函数进阶

函数提升

> 类似于变量提升 **会把函数的声明提升到当前作用域的最顶部**

```
fn() //会把函数的声明提升到当前作用域的最顶部
functino fn() {
console.log(`提升`)
}
```

可以

不提升调用 不提升赋值

```
fun()
var fun = funtion (){console.log()}
```

相当于

```
var fun()
fun()
fun = funtion () {}
```

不可以 函数表达式必须声明赋值好再调用

函数参数

形参实参

arguments 动态参数

> arguments只存在函数里面 **实际上是一个伪数组**

```
funtion getSum()
{
    for( let i = 0 ; i < arguments.length;i++)
  {
  sum += arguments[i]
  }
}
getSum(1,1,2,2,31,4)
getSum(1,2,3)
```

arguments在箭头箭头函数里没有 而且这是个伪数组 他的方法式特殊的

rest 剩余参数

> …arr 这个**真数组，他只接收形参以外的东西，必须要放在最后**

```
funtion getSum (a,b,...list)
{
    console.log(list)//不用写...
}
```

对象中的rest

```
function connect({ host, port, ...user }) {
            console.log(host)
            console.log(port)
            console.log(user)
        }
        connect({
            host: '127.0.0.1',
            port: 3306,
            username: 'root',
            password: 'root',
            type: 'master'
        })
```

```
127.0.0.1
3306
▶ {username: 'root', password: 'root', type: 'master'}
```

对象中的展开运算符

做对象的合并

```
const a = {
            q: "a"
        }
        const b = {
            w: "b"
        }
        const c = {
            e: "c"
        }
        const d = { ...a, ...b, ...c }
        console.log(d)//{q: 'a', w: 'b', e: 'c'}
```

⚠️区分展开运算符 ...

> 不一定在函数里的

```
const arr = [1,2,3]
console.log(...arr) // 1 2 3
//...arr === 1,2,3
```

不会修改数组

应用：求最大值

```
console.log(Math.arr(..arr) //3 Math.arr(不能传数组)
```

应用：合并数组

```
const arr2 = [3,4,5]
const arr3 = [...arr,...arr2]// [1,2,3,3,4,5]
```

箭头函数

> 箭头函数没有arguments变量
> 箭头函数不能作为构造实例化对象
> 箭头函数的this是静态的，它的this就是声明的时候的this
> 箭头函数省略花括号之后要省略return

基本语法

> ！important 目的是为了更短的代码，把函数表达式更加简洁

```
const fn = funtion() {
console.log(123)}
//可以等价于=>
const fn = (x) => {
console.log(123)}
```

案例：

```
cosnt arr = [1,6,7,11,2,90]
cosnt result = arr.filter(item => item % 2 === 0 )
```

省略

```
//'这个小括号在只有一个参数的时候可以省略'
const fn = x => console.log(123)//只有一行代码的时候大括号可以省略
const fn = x => x + x //省略了return
console.log(fn(1))//2
```

直接返回对象

> 用小括号包含对象，因为函数体也是{}

```
const fn = (uname) => ({name:uname})//用小括号包含起来
```

箭头函数参数

> 没有arguments 但是有…arr剩余参数

```
const getSum = (...arr) => {
let sum =  0
for ( let i = 0 ; i < arr.lenght; i++
sum += arr[i]
return result
}
```

this

> this就是调用此函数的对象

对于普通情况而言：

```
console.log(this) //window
function fn() {
console.log(this)//window
}

//对象方法中的this
const obj = {
name : 'andy',
sayHi: function() {
console.log(this)//得到this==obj
}
obj.sayHi
```

对于箭头函数而言

> 箭头函数不会创this，只会从作用域链的上一层来找this
> 所以在dom中**用回调函数的时候就不用箭头函数**（丢失this 只会指向window）

```
const fn = () => {
console.log(this)//在这个局部作用域中没有this，
//所以找到的是window，实际上不是window调用的
}

//对象方法的箭头函数的this
const obj = {
name : 'andy',
sayHi : () => {
console.log(this)//这个作用域里没有this，所以找到的this也是window
}
obj.sayHi
```

处理this

> 严格模式下没有调用调用主体的时候this = undefine

普通函数

> 谁调用就指向谁

箭头函数

> 箭头函数会找最近的外层的this，本身不会产生this

操作dom对象的时候要用this就不要用箭头函数

再构造函数和原型函数中也不要用箭头函数

改变this

call()

> 调用的同时改变this指向
> fun.call(thisArg,arg1,arg2)

```
funciton fn() {
console.log(this)
}
fn.call(obj)//输出obj
```

apply()

> 类似call
> fun.apply(thisArg,[argArr])

```
function fn (x, y){
console.log(this)
console.log(x+y)
}
fn.apply(obj,[1,2])
```

使用场景:求最大值

```
//const max = Math.max(...arr)
const max = Math.max.apply(Math,arr)
const max = Math.max.apply(null,arr)
```

bind()

> bind()不会调用函数
> 语法和call（）相似

```
function fn()  {
  console.log(this)
}
const newFn fn.bind(obj)//返回值是一个函数（原函数的拷贝）
```