JavaScript 进阶第四天

高阶技巧

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

目录

Contents

学习目标

Learning Objectives

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

- 浅拷贝
- 深拷贝

1.1 浅拷贝

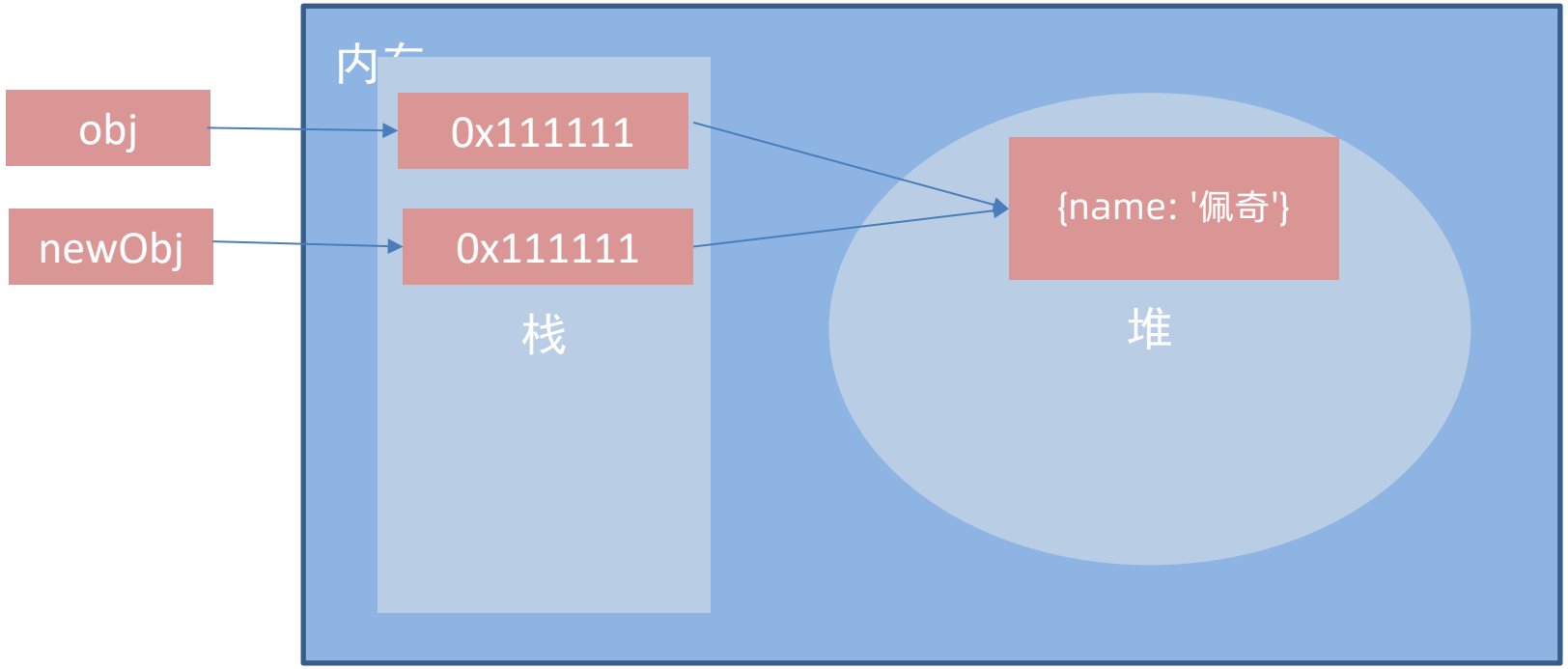
浅拷贝：把对象拷贝给一个新的对象，开发中我们经常需要复制一个对象

如果直接赋值，则复制的是地址，修改任何一个对象，另一个对象都会变化

```
//直接赋值复制的是地址
const obj = {
  name: '佩奇'
}
const newObj = obj

newObj.name = '乔治'

console.log(obj.nam
e)  // 也变成了乔治
```

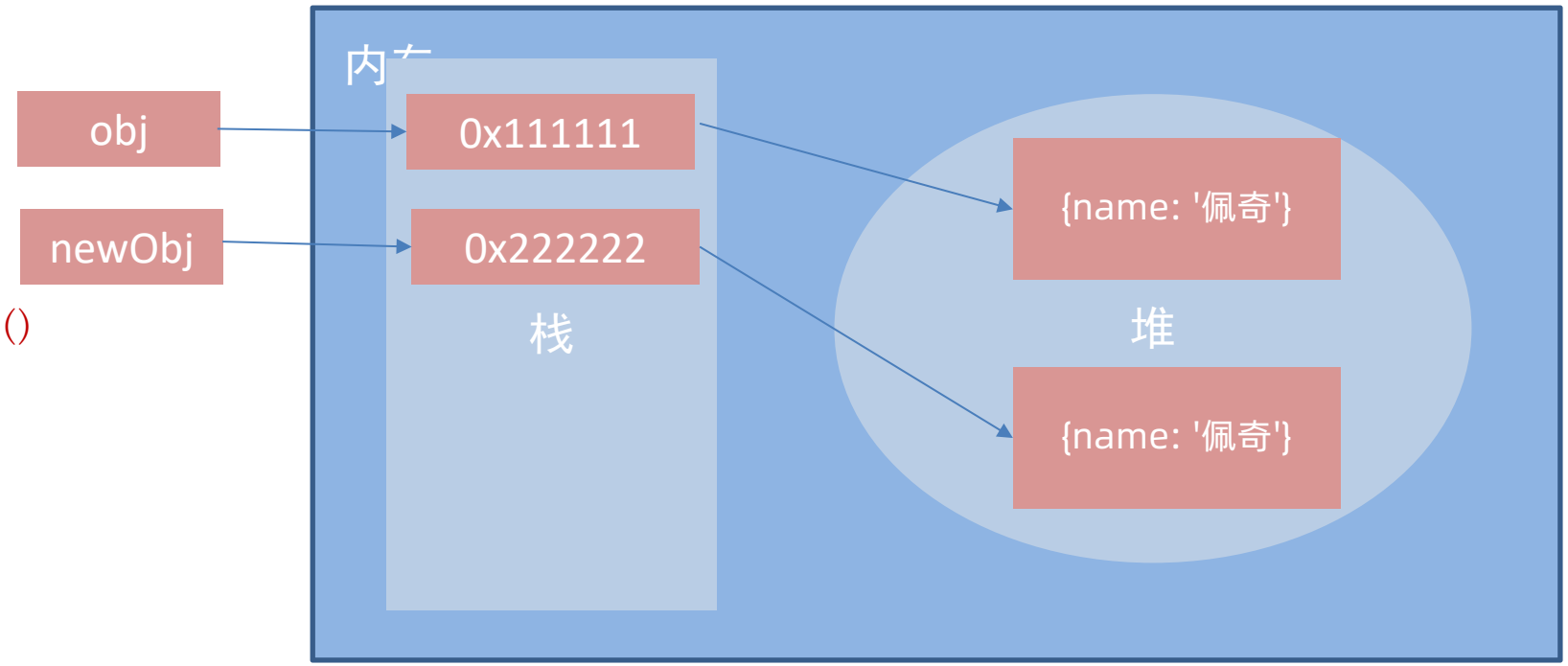


1.1 浅拷贝

浅拷贝：拷贝给一个新的对象

常见方法：

1. 拷贝对象：
   - Object. assign()
   - 展开运算符 {...obj}
2. 拷贝数组：
   - Array.prototype.concat()
   - 展开运算符 [...arr]



总结

1. =赋值实现复制对象有什么问题？
   - = 赋值是复制地址，两个对象使用同一个地址，修改会相互影响
2. 浅拷贝有几种方法？
   - 对象浅拷贝有 Object.assign() 和 展开运算符 {...obj}
   - 数组浅拷贝有 concat 和展开运算符 [...arr]

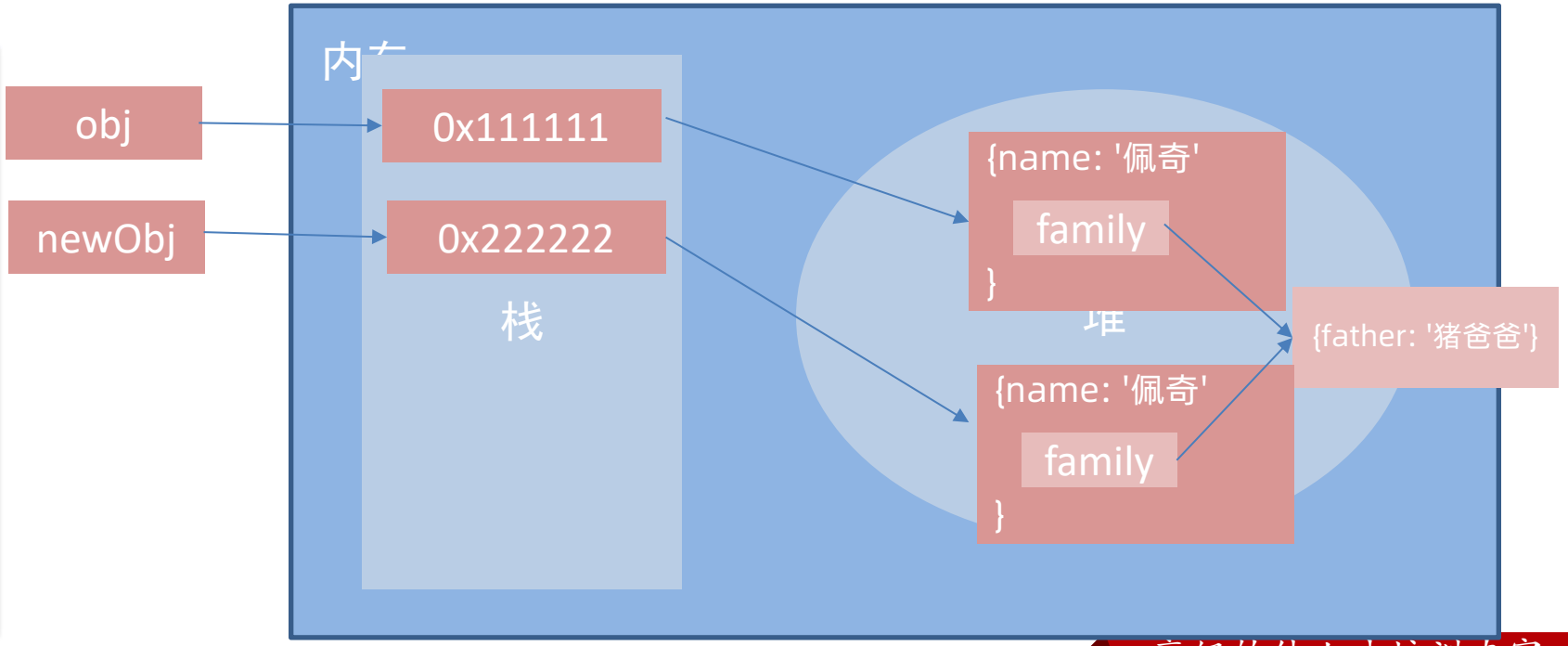
1.1 浅拷贝

浅拷贝注意：

- 如果是基本数据类型拷贝值
- 如果是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，还是会影响原来对象

```
// 浅拷贝: 引用数据类型拷贝的还是地址
const obj = {
  name: '佩奇',
  family: {
    father: '猪爸爸'
  }
}
// 浅拷贝
const newObj = { ...obj }
// 修改了新对象的名字
newObj.family.father = '猪猪'
console.log(obj.family.father) //猪猪
 // 也变成了猪猪
```



总结

1. 浅拷贝怎么理解？会有什么问题呢？
   - 拷贝给一个新的对象，拷贝单层
   - 如果是基本数据类型则拷贝值
   - 如果是引用数据类型则拷贝的还是地址，还会相互影响

深浅拷贝

- 浅拷贝
- 深拷贝

1.2 深拷贝

深拷贝：拷贝多层，不再拷贝地址

常见方法：

1. 通过 JSON 序列化实现
2. lodash库 实现
3. 通过递归实现

1.2 深拷贝

1. 通过JSON序列化实现（开发中使用较多）

JSON.stringify() 序列化为 JSON 字符串，然后再JSON.parse() 转回对象格式

```
// 深拷贝实现方式一：JSON序列化（常用的方式）
const obj = {
  name: '佩奇',
  family: {
    father: '猪爸爸'
  },
  hobby: ['跳泥坑', '唱歌']
}

const newObj = JSON.parse(JSON.stringify(obj))
```

注意事项

缺点：function 或 undefined等，在序列化过程中会被忽略

1.2 深拷贝

2. js库 lodash里面 _.cloneDeep 内部实现了深拷贝

官网地址：https://www.lodashjs.com/

```
<script src="./js/lodash.min.js"></script>
<script>
  const obj = {
    name: '佩奇',
    love: undefined,
    family: {
      father: '猪爸爸'
    },
    hobby: ['跳泥坑', '唱歌'],
    sayHi() {
      console.log('我会唱歌')
    }
  }
  // lodash 库实现
  const newObj = _.cloneDeep(obj)
  console.log(newObj)
</script>
```

1.2 深拷贝

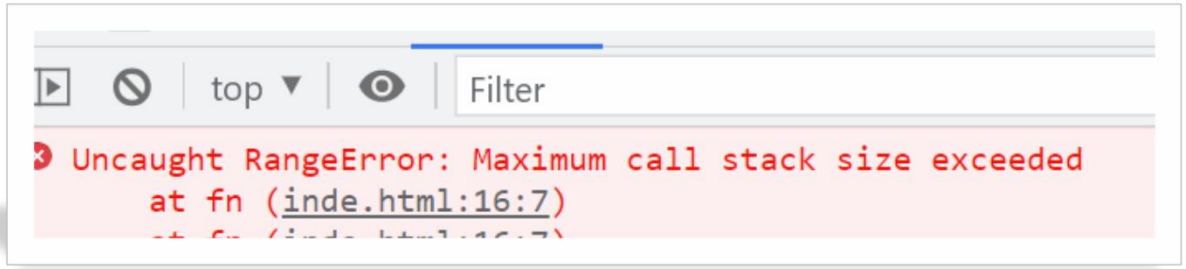
3. 通过递归实现深拷贝

递归：

所谓递归就是一种函数调用自身的操作

```
function fn() {
  // 业务逻辑
  fn()
}
```

- 简单理解:函数内部自己调用自己, 就是递归，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），所以记得添加退出条件 return



练习

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()

练习

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

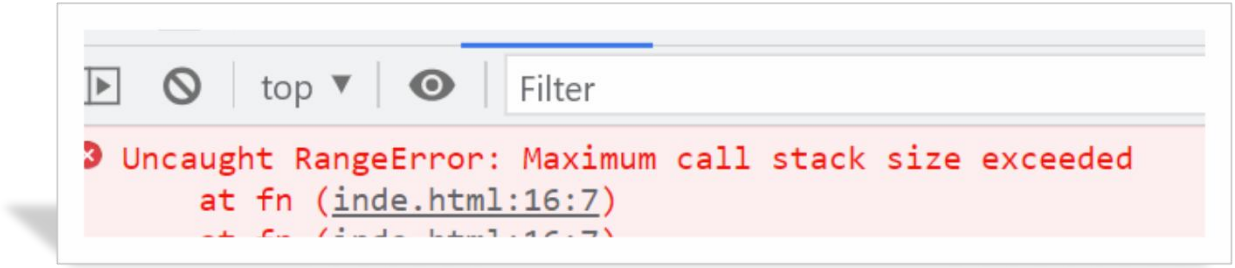
②：输出当前时间可以使用：new Date().toLocaleString()

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

总结

1. 什么是递归？有什么注意事项？

- 函数内部自己调用自己，就是递归，这个函数就是递归函数
- 递归很容易发生“栈溢出”错误（stack overflow），所以记得添加退出条件 return

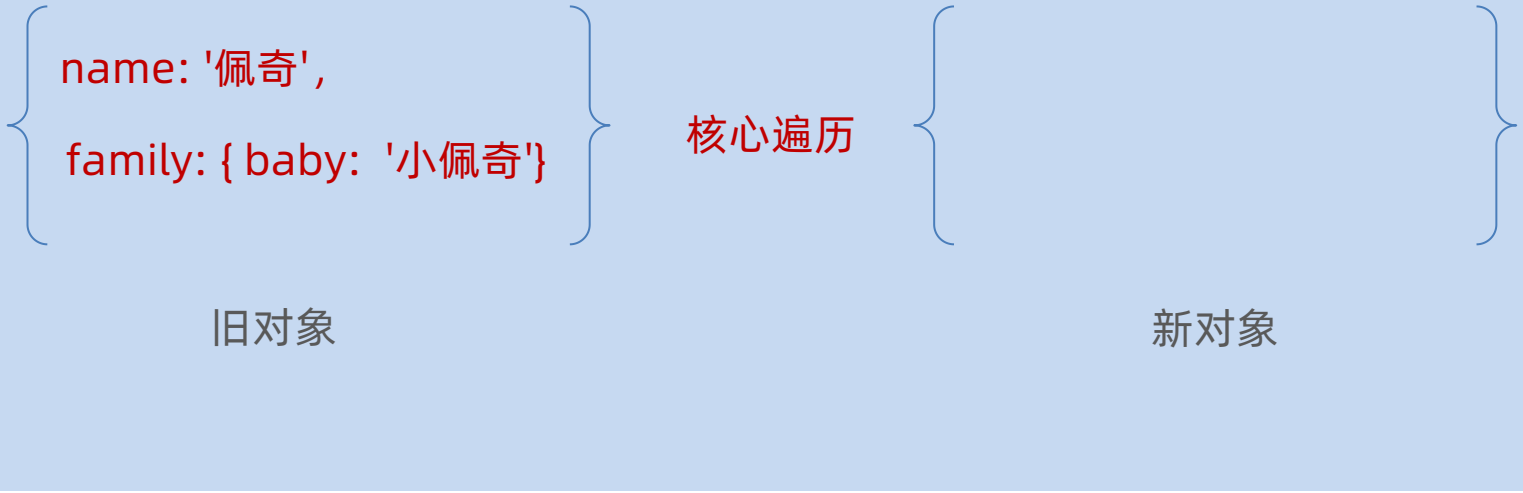


1.2 深拷贝

3. 通过递归函数实现深拷贝（简版）

深拷贝函数 cloneDeep()

先判断拷贝的是数组还是对象？



1.2 深拷贝

3. 通过递归函数实现深拷贝（简版）

深拷贝函数 cloneDeep()

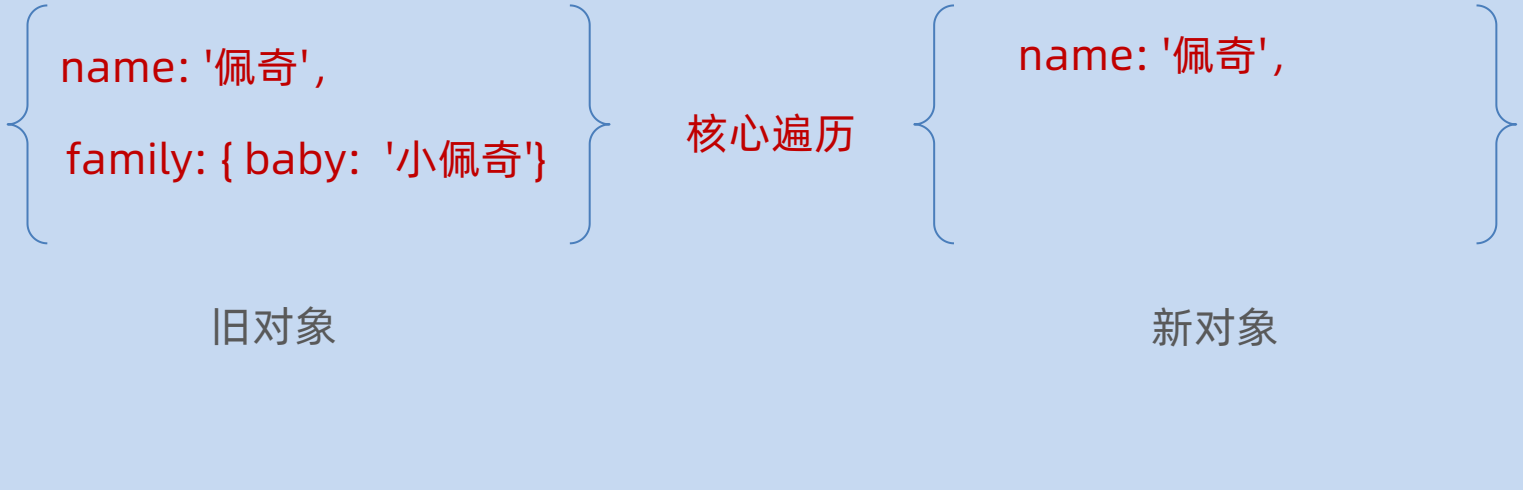
先判断拷贝的是数组还是对象？

1.2 深拷贝

3. 通过递归函数实现深拷贝（简版）

深拷贝函数 cloneDeep()

先判断拷贝的是数组还是对象？



1.2 深拷贝

3. 通过递归函数实现深拷贝（简版）

深拷贝函数 cloneDeep()

先判断拷贝的是数组还是对象？

1.2 深拷贝

3. 通过递归函数实现深拷贝（简版）

深拷贝函数 cloneDeep() ← { baby: '小佩奇'}

先判断拷贝的是数组还是对象？

1.2 深拷贝

3. 通过递归函数实现深拷贝（简版）

深拷贝函数 cloneDeep({ baby: '小佩奇'})

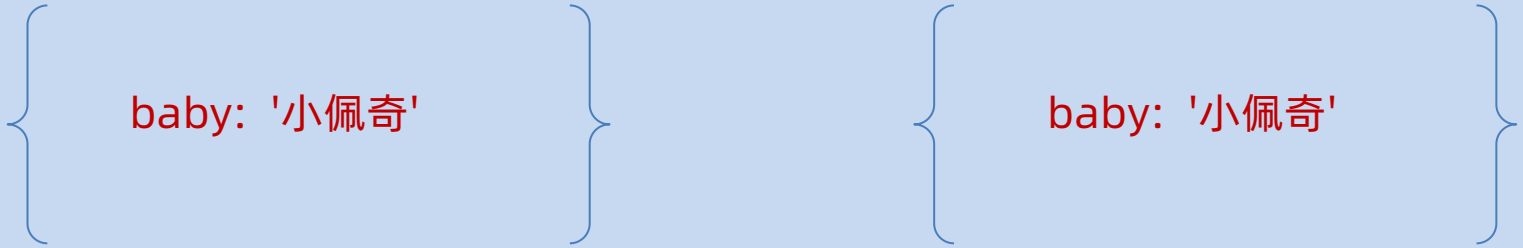
先判断拷贝的是数组还是对象？

1.2 深拷贝

3. 通过递归函数实现深拷贝（简版）

深拷贝函数 cloneDeep()

先判断拷贝的是数组还是对象？



1.2 深拷贝

3. 通过递归函数实现深拷贝（简版）

深拷贝函数 cloneDeep()

先判断拷贝的是数组还是对象？

```
{
  name: '佩奇',
  family: { baby: '小佩奇'}
}
```

旧对象

```
{
  name: '佩奇',
  family: { baby: '小佩奇'}
}
```

返回新对象

1.2 深拷贝

3. 通过递归函数实现深拷贝（简版）

深拷贝思路：

1. 深拷贝的核心是利用函数递归
2. 封装函数，里面先判断拷贝的是数组还是对象
3. 然后开始遍历
4. 如果属性值是引用数据类型（比如数组或者对象），则再次递归函数
5. 如果属性值是基本数据类型，则直接赋值即可

```
function cloneDeep(obj) {
  let newObj = Array.isArray(obj) ? [] : {}
  for (let key in obj) {
    if (typeof obj[key] === 'object') {
      newObj[key] = cloneDeep(obj[key])
    } else {
      newObj[key] = obj[key]
    }
  }
  return newObj
}
const o = cloneDeep(obj)
```

总结

1. 实现深拷贝三种方式？
   - 利用JSON序列化（常用）
   - 利用js库 lodash里面的 _.cloneDeep()（常用）
   - 自己利用递归函数书写深拷贝（理解）

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

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

目录

Contents

异常处理

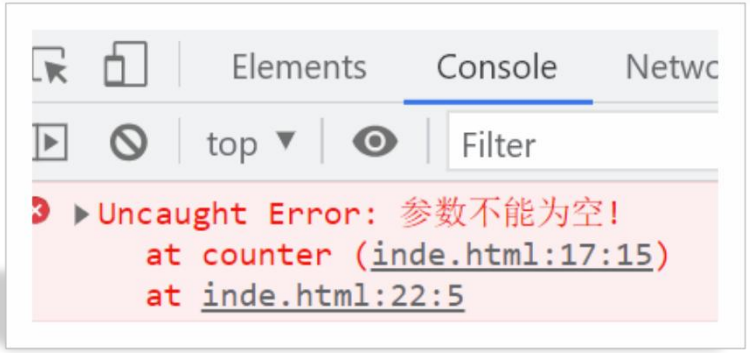
- throw 抛异常
- try /catch 捕获异常
- debugger

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()
```



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

总结

1. 抛出异常我们用哪个关键字？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()
```

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

我们想要测试某些代码是否有异常，可以通过try / catch 捕获错误信息（浏览器提供的错误信息）

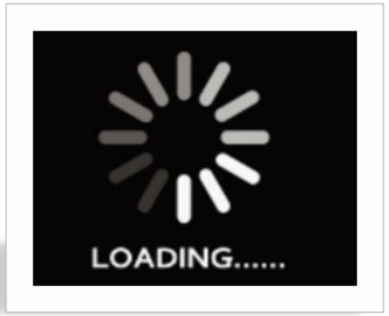
try 试试　catch 拦住　finally　最后

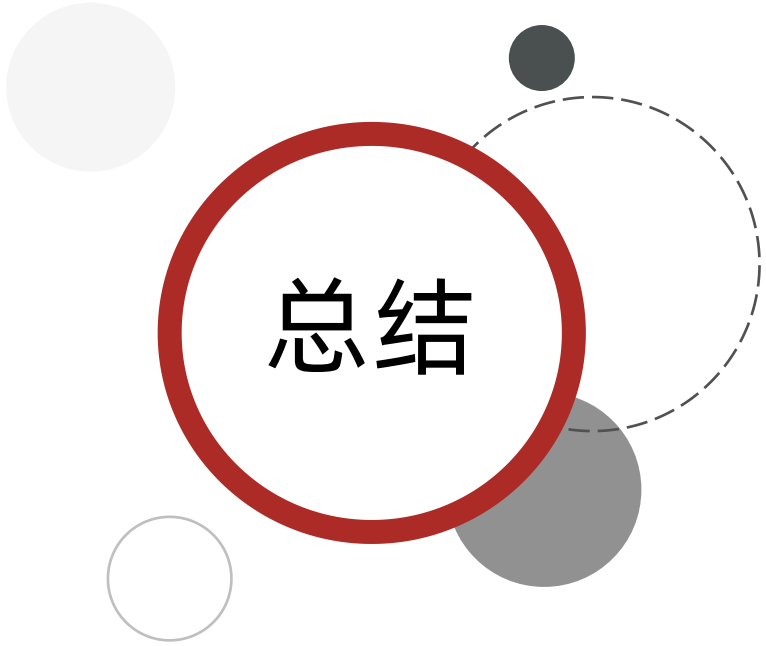
```
try {
  // 可能出现错误的代码
  const p = document.querySelector('.p')
  p.style.color = 'red'
} catch (error) {
  // try出现错误会进入 catch并捕获异常  error 错误参数
  console.log(error) // 错误信息

} finally {
  // 不管有么有错误都会进入 finally
  console.log('不管有没有错误都会执行')
}
```

说明：

1. 将预估可能发生错误的代码写在 try 代码段中
2. 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
3. finally　不管是否有错误，都会执行





1. 捕获异常我们用那3个关键字？可能会出现的错误代码写到谁里面
   - try / catch / finally
   - try
2. 怎么调用错误信息？
   - 利用 catch 的参数， 比如 error
3. finally 什么时候执行呢？
   - 不管有没有错误都会执行
   - 比如页面加载的loading动画，不管页面加载有没有成功，时间到了都会消失

异常处理

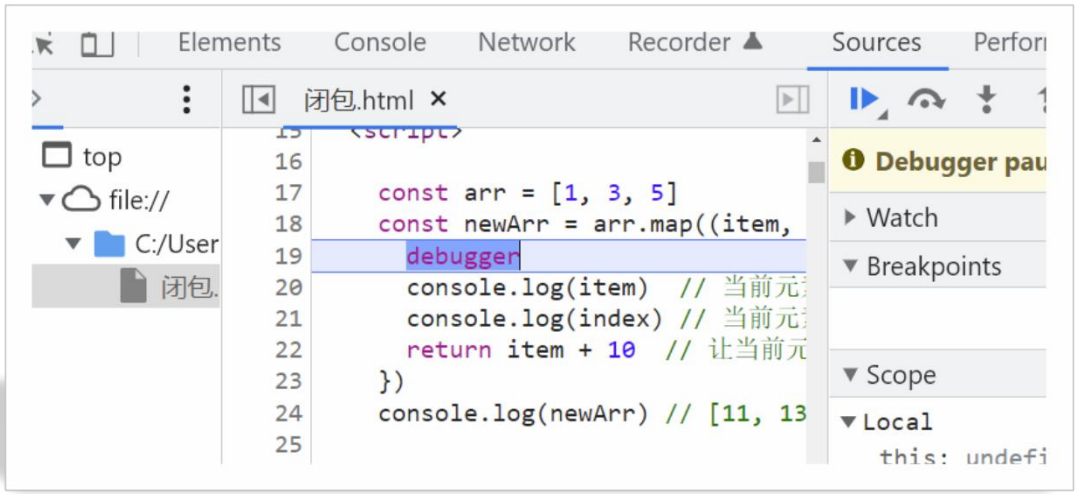
- throw 抛异常
- try /catch 捕获异常
- debugger

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

debugger 语句调用调试功能，例如设置断点

```
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]
```



目录

Contents

处理this

- 改变this
- this指向

3.1 改变this

JavaScript 中允许指定（改变）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()

3.1 改变this

1. call() -了解

使用 call 方法调用函数，同时指定被调用函数中 this 的指向

- **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
- 使用场景： Object.prototype.toString.call(数据) 检测数据类型


```
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的应用场景是？
   - Object.prototype.toString.call(数据) 检测数据类型

3.1 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()

3.1 改变this

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])
```

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

- 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
- 使用场景： apply 主要跟数组有关系，比如使用 Math.max() 求数组的最大值

3.1 改变this

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，call是传递参数列表，apply传递的必须是数组
   - call可以用来检测数据类型，apply可以求数组最大值

3.2 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()

3.1 改变this

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）
- 使用场景： 当我们只是想改变 this 指向，并且不想调用这个函数时，可以使用 bind，比如改变定时器内部的this指向.

call apply bind 总结

方法	相同点	传递参数	是否调用函数	使用场景
call	改变this指向	传递参数列表 arg1, arg2...	调用函数	Object.prototype.toString.call() 检测数据类型
apply	改变this指向	参数是数组	调用函数	跟数组相关，比如求数组最大值和最小值等
bind	改变this指向	传递参数列表 arg1, arg2...	不调用函数	改变定时器内部的this指向

处理this

- 改变this
- this指向

3.2 this指向-普通函数

- this的取值 不取决于函数的定义，而是取决于怎么调用的（this指向**调用者**）
  - 全局内调用： fn() 指向window
  - 对象内的方法调用：obj.fn() 指向调用对象
  - 构造函数调用：new Person() 指向实例对象
  - 事件处理函数中调用：指向当前触发事件的DOM元素
  - 特殊调用 比如 call、apply、bind可以改变this指向，fun.call(obj) 指向 obj

3.2 this指向-箭头函数

箭头函数本身没有this，而是沿用上层函数作用域(非箭头函数)或者全局作用域的this

目录

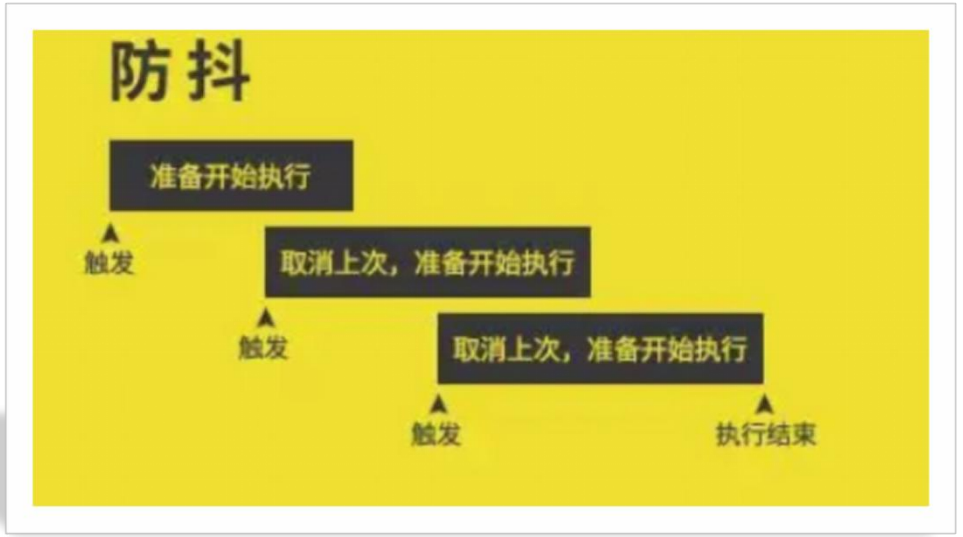
Contents

性能优化

- 防抖
- 节流

4.1 防抖（debounce）

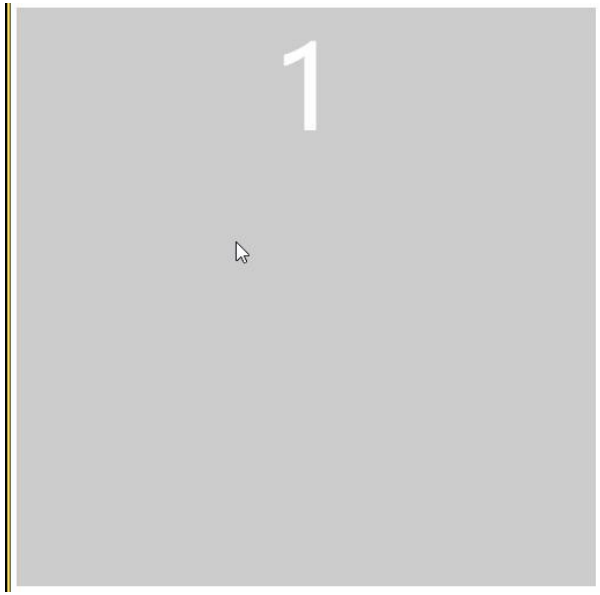
- **防抖**：单位时间内，频繁触发事件，只执行最后一次
- **举个栗子**：王者荣耀回城，只要被打断就需要重新来
- **使用场景**：
  - 搜索框搜索输入。只需用户最后一次输入完，再发送请求
  - 手机号、邮箱验证输入检测



案例

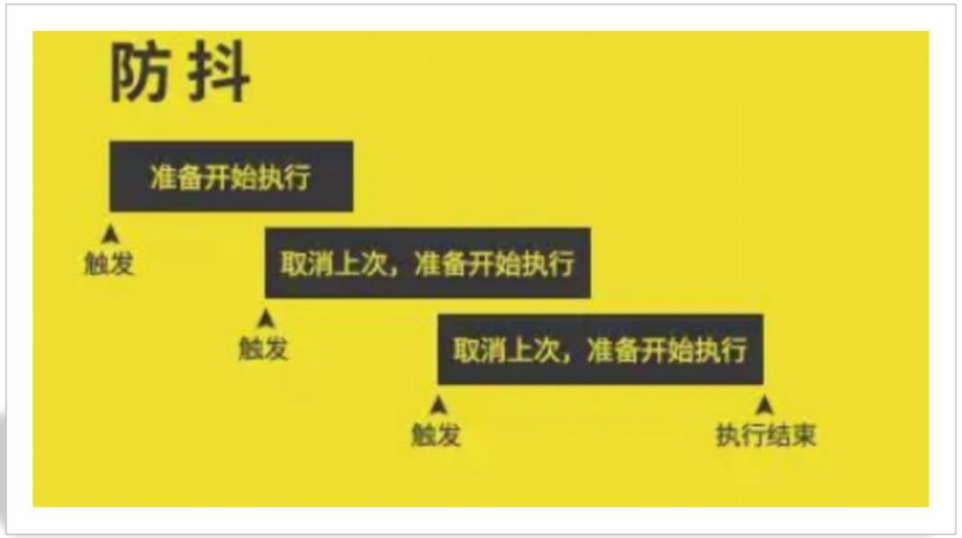
利用防抖来处理-鼠标滑过盒子显示文字

要求： 鼠标在盒子上移动，里面的数字就会变化+1



总结

1. 防抖是什么？
   - 单位时间内，频繁触发事件，只执行最后一次
2. 有什么使用场景呢？
   - 搜索框搜索输入。只需用户最后一次输入完，再发送请求
   - 手机号、邮箱验证输入检测



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500ms之后，里面的数字才会变化+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
}
box.addEventListener('mousemove', mouseMove)
```

实现方式：

1. lodash 提供的防抖来处理
2. 手写一个防抖函数来处理

利用 Lodash 库 实现防抖

```
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))
```

案例

利用防抖来处理-鼠标滑过盒子显示文字（手写防抖函数）

要求： 鼠标在盒子上移动，鼠标停止500ms之后，里面的数字才会变化+1

核心思路：

防抖的核心就是利用定时器 (setTimeout) 来实现

①：声明一个定时器变量

②：当鼠标每次滑动都先判断是否有定时器了，如果有定时器先清除以前的定时器

③：如果没有定时器则开启定时器，记得存到变量里面

④：在定时器里面调用要执行的函数

案例

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))
```

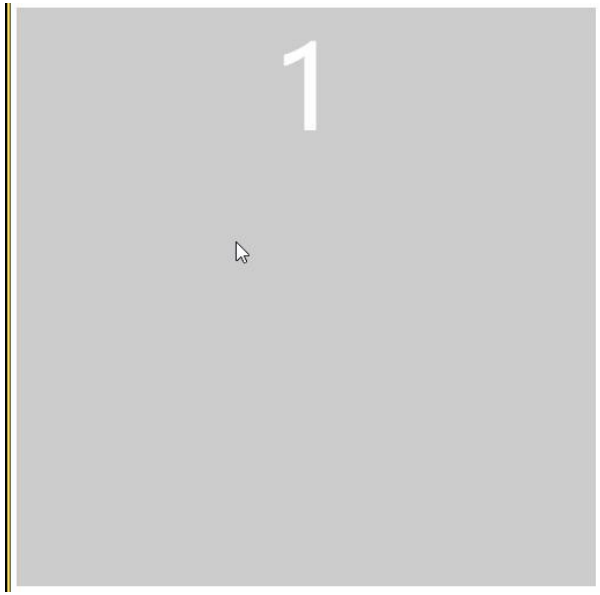
4.2 节流 - throttle

- **节流：**单位时间内，频繁触发事件，只执行一次
- **举个栗子：**
  - 王者荣耀技能冷却，期间无法继续释放技能
  - 和平精英 98k 换子弹期间不能射击
- **使用场景：**
  - 高频事件:鼠标移动 mousemove、页面尺寸缩放 resize、滚动条滚动scroll 等等

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



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，不管移动多少次，每隔500ms才 + 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
}
box.addEventListener('mousemove', mouseMove)
```

实现方式：

1. lodash 提供的节流函数来处理

2. 手写一个节流函数来处理

总结

1. 节流是什么？
   - 单位时间内，频繁触发事件，只执行一次
   - 简单理解：在500ms内，不管触发多少次事件，只执行一次
2. 有什么使用场景呢？
   - 高频事件:鼠标移动 mousemove、页面尺寸缩放 resize、滚动条滚动scroll

案例

利用节流来处理-鼠标滑过盒子显示文字（手写节流函数）

要求： 鼠标在盒子上移动，不管移动多少次，每隔500ms才 + 1

核心思路：

节流的核心就是利用定时器 (setTimeout) 来实现

①：声明一个定时器变量

②：当鼠标每次滑动都先判断是否有定时器了，如果有定时器则不开启新定时器

③：如果没有定时器则开启定时器，记得存到变量里面

- 定时器里面调用执行的函数
- 定时器里面要把定时器清空

注意：在 setTimeout 中是无法删除定时器的，因为定时器还在运作，所以使用 timer = null 而不是 clearTimeout(timer)

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，不管滑动多少次，500ms之内加+

```
function throttle(fn, t) {
  let timer = null
  return function () {
    if (!timer) {
      timer = setTimeout(function () {
        fn()
        // 清空定时器
        timer = null
      }, t)
    }
  }
}

box.addEventListener('mousemove', throttle(mouseMove, 3000))
```

防抖和节流总结

性能优化	说明	使用场景
防抖	单位时间内，频繁触发事件，只执行最后一次	搜索框搜索输入、手机号、邮箱验证输入检测
节流	单位时间内，频繁触发事件，只执行一次	高频事件:鼠标移动 mousemove、页面尺寸缩放 resize、滚动条滚动scroll 等等

Lodash 库 实现节流和防抖

```
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', _.throttle(mouseMove, 1000))
```

```
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))
```

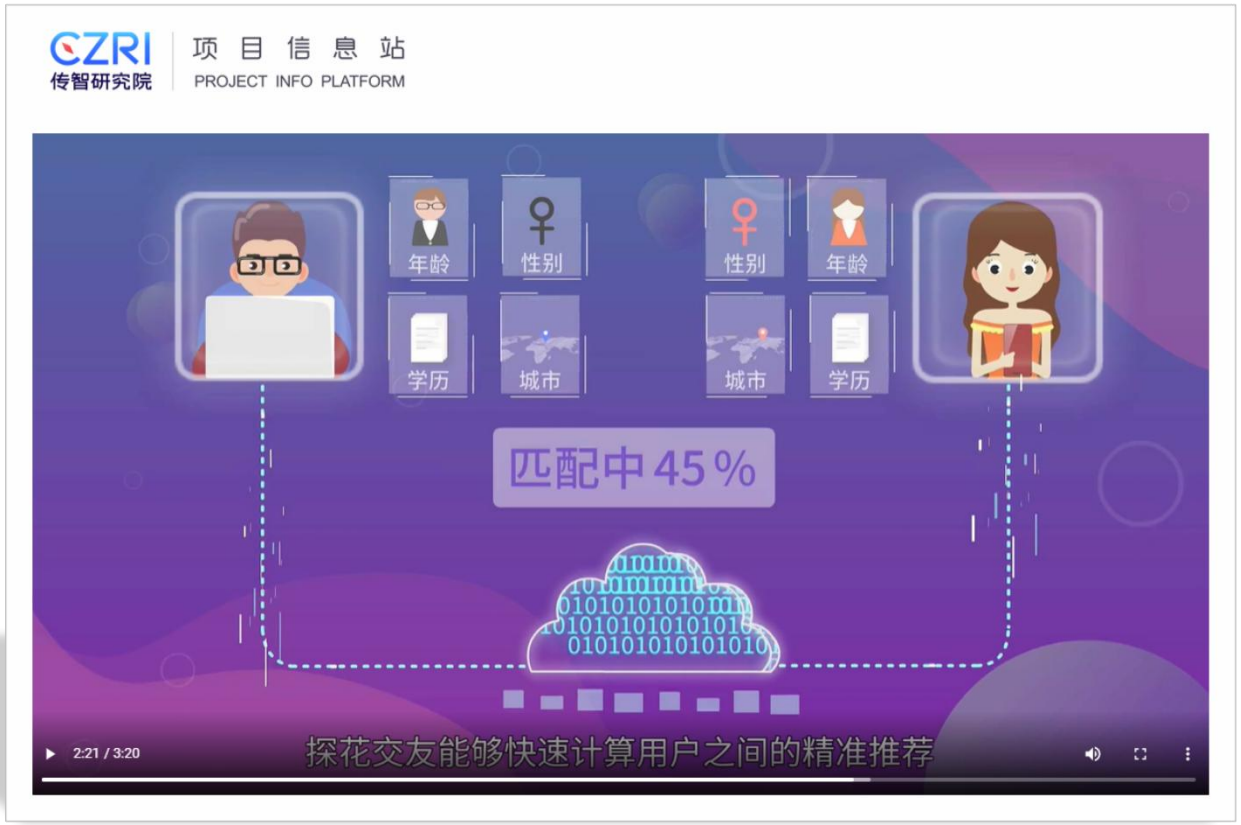
目录

Contents

案例

项目库页面效果

业务分析：



- 视频记录播放位置
- 电梯导航激活效果

案例 **项目库页面效果**

视频记录播放位置

分析：

记录当前时间：

①：timeupdate 事件在视频/音频（audio/video）当前的播放位置发生改变时触发

②：把当前时间（视频.currentTime） 持久化到本地

③： timeupdate 触发频次太高了，我们可以设定 1秒钟触发一次（节流）

案例 **项目库页面效果**

视频记录播放位置

分析：

页面打开视频播放到上一次记录的位置：

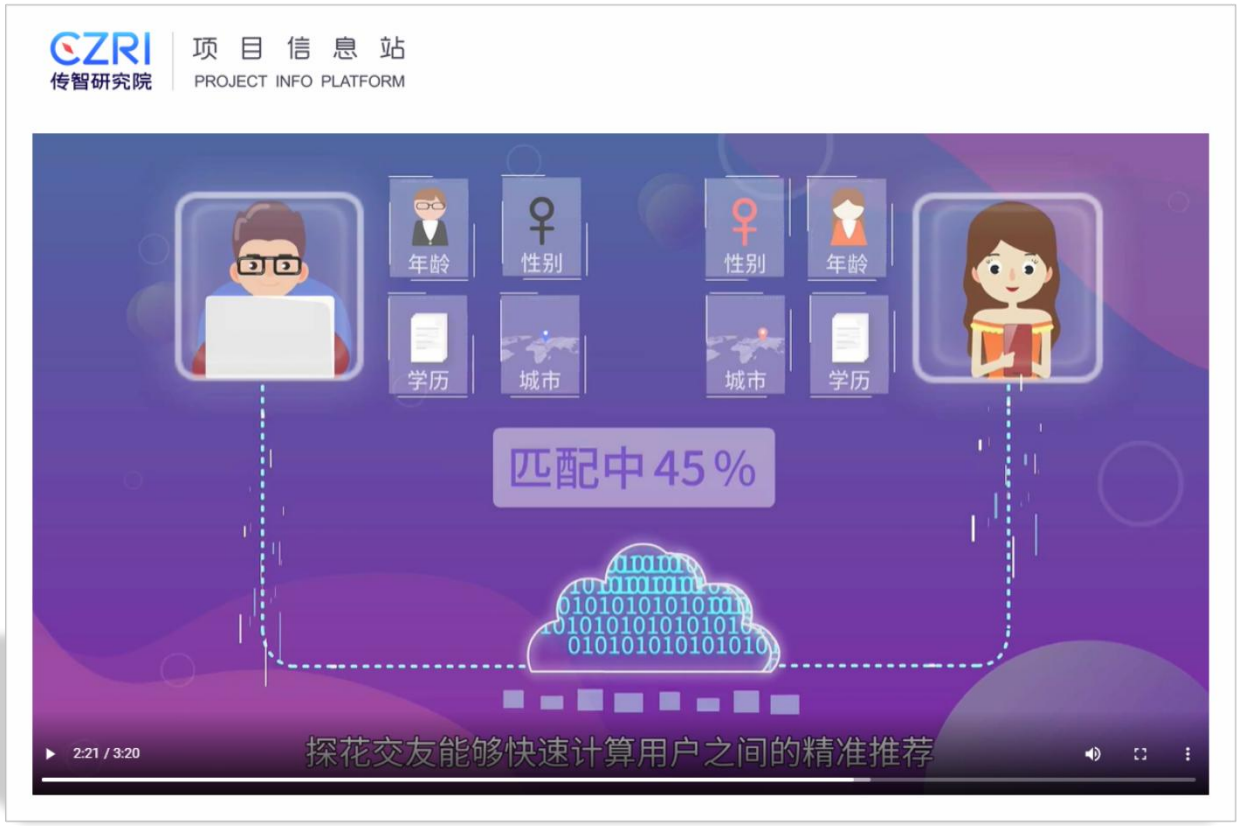
①：loadeddata 事件：视频当前播放位置的视频帧（通常是第一帧）加载完成后触发

②：把视频的当前时间改为本地存储（持久化）的时间，如果没有数据就默认为0

案例

项目库页面效果

业务分析：



- 视频记录播放位置
- 电梯导航激活效果

案例 **页面打开，可以记录上一次的视频播放位置**

电梯导航激活效果

思路：

1. 每次滚动1px都会触发处理函数频率太高了，我们可以设定 200ms 内 只执行一次（节流）

传智教育旗下高端IT教育品牌