使用的变量，然后释放其内存。

1.标记方式

变量被引用，身上绑上一个进入环境的标记，如果变量一直被使用，一直表示进入环境，如果变量不再引用，标记变成离开环境，垃圾回收机制可以回收。

2.计数方式

变量被引用，身上的计数器+1，随着引用，计数器的值会发生变化，如果不再引用，计数器的值减少，如果计数器的值为0，可以通知垃圾回收机制回收。

内存泄漏

如果变量不被使用，继续占据内存，无法释放，视为内存泄漏(引起性能问题)。

不会销毁的执行空间

闭包，就是要利用这个不销毁的执行空间.

```
function fn() {
  var a = 1;
  var b = 2;
  return {
    b: b
  }
}
```

函数的执行空间会在函数执行完毕之后销毁

- 但是，一旦函数内部返回了一个引用数据类型，并且在函数外部有变量接受的情况下
- 那么这个函数执行空间就不会销毁了

```
var c = fn();// a已经销毁，b没有销毁，执行空间也没有销毁
console.log(c);//{b: 2}
c = null;//执行空间销毁了

function fn1() {
  var a = 1;
  return function () {
    return a++;
  }
}

// 函数的 执行空间会在函数执行完毕之后销毁
var f1 = fn1(); //函数执行完成，理论上是要销毁执行空间，但是根据下面的输出证明没有被销毁。
console.log(f1());//1
```

```
console.log(f1());//2
console.log(f1());//3
```

闭包的概述

1.闭包概念

闭包就是能够读取其他函数内部变量的函数。或者把闭包理解成"定义在一个函数内部的函数"。

函数嵌套函数。

2.闭包形成的条件

2.1.不被销毁的空间(返回一个函数数据类型)

2.2.内部函数引用外部函数中的变量

```
function fn1() {
  var a = 1;
  return function () {//闭包
    return a++;
  }
}
let f1 = fn1();
console.log(f1());//1
console.log(f1());//2
console.log(f1());//3
f1 = null;//释放执行空间，里面的一切销毁了

function fn2(a) {//a是形成，形成相当于函数内部的变量
  return function () {//闭包
    return a++
  }
}
```

3.闭包的特点

3.1.作用域空间不销毁

优点：作用域空间不销毁,变量也不会销毁，增加了变量的生命周期(变量一直存在)

缺点：因为不销毁，变量会一直占用内存，多了以后就会导致内存溢出(手动销毁设为null)

3.2.可以利用闭包在一个函数外部访问函数内部的变量

优点： 可以再函数外部访问内部数据

缺点： 必须要时刻保持引用，导致函数执行空间不被销毁

```
function fn2() {
  var num = 10;
  return function () {
    return num;
  };
}
var copynum = fn2()()
console.log(copynum);
```

3.3.保护私有变量

优点： 可以把一些变量放在函数里面，不会污染全局

缺点： 要利用闭包函数才能访问，不是很方便

```
function fn2() {
  var num = 10;//局部变量，不会污染全局
  return function () {//要利用闭包函数才能访问,函数内部的变量只能给函数内部使用
    return num;
  };
}
```

闭包的简单应用

简单应用1

配合定时器返回循环的值

```
for (var i = 1; i <= 5; i++) {
  setTimeout(function () {
    console.log(i);//5次6  循环结束才执行定时器里面的函数，i=6
  }, 1000);
}

for (let i = 1; i <= 5; i++) {
  setTimeout(function () {
    console.log(i);//i:1,2,3,4,5,这里是因为let具有块级作用域，声明的变量绑定在这个区域
  }, 1000);
}

// 闭包实现
// 闭包让里面的变量延迟作用域，一直存在。
for (let i = 1; i <= 5; i++) {
  var fn = function (i) {//i:一直存在
    return function () {
      setTimeout(function () {
        console.log(i);//i:1,2,3,4,5,这里是因为let具有块级作用域，声明的变量绑定在这个区域
      }, 1000);
    }
  }
  fn(i)();//i:1,2,3,4,5
}
for (let i = 1; i <= 5; i++) {
  !function (i) {//i:一直存在
    setTimeout(function () {
      console.log(i);//i:1,2,3,4,5,这里是因为let具有块级作用域，声明的变量绑定在这个区域
    }, 1000);
  }(i);//i:1,2,3,4,5
```

```
}
```

简单应用2

```
<body>
  <ul>
    <li>1111</li>
    <li>2222</li>
    <li>3333</li>
    <li>4444</li>
  </ul>
</body>

const list = document.querySelectorAll('li');//4个
for (var i = 0; i < list.length; i++) {
  !function (i) {
    list[i].onclick = function () {
      // console.log(i);//4  循环结束的值，循环不结束，没有点击事件
      console.log(list[i].innerHTML);
    }
  }(i);
}
```

函数的防抖和节流

一.高频事件

高频事件：事件触发频率比较高，比如：input，mousemove，scroll...

```
let num = 0;
function dosomething() {//事件处理函数
  num++;
  console.log(num);
}
search.oninput = dosomething;
search.oninput = function dosomething() {//事件处理函数
  num++;
  console.log(num);
}
```

注意：当前的事件有很大的浪费，通过函数的防抖和节流来优化高频事件

二.函数的防抖

函数防抖(debounce)：当事件被触发一段时间后再执行事件，如果在这段时间内事件又被触发，则重新计时。

过程：在事件触发时开始计时，在规定的时间内，若再次触发事件，将上一次计时清空，然后重新开始计时。保证只有在规定时间内没有再次触发事件之后，再去执行这个事件。

封装函数实现函数的防抖，利用封装好的函数执行高频事件，减少高频事件触发的频率

封装好防抖函数出现两个问题

1.事件处理函数里面的this指向出现问题 - call或者apply改变this指向

2.事件处理函数里面的事件对象没有了 - arguments

例如：约定1s后执行事件，如果1s内事件再次触发，继续等待1s后执行。

```
<body>
  请输入搜索的内容：<input type="text" id="search">
</body>

function debounce(fn, time) {//防抖函数  fn:事件处理函数    time:事件
  let timer = null;//定时器的返回值
  return function () {//根据分析，这里返回的函数现在变成事件处理函数，将其还给fn。
    clearTimeout(timer);//将上一次计时清空,然后重新开始计时
    timer = window.setTimeout(() => {//将事件处理函数延迟1s执行，平时的开发中自由约定合理的时间
      // fn.call(this, arguments[0]);//箭头函数里面没有自己的this，箭头函数的this来自于父级，this指向父级，父级的this刚好就是当前操作的元素对象
      fn.apply(this, arguments);
    }, time);
  }
}
function dosomething(e) {//事件处理函数
  console.log('事件触发了');
  console.log(this);//this指向当前操作的元素对象
  console.log(e);//事件对象
}
search.oninput = debounce(dosomething, 1000);
```

三.函数的节流

函数节流（throttle）：指定时间间隔内，若事件被多次触发，只会执行一次.

在事件触发之后开始计时，在规定的时间内，若再次触发事件，不对此事件做任何处理。保证在规定时间内只执行一次事件.

例如：约定1s钟执行一次事件，如果1s内事件触发10，执行1次。

利用时间差计算

```
function throttle(fn, time) {//fn:事件处理函数  time:约定的时间
  let startTime = 0;// 起始时间
  return function () {
    let currentTime = new Date().getTime();//获取当前的时间戳
    if (currentTime - startTime >= time) {//计算时间差
      fn.call(this, arguments[0]);
      startTime = currentTime;//不断的将当前时间给起始时间，准备计算下一次的时间差
    }
  }
}
```

```
function dosomething(e) {//事件处理函数
  console.log('事件触发了');
  console.log(this);
  console.log(e);
}
search.oninput = throttle(dosomething, 1000);
```

函数柯里化

概念：在计算机科学中，柯里化（Currying）是把接受多个参数的函数变换成接受一个单一参数(最初函数的第一个参数)的函数，并且返回接受余下的参数且返回结果的新函数的技术。

```
function sum(n1, n2) {
  return n1 + n2;
}
console.log(sum(1, 2));//3
function sum(n1) {
  return function (n2) {
    return n1 + n2;
  }
}
console.log(sum(1)(2));//3
```

利用柯里化实例求和的案例

核心：递归拼接参数，配合push进数组

```
function sum(...res) {//res:获取第一组参数,变成数组  es6新增的rest参数
  let arg = [...res];//第一次的参数追加进数组
  return function result(...res1) {//result:递归返回函数 ...res1接受第二组参数,
    if (res1.length === 0) {//参数收集完成了
      return arg.reduce((curr, next) => curr + next);
    } else {//不断的收集更多的参数，追加进arg数组中
      arg.push(...res1);
      return result;//递归返回此函数，接收后续更多的参数
    }
  }
}

console.log(sum(1)(2)(3)(4)(5)());//15
console.log(sum(1, 2, 3)());//6
console.log(sum(1, 2)(3)(4, 5)());//15
console.log(sum(1)(2, 3, 4)());//10
console.log(sum(1)(2, 3)(4)());//10

// function sum(a) {
//   return function (b) {
//     return function (c) {
//       return function (d) {
//         return function (e) {
//           return a + b + c + d + e
//         }
```

```
//         }
//       }
//     }
// }
// console.log(sum(1)(2)(3)(4)(5));
```

弊端：如果参数继续添加，里面嵌套的函数非常，而且希望传参的方式可以一个或者多个一起，进行封装

继承的概述

一.面向对象的特点

企业级语言：抽象(关注核心应用)，封装，继承，多态

javascript：封装，继承，多态

封装：通过类封装属性和方法

继承：类出现相同的属性和方法的时候，可以继承(子类继承父类，但不能影响父类，可以扩展)

多态：函数对象根据不同的参数获取不同的结果

二.继承

让一个构造函数去继承另一个构造函数的属性和方法

子类继承父类的属性和方法，但不能影响父类，可以继承完成后进行扩展。

三.常见的继承方式

1.构造函数继承

2.原型(prototype)继承

3.组合继承

4.原型链继承

5.class继承 - 重点

混合开发的继承

子类继承父类的属性和方法，但不能影响父类，可以继承完成后进行扩展

```
// 新建一个父类Person
function Person(name, age, sex) {
  this.name = name;
  this.age = age;
  this.sex = sex;
}
// 原型方法
Person.prototype.showName = function () {
  console.log('来自于父级的showName方法');
}
Person.prototype.showAge = function () {
  console.log('来自于父级的showAge方法');
}
Person.prototype.showSex = function () {
  console.log('来自于父级的showSex方法');
```

```
}
```

1.构造函数的继承

```
// 新建一个子类
function Student(name, age, sex, grade) {
```

如果父类拥有的属性，直接继承过来，如果父类没有的属性，写在子类里面

如何继承构造函数里面的属性，通过call或者apply

this:表示子类的实例对象

```
  Person.call(this, name, age, sex);//父类构造函数里面this指向子类的this，继承父类构造函数上面的属性和方法。
  this.grade = grade;//不是继承的，是子类自己的
}
```

2.原型上面继承

2.1.拷贝继承

```
Student.prototype = Person.prototype;

for (let key in Person.prototype) {
  Student.prototype[key] = Person.prototype[key];
}

Object.assign(Student.prototype, Person.prototype)
```

2.2.原型链继承

```
Student.prototype = new Person; //通过原型链实现，父类的实例指向父类的原型(Person.prototype),Student.prototype指向了Person.prototype
Student.prototype.constructor = Student;//将子类的构造函数修改回来

Student.prototype.showName = function () {//自己拥有的方法，直接使用自己的。
  console.log('我是来自子类student上面的showName方法');
}
let s1 = new Student('zhangsan', 18, '男', '2303班')
console.log(s1.name);
console.log(s1.age);
console.log(s1.sex);
s1.showName()
s1.showAge()
s1.showSex()
console.log(Student);//子类的构造函数
console.log(s1.constructor);//子类的构造函数
console.log(Student.prototype.constructor);//子类的构造函数
```

ES6的class继承

Class 可以通过extends关键字实现继承，让子类继承父类的属性和方法。

extends 的写法比 ES5 的原型链继承，要清晰和方便很多。

```
class Person {
  constructor(name, age, sex) {
    this.name = name;
    this.age = age;
    this.sex = sex;
  }
  showName() {
    console.log('来自于父级的showName方法');
  }
  showAge() {
    console.log('来自于父级的showAge方法');
  }
  showSex() {
    console.log('来自于父级的showSex方法');
  }
  show() {
    return `我的名字叫${this.name},我今年${this.age}岁,我是${this.sex}的`;
  }
}
```

1.extends

可以通过extends关键字实现继承，让子类继承父类的属性和方法。

2.super

第一种情况，super作为函数调用时，代表父类的构造函数。

第二种情况，super作为对象时，代表时父类的原型。

子类在调用super()之前，是没有this对象的,任何对this的操作都要放在super()的后面。

子类的this是父类的构造函数塑造的

```
class Student extends Person {
  constructor(name, age, sex, grade) {
    super(name, age, sex);//利用super继承父类的属性和方法,super代表父类的构造函数
    this.grade = grade;//注意：放到super下面
    // 子类在调用super()之前，是没有this对象的,任何对this的操作都要放在super()的后面。
    // 子类的this是父类的构造函数塑造的
  }
  showName() {
    console.log('来自于子类自己的showName方法');
  }
  show() {
    return `${super.show()},我来自${this.grade}`;
  }
}
```

```
let s1 = new Student('zhangsan', 18, '男', '2303班')
// console.log(s1.name);
// console.log(s1.age);
// console.log(s1.sex);
s1.showName()
// s1.showAge()
// s1.showSex()
// console.log(s1.show());
```