

高阶技巧







- 1. 深入this学习,知道如何判断this指向和改变this指向
- 2. 知道在JS中如何处理异常,学习深浅拷贝,理解递归





- ◆ 深浅拷贝
- ◆ 异常处理
- ◆ 处理this
- ◆ 性能优化
- ◆ 综合案例





深浅拷贝

- · 浅拷贝
- 深拷贝

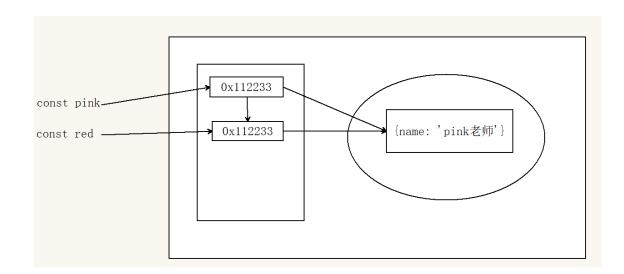


1. 深浅拷贝

开发中我们经常需要复制一个对象。如果直接用赋值会有下面问题:

```
// 一个pink对象
const pink = {
 name: 'pink老师',
 age: 18
const red = pink
console.log(red) // {name: 'pink老师', age: 18}
red.name = 'red老师'
console.log(red) // {name: 'red老师', age: 18}
// 但是 pink对象里面的name值也发生了变化
console.log(pink) // {name: 'red老师', age: 18}
```

这好比有同学来pink老师这里拷视频,竟然用的是剪切...气人不







深浅拷贝

- · 浅拷贝
- 深拷贝



1.1 浅拷贝

首先浅拷贝和深拷贝只针对引用类型

浅拷贝: 拷贝的是地址

常见方法:

1. 拷贝对象: Object.assgin() / 展开运算符 {...obj} 拷贝对象

2.拷贝数组: Array.prototype.concat() 或者 [...arr]

```
const obj = {
   uname: 'pink'
}
const o = { ...obj }
console.log(o) // {uname: 'pink'}
o.uname = 'red'
console.log(o) // {uname: 'red'}
console.log(obj) // {uname: 'pink'}

COUSOJE.JOG(opl) \\ {uname: 'pink'}
```

```
// 一个pink对象
const pink = {
  name: 'pink老师',
  age: 18
const red = {}
Object.assign(red, pink)
console.log(red) // {name: 'pink老师', age: 18}
red.name = 'red老师'
console.log(red) // {name: 'red老师', age: 18}
// 不会影响pink对象
console.log(pink) // {name: 'pink老师', age: 18}
```



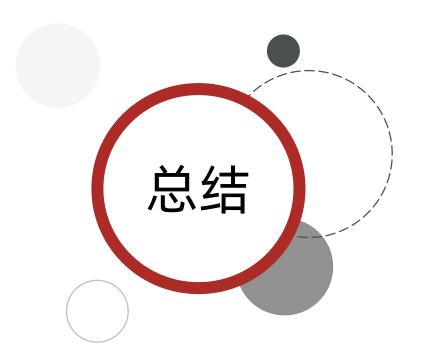
1.1 浅拷贝

```
·个pink对象
const pink = {
 name: 'pink老师',
 age: 18,
 famliy: {
   mother: 'pink妈妈'
const red = {}
Object.assign(red, pink)
console.log(red) // {name: 'pink老师', age: 18}
red.name = 'red老师'
// 更改对象里面的 family 还是会有影响
red.famliy.mother = 'red妈妈'
console.log(red) // {name: 'red 老师', age: 18}
// 不会影响pink对象
console.log(pink) // {name: 'pink老师', age: 18}
```

```
Elements
                   Console Network
                                      Recorder L
▶ O top ▼ O Filter
 ▶ {name: 'pink老师', age: 18, famliy: {...}}
 ▼{name: 'red老师', age: 18, famliy: {...}} 1
    age: 18
   ▶ famliy: {mother: 'red妈妈'}
    name: "red老师"
   ▶ [[Prototype]]: Object
 ▼{name: 'pink老师', age: 18, famliy: {...}} i
    age: 18
   ▶ famliy: {mother: 'red妈妈'}
    name: pink老师
   ▶ [[Prototype]]: Object
```

如果是简单数据类型拷贝值,引用数据类型拷贝的是地址(简单理解:如果是单层对象,没问题,如果有多层就有问题)





1. 直接赋值和浅拷贝有什么区别?

- ▶ 直接赋值的方法,只要是对象,都会相互影响,因为是直接拷贝对 象栈里面的地址
- 浅拷贝如果是一层对象,不相互影响,如果出现多层对象拷贝还会相互影响

2. 浅拷贝怎么理解?

- ▶ 拷贝对象之后,里面的属性值是简单数据类型直接拷贝值
- ▶ 如果属性值是引用数据类型则拷贝的是地址





深浅拷贝

- 浅拷贝
- 深拷贝



首先浅拷贝和深拷贝只针对引用类型

深拷贝: 拷贝的是对象, 不是地址

常见方法:

- 1. 通过递归实现深拷贝
- lodash/cloneDeep
- 3. 通过JSON.stringify()实现



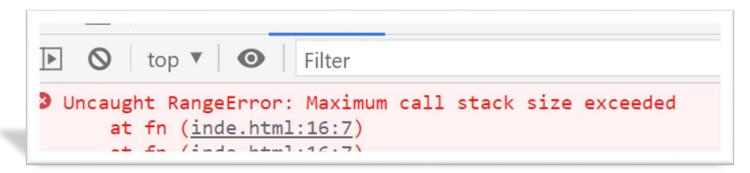
常见方法:

1. 通过递归实现深拷贝

函数递归:

如果一个函数在内部可以调用其本身,那么这个函数就是递归函数

- 简单理解:函数内部自己调用自己,这个函数就是递归函数
- 递归函数的作用和循环效果类似
- 由于递归很容易发生"栈溢出"错误(stack overflow),所以必须要加退出条件 return





常见方法:

1. 通过递归实现深拷贝

函数递归:

如果一个函数在内部可以调用其本身,那么这个函数就是递归函数

```
let num = 1
// fn就是递归函数
function fn() {
    console.log('我要打印6次')
    if (num >= 6) {
        return
    }
    num++
    fn() // 函数内部调用函数自己
}
fn()
```



函数递归:

利用递归函数实现 setTimeout 模拟 setInterval效果





• 利用递归函数实现 setTimeout 模拟 setInterval效果

需求:

①:页面每隔一秒输出当前的时间

②:输出当前时间可以使用: new Date().toLocaleString()





• 利用递归函数实现 setTimeout 模拟 setInterval效果

需求:

①:页面每隔一秒输出当前的时间

②:输出当前时间可以使用: new Date().toLocaleString()

```
function getTime() {
  const time = new Date().toLocaleString()
  console.log(time)
  setTimeout(getTime, 1000) // 定时器调用当前函数
}
getTime()
```



常见方法:

1. 通过递归函数实现深拷贝(简版)

```
const o = {}
function deepCopy(newObj, oldObj) {
  for (let k in oldObj) {
    if (oldObj[k] instanceof Array) {
        newObj[k] = []
        deepCopy(newObj[k], oldObj[k])
    } else if (oldObj[k] instanceof Object) {
        newObj[k] = {}
        deepCopy(newObj[k], oldObj[k])
    }
    else {
        newObj[k] = oldObj[k]
    }
}
```

```
const o = {}
function deepCopy(newObj, oldObj) {
   for (let k in oldObj) {
        newObj[k] = oldObj[k]
   }
}
deepCopy(o, obj)

qeebCobx(o' opj)
```

```
const o = {}
function deepCopy(newObj, oldObj) {
  for (let k in oldObj) {
    if (oldObj[k] instanceof Array) {
        newObj[k] = []
        deepCopy(newObj[k], oldObj[k])
    } else if (oldObj[k] instanceof Object) {
        newObj[k] = {}
        deepCopy(newObj[k], oldObj[k])
    }
    else {
        newObj[k] = oldObj[k]
    }
}
```



常见方法:

2. js库lodash里面cloneDeep内部实现了深拷贝

```
const obj = {
 uname: 'pink',
 age: 18,
 hobby: ['篮球', '足球'],
 family: {
   baby: '小pink'
// 语法: _.cloneDeep(要被克隆的对象)
const o = _.cloneDeep(obj)
console.log(o)
o.family.baby = '老pink'
console.log(obj)
```

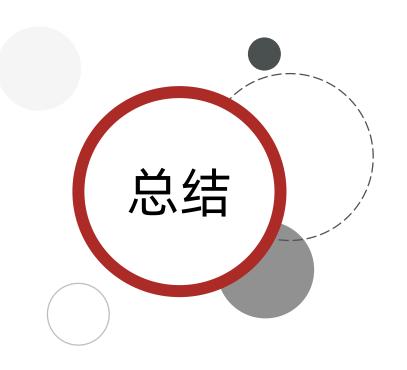


常见方法:

3. 通过JSON.stringify()实现

```
const obj = {
 uname: 'pink',
 age: 18,
 hobby: ['篮球', '足球'],
 family: {
   baby: '小pink'
const o = JSON.parse(JSON.stringify(obj))
console.log(o)
o.family.baby = '老pink'
console.log(obj)
```





1. 实现深拷贝三种方式?

- ▶ 自己利用递归函数书写深拷贝
- ➤ 利用js库 lodash里面的 _.cloneDeep()
- ➤ 利用JSON字符串转换

```
// 语法: _.cloneDeep(要被克隆的对象)
const o = _.cloneDeep(obj)
```

const o = JSON.parse(JSON.stringify(obj))





- ◆ 深浅拷贝
- ◆ 异常处理
- ◆ 处理this
- ◆ 性能优化
- ◆ 综合案例





异常处理

- · throw 抛异常
- try /catch 捕获异常
- debugger

了解 JavaScript 中程序异常处理的方法,提升代码运行的健壮性。



2.1 throw 抛异常

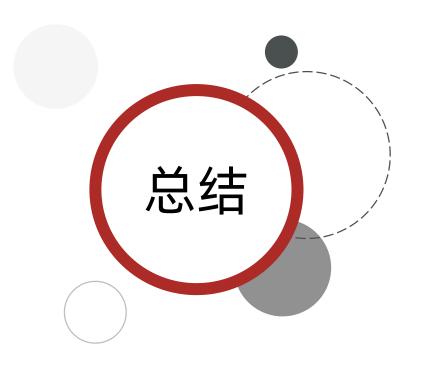
异常处理是指预估代码执行过程中可能发生的错误,然后最大程度的避免错误的发生导致整个程序无法继续运行

```
function counter(x, y) {
  if (!x || !y) {
    // throw '参数不能为空!';
    throw new Error('参数不能为空!')
  return x + y
counter()
                            Netwo
         Elements
                   Console
   O top ▼ O
                   Filter
3 ▶ Uncaught Error: 参数不能为空!
     at counter (inde.html:17:15)
     at inde.html:22:5
```

总结:

- 1. throw 抛出异常信息,程序也会终止执行
- 2. throw 后面跟的是错误提示信息
- 3. Error 对象配合 throw 使用,能够设置更详细的错误信息





- 1. 抛出异常我们用那个关键字? 它会终止程序吗?
 - ➤ throw 关键字
 - > 会中止程序
- 2. 抛出异常经常和谁配合使用?
 - ➤ Error 对象配合 throw 使用

```
function counter(x, y) {
    if (!x || !y) {
        // throw '参数不能为空!';
        throw new Error('参数不能为空!')
    }
    return x + y
}
counter()
conuter()
```





异常处理

- throw 抛异常
- try /catch 捕获异常
- debugger

了解 JavaScript 中程序异常处理的方法,提升代码运行的健壮性。



2.2 try/catch 捕获错误信息

我们可以通过try / catch 捕获错误信息(浏览器提供的错误信息)

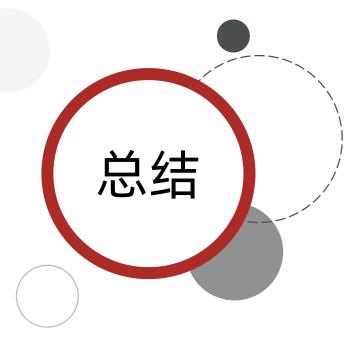
```
function foo() {
  try {
   const p = document.querySelector('.p')
   p.style.color = 'red'
  } catch (error) {
   console.log(error.message)
   return
     alert('执行')
 console.log('如果出现错误, 我的语句不会执行')
```

```
Cannot read properties of null (reading 'style')
```

总结:

- 1. try...catch 用于捕获错误信息
- 2. 将预估可能发生错误的代码写在 try 代码段中
- 3. 如果 try 代码段中出现错误后,会执行 catch 代码段,并截获到错误信息
- 4. finally 不管是否有错误,都会执行





- 1. 捕获异常我们用那2个关键字? 可能会出现的错误代码写到谁里面
 - > try catch
 - > try
- 2. 怎么调用错误信息?
 - ▶ 利用catch的参数

```
function foo() {
 try {
   // 查找 DOM 节点
   const p = document.querySelector('.p')
   p.style.color = 'red'
 } catch (error) {
   // 查看错误信息
   console.log(error.message)
   return
 console.log('如果出现错误,我的语句不会执行')
foo()
```





异常处理

- throw 抛异常
- try /catch 捕获异常
- debugger

了解 JavaScript 中程序异常处理的方法,提升代码运行的健壮性。



2.3 debugger

我们可以通过try / catch 捕获错误信息(浏览器提供的错误信息)

```
const arr = [1, 3, 5]
const newArr = arr.map((item, index) => {
   debugger
   console.log(item) // 当前元素
   console.log(index) // 当前元素索引号
   return item + 10 // 让当前元素 + 10
})
console.log(newArr) // [11, 13, 15]
console.log(newArr) // [11, 13, 15]
```

```
Console
                           Network
                                     Recorder A
                                                           Perfori
K |
        Elements
                                                  Sources
                                              闭包.html ×
top
                                                  1 Debugger pau
              17
                     const arr = [1, 3, 5]
▼ 🔷 file://
                                                  ▶ Watch
                     const newArr = arr.map((item,
 ▼ C:/User
                       debuggen
                                                  ▼ Breakpoints
     闭包.
                       console.log(item) // 当前元
                       console.log(index) // 当前元:
               22
                       return item + 10 // 让当前元
                                                  ▼ Scope
               23
                     console.log(newArr) // [11, 13 ▼Local
              24
               25
                                                     this: undefi
```





- ◆ 深浅拷贝
- ◆ 异常处理
- ◆ 处理this
- ◆ 性能优化
- ◆ 综合案例





处理this

- this指向
- · 改变this



3.1 处理this

this 是 JavaScript 最具"魅惑"的知识点,不同的应用场合 this 的取值可能会有意想不到的结果,在此我们对以往学习过的关于【this 默认的取值】情况进行归纳和总结。

目标: 了解函数中 this 在不同场景下的默认值,知道动态指定函数 this 值的方法

学习路径:

- 1. 普通函数this指向
- 2. 箭头函数this指向



3.1 this指向-普通函数

目标: 能说出普通函数的this指向

普通函数的调用方式决定了 this 的值,即【谁调用 this 的值指向谁】

```
function sayHi() {
  console.log(this)
// 函数表达式
const sayHello = function () {
  console.log(this)
// 函数的调用方式决定了 this 的值
sayHi() // window
window.sayHi()
```

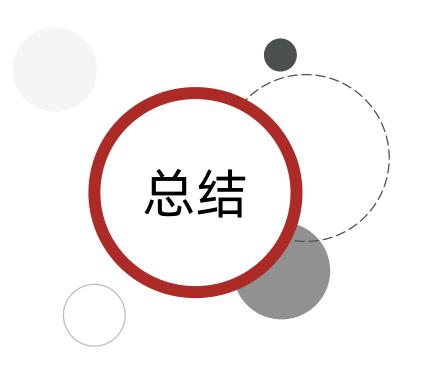
```
const user = {
  name: '小明',
  walk: function () {
    console.log(this)
user.sayHi = sayHi
uesr.sayHello = sayHello
user.sayHi()
user.sayHello()
```

```
'use strict'
function fn() {
   console.log(this) // undefined
}
fn()

tu()
```

普通函数没有明确调用者时 this 值为 window, 严格模式下没有调用者时 this 的值为 undefined





- 1. 普通函数this指向我们怎么记忆?
 - ▶ 【谁调用 this 的值指向谁】
- 2. 普通函数严格模式下指向谁?
 - ▶ 严格模式下指向 undefined



3.1 this指向-箭头函数

目标: 能说出箭头函数的this指向

箭头函数中的 this 与普通函数完全不同,也不受调用方式的影响,事实上箭头函数中并不存在 this!

- 1. 箭头函数会默认帮我们绑定外层 this 的值,所以在箭头函数中 this 的值和外层的 this 是一样的
- 2.箭头函数中的this引用的就是最近作用域中的this
- 3.向外层作用域中,一层一层查找this,直到有this的定义

```
console.log(this) // 此处为 window
// 箭头函数
const sayHi = function() {
   console.log(this) // 该箭头函数中的 this 为函数声明环境中 this 一致
}
```

```
// 普通对象
const user = {
    name: '小明',
    // 该箭头函数中的 this 为函数声明环境中 this 一致
    walk: () => {
        console.log(this)
    },
```



3.1 this指向-箭头函数

注意情况1:

在开发中【使用箭头函数前需要考虑函数中 this 的值】,事件回调函数使用箭头函数时,this 为全局的 window 因此DOM事件回调函数如果里面需要DOM对象的this,则不推荐使用箭头函数

```
// DOM 节点
const btn = document.querySelector('.btn')
// 箭头函数 此时 this 指向了 window
btn.addEventListener('click', () => {
   console.log(this)
})
// 普通函数 此时 this 指向了 DOM 对象
btn.addEventListener('click', function () {
   console.log(this)
})
})
```



3.1 this指向-箭头函数

注意情况2:

同样由于箭头函数 this 的原因,基于原型的面向对象也不推荐采用箭头函数

```
function Person() {
}
// 原型对像上添加了箭头函数

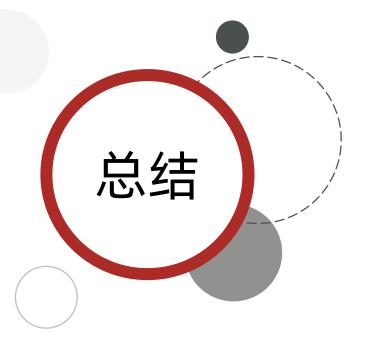
Person.prototype.walk = () => {
  console.log('人都要走路...')
  console.log(this); // window
}

const p1 = new Person()
p1.walk()
```

总结:

- 1. 函数内不存在this,沿用上一级的
- 2.不适用
- ▶ 构造函数,原型函数,dom事件函数等等
- 3. 适用
- > 需要使用上层this的地方
- 4. 使用正确的话,它会在很多地方带来方便,后面我们会大量使用慢慢体会





- 1. 函数内不存在this,沿用上一级的,过程:向外层作用域中,一层
 - 一层查找this, 直到有this的定义

2. 不适用

▶构造函数,原型函数,字面量对象中函数,dom事件函数

3. 适用

▶需要使用上层this的地方





处理this

- this指向
- 改变this



JavaScript 中还允许指定函数中 this 的指向,有 3 个方法可以动态指定普通函数中 this 的指向

- call()
- apply()
- bind()



1. call() -了解

使用 call 方法调用函数,同时指定被调用函数中 this 的值

● 语法:

fun.call(thisArg, arg1, arg2, ...)

- ▶ thisArg: 在 fun 函数运行时指定的 this 值
- ▶ arg1, arg2: 传递的其他参数
- ▶ 返回值就是函数的返回值,因为它就是调用函数

```
const obj = {
 name: 'pink'
function fn() {
  console.log(this) // 指向 obj {name: 'pink'}
fn.call(obj)
const obj = {
 name: 'pink'
function fn(x, y) {
  console.log(this) // 指向 obj {name: 'pink'}
  console.log(x + y) // 传递过来的参数相加
fn.call(obj, 1, 2)
```





- 1. call的作用是?
 - ▶ 调用函数,并可以改变被调用函数里面的this指向
- 2. call 里面第一个参数是 指定this, 其余是实参,传递的参数整体做个了解,后期用的很少

fn.call(obj, 1, 2)



JavaScript 中还允许指定函数中 this 的指向,有 3 个方法可以动态指定普通函数中 this 的指向

- call()
- apply()
- bind()



2. apply()-理解

使用 apply 方法调用函数,同时指定被调用函数中 this 的值

● 语法:

fun.apply(thisArg, [argsArray])

- ▶ thisArg: 在fun函数运行时指定的 this 值
- ➤ argsArray: 传递的值,必须包含在数组里面
- ▶ 返回值就是函数的返回值,因为它就是调用函数
- ➤ 因此 apply 主要跟数组有关系,比如使用 Math.max() 求数组的最大值

```
let result = counter.apply(null, [5, 10])
console.log(result)
console.log(result)

// 调用 counter 函数, 并传入参数
let result = counter.apply(null, [5, 10])
console.log(result)
```



2. apply()

求数组最大值2个方法:

```
// 求数组最大值

const arr = [3, 5, 2, 9]

console.log(Math.max.apply(null, arr)) // 9 利用apply

console.log(Math.max(...arr)) // 9 利用展开运算符
```





- 1. call和apply的区别是?
 - ▶ 都是调用函数,都能改变this指向
 - ▶ 参数不一样,apply传递的必须是数组



JavaScript 中还允许指定函数中 this 的指向,有 3 个方法可以动态指定普通函数中 this 的指向

- call()
- apply()
- bind()



3. bind()-重点

- bind() 方法不会调用函数。但是能改变函数内部this 指向
- 语法:

```
fun.bind(thisArg, arg1, arg2, ...)
```

- ➤ thisArg: 在 fun 函数运行时指定的 this 值
- ➤ arg1, arg2: 传递的其他参数
- ▶ 返回由指定的 this 值和初始化参数改造的 原函数拷贝 (新函数)
- ▶ 因此当我们只是想改变 this 指向,并且不想调用这个函数的时候,可以使用 bind,比如改变定时器内部的 this指向.

```
function sayHi() {
 console.log(this)
let user = {
 name: '小明',
let sayHello = sayHi.bind(user);
sayHello()
```



call apply bind 总结

● 相同点:

》都可以改变函数内部的this指向.

● 区别点:

- > call 和 apply 会调用函数,并且改变函数内部this指向.
- > call 和 apply 传递的参数不一样, call 传递参数 aru1, aru2..形式 apply 必须数组形式[arg]
- bind 不会调用函数,可以改变函数内部this指向.

● 主要应用场景:

- > call 调用函数并且可以传递参数
- > apply 经常跟数组有关系. 比如借助于数学对象实现数组最大值最小值
- bind 不调用函数,但是还想改变this指向. 比如改变定时器内部的this指向.





- ◆ 深浅拷贝
- ◆ 异常处理
- ◆ 处理this
- ◆ 性能优化
- ◆ 综合案例





性能优化

- 防抖
- 节流



4.2 节流

● 节流 (throttle)

所谓节流,就是指连续触发事件但是在 n 秒中只执行一次函数



只有等到了上一个人做完核酸,整个动作完成了,第二个人才能排队跟上。



4.2 节流

● 节流 (throttle)

所谓节流,就是指连续触发事件但是在 n 秒中只执行一次函数

● 开发使用场景 - 小米**轮播图点击效果 、 鼠标移动、页面尺寸缩放resize、滚动条滚动 就可以加节流**

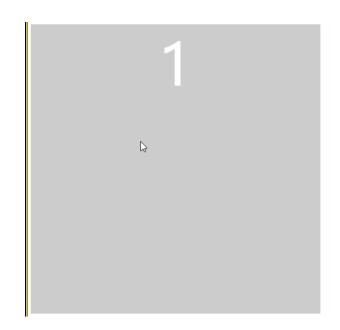


- 假如一张轮播图完成切换需要300ms, 不加节流效果, 快速点击, 则嗖嗖嗖的切换
- 加上节流效果, 不管快速点击多少次, 300ms时间内, 只能切换一张图片。





要求: 鼠标在盒子上移动, 里面的数字就会变化+1







要求: 鼠标在盒子上移动, 里面的数字就会变化+1

①: 如果以前方式,每次鼠标移动就会有大量操作,触发频次太高

```
const box = document.querySelector('.box')
let i = 1
function mouseMove() {
  box.innerHTML = i++
  // 如果存在开销较大操作, 大量数据处理, 大量dom操作, 可能会卡
}
box.addEventListener('mousemove', mouseMove)
```





要求: 鼠标在盒子上移动, 里面的数字就会变化+1

利用节流的方式, 鼠标经过, 500ms, 数字才显示

核心思路:

利用时间相减:移动后的时间 - 刚开始移动的时间 >= 500ms 我才去执行 mouseMove函数

①: 写一个节流函数throttle,来控制这个操作函数(mouseMove),500ms之后才去执行这个函数

②: 节流函数传递2个参数, 第一个参数 mouseMove函数,第二个参数 指定时间500ms

③: 鼠标移动事件, 里面写的是节流函数



国 案例

利用节流来处理-鼠标滑过盒子显示文字

要求: 鼠标在盒子上移动, 里面的数字就会变化+1

利用节流的方式, 鼠标经过, 500ms, 数字才显示

核心思路:

利用时间相减:移动后的时间 - 刚开始移动的时间 >= 500ms 我才去执行 mouseMove函数

④: 声明一个起始时间 startTime = 0

⑤: 但是节流函数因为里面写的函数名 throttle(mouseMove, 500), 是调用函数, 无法再次调用执行, 所以需要在节流函数里面写return 函数 这样可以多次执行

⑥: 记录当前时间 now = Date.now()

⑦:进行判断 如果大于等于 500ms,则执行函数,但是千万不要忘记 让起始时间 = 现在时间





要求: 鼠标在盒子上移动, 里面的数字就会变化+1 利用节流的方式, 鼠标经过, 500ms, 数字才显示

```
const box = document.querySelector('.box')
let i = 1
function mouseMove() {
 box.innerHTML = i++
 // 如果存在开销较大操作,大量数据处理,大量dom操作,可能会卡
function throttle(fn, t = 500) {
 let startTime = 0
 return function () {
   let now = Date.now()
   if (now - startTime >= t) {
    // console.log(1)
     fn()
     startTime = now
box.addEventListener('mousemove', throttle(mouseMove, 500))
```



4.1 防抖

● 防抖 (debounce)

所谓防抖,就是指触发事件后在 n 秒内函数只能执行一次,如果在 n 秒内又触发了事件,则会重新计算函数执行时间

● 举个栗子:

北京买房政策:需要连续5年的社保,如果中间有一年断了社保,则需要从新开始计算 比如,我 2020年开始计算,连续交5年,也就是到2024年可以买房了,包含2020年 但是我 2024年断社保了,整年没交,则需要从2025年开始算第一年往后推5年...也就是 2029年才能买房...



4.1 防抖

- 防抖 (debounce)

 所谓防抖,就是指触发事件后在 n 秒内函数只能执行一次,如果在 n 秒内又触发了事件,则会重新计算函数执行时间
- 开发使用场景-搜索框防抖

假设输入就可以发送请求,但是不能每次输入都去发送请求,输入比较快发送请求会比较多 我们设定一个时间,假如300ms, 当输入第一个字符时候,300ms后发送请求,但是在200ms的时候又输入了一个字符, 则需要再等300ms 后发送请求





利用防抖来处理-鼠标滑过盒子显示文字

要求: 鼠标在盒子上移动, 鼠标停止之后, 500ms后里面的数字就会变化+1

```
const box = document.querySelector('.box')
let i = 1
function mouseMove() {
  box.innerHTML = i++
  // 如果存在开销较大操作, 大量数据处理, 大量dom操作, 可能会卡
}
```



1 案例

利用防抖来处理-鼠标滑过盒子显示文字

要求: 鼠标在盒子上移动, 鼠标停止之后, 500ms后里面的数字就会变化+1

利用防抖的方式实现

核心思路:

利用定时器实现,当鼠标滑过,判断有没有定时器,还有就清除,以最后一次滑动为准开启定时器

①:写一个防抖函数debounce ,来控制这个操作函数(mouseMove)

②:防抖函数传递2个参数,第一个参数 mouseMove函数,第二个参数 指定时间500ms

③: 鼠标移动事件, 里面写的是防抖函数

④: 声明定时器变量 timeld

⑤: 但是节流函数因为里面写的函数名 debounce(mouseMove, 500), 是调用函数, 无法再次调用执行, 所以需要在节流函数里面写return 函数 这样可以多次执行



国 案例

利用防抖来处理-鼠标滑过盒子显示文字

要求: 鼠标在盒子上移动, 鼠标停止之后, 500ms后里面的数字就会变化+1

利用防抖的方式实现

核心思路:

利用定时器实现, 当鼠标滑过, 判断有没有定时器, 还有就清除, 以最后一次滑动为准开启定时器

⑥: 如果有定时器,则清除定时器

②: 否则开启定时器, 在设定时间内, 调用函数





利用防抖来处理-鼠标滑过盒子显示文字

```
const box = document.querySelector('.box')
let i = 1
function mouseMove() {
 box.innerHTML = i++
function debounce(fn, t = 500) {
 let timeId
 return function () {
   // 如果有定时器, 先清除
   if (timeId) clearTimeout(timeId)
   // 开启定时器
   timeId = setTimeout(function () {
     fn()
    }, t)
box.addEventListener('mousemove', debounce(mouseMove, 500))
```





1. 节流和防抖的区别是?

- ▶ 节流: 就是指连续触发事件但是在 n 秒中只执行一次函数,比如可以利用节流实现 1s之内 只能触发一次鼠标移动事件
- ▶ 防抖:如果在 n 秒内又触发了事件,则会重新计算函数执行时间

2. 节流和防抖的使用场景是?

- 节流: 鼠标移动,页面尺寸发生变化,滚动条滚动等开销比较 大的情况下
- ▶ 防抖:搜索框输入,设定每次输入完毕n秒后发送请求,如果期间还有输入,则从新计算时间



Lodash 库 实现节流和防抖

```
box.addEventListener('mousemove', _.throttle(mouseMove, 1000))
let i = 1
function mouseMove() {
   pox.addEventListener('mousemove', _.throttle(mouseMove, 1000))

box.addEventListener('mousemove', _.throttle(mouseMove, 1000))

const box = document.dnernSelector('.pox')

let i = 1

function mouseMove() {
   pox.addEventListener('mousemove', _.throttle(mouseMove, 1000))

const box = document.dnernSelector('.pox')

let i = 1

function mouseMove() {
   pox.addEventListener('mousemove', _.throttle(mouseMove, 1000))

pox.addEventListener('mousemove', _.throttle(mousemove', _.thr
```

```
const box = document.querySelector('.box')
let i = 1
function mouseMove() {
  box.innerHTML = i++
    // 如果存在开销较大操作, 大量数据处理, 大量dom操作, 可能会卡
}
box.addEventListener('mousemove', _.debounce(mouseMove, 1000))
```





- ◆ 深浅拷贝
- ◆ 异常处理
- ◆ 处理this
- ◆ 性能优化
- ◆ 节流综合案例





页面打开,可以记录上一次的视频播放位置

分析:

两个事件:

①: ontimeupdate 事件在视频/音频 (audio/video) 当前的播放位置发送改变时触发

②: onloadeddata 事件在当前帧的数据加载完成且还没有足够的数据播放视频/音频(audio/video)的

下一帧时触发

谁需要节流?

ontimeupdate, 触发频次太高了,我们可以设定 1秒钟触发一次





页面打开,可以记录上一次的视频播放位置

思路:

- 1. 在ontimeupdate事件触发的时候,每隔1秒钟,就记录当前时间到本地存储
- 2. 下次打开页面, onloadeddata 事件触发,就可以从本地存储取出时间,让视频从取出的时间播放,如果没有就默认为0s
- 3. 获得当前时间 video.currentTime



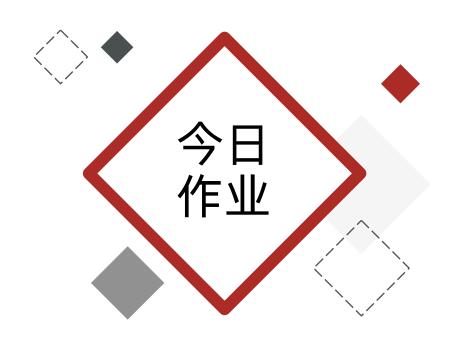


页面打开,可以记录上一次的视频播放位置

答案:

```
// 记录播放位置: 节流
const video = document.querySelector('video')
video.ontimeupdate = _.throttle(() => {
    localStorage.setItem('currentTime', video.currentTime)
}, 1000)
video.onloadeddata = () => {
    video.currentTime = localStorage.getItem('currentTime') || 0
}
```





- 1. 整理笔记
- 2. 开始做测试题: PC端地址: https://ks.wjx.top/vj/QARCGSJ.aspx
- 3. 总结整个js阶段内容, 重点看新语法
- 4. 明天上午考试,下午预习





传智教育旗下高端IT教育品牌