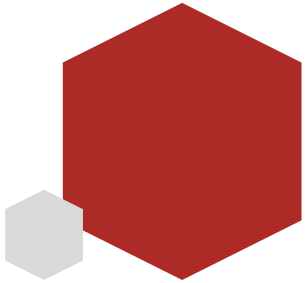
JS原理课

黑马程序员
www.itheima.com

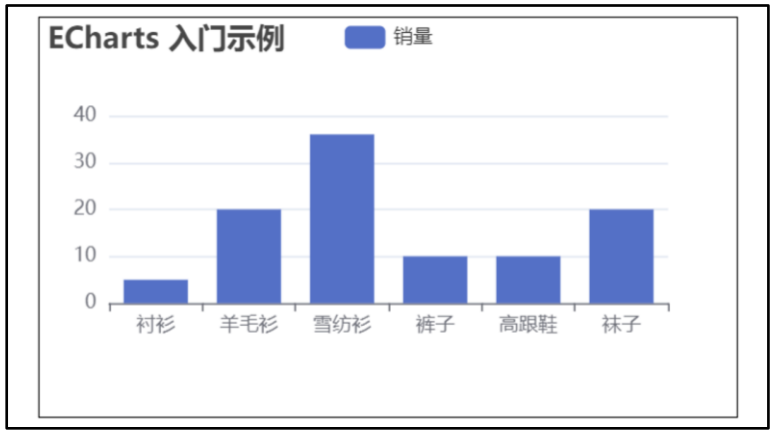
传智教育旗下
高端IT教育品牌

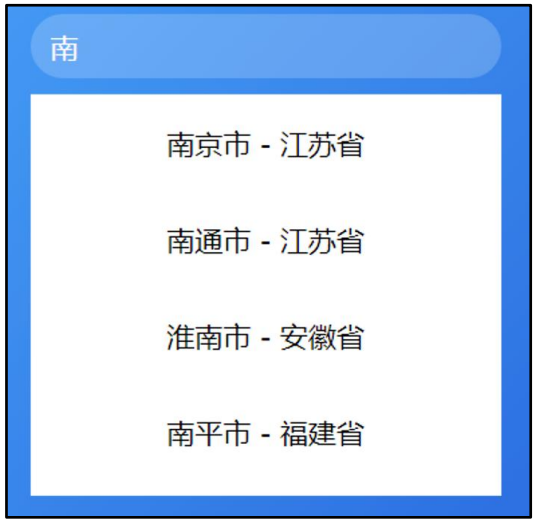


性能优化-防抖

防抖

常见的前端性能优化方案，它可以防止JS高频渲染页面时出现的视觉抖动(卡顿)





适用场景:在触发频率高的事件中，执行耗费性能操作，连续操作之后只有最后一次生效

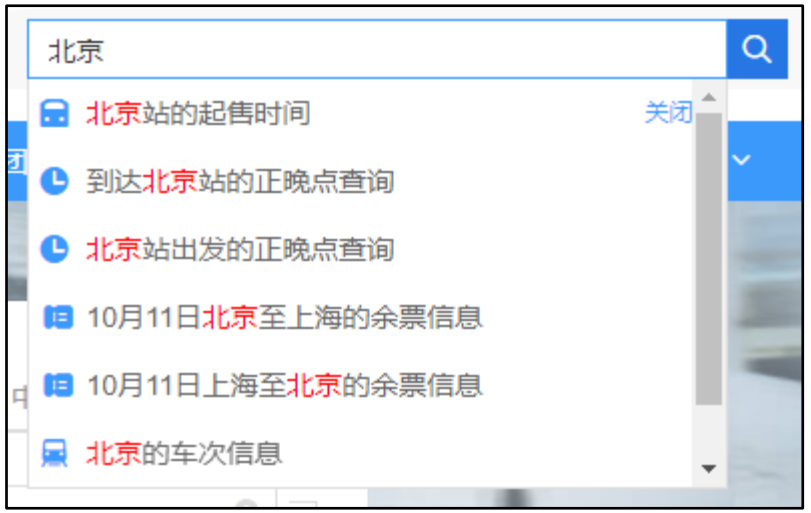
1. 频率高的事件: resize、input 、scroll 、keyup....
2. 耗费性能的操作: 操纵页面、网络请求....

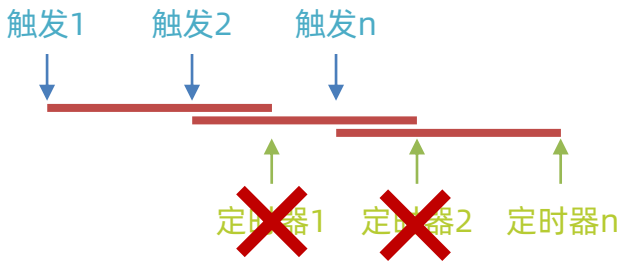
防抖

连续事件停止触发后,一段时间内没有再次触发,就执行业务代码

核心步骤:

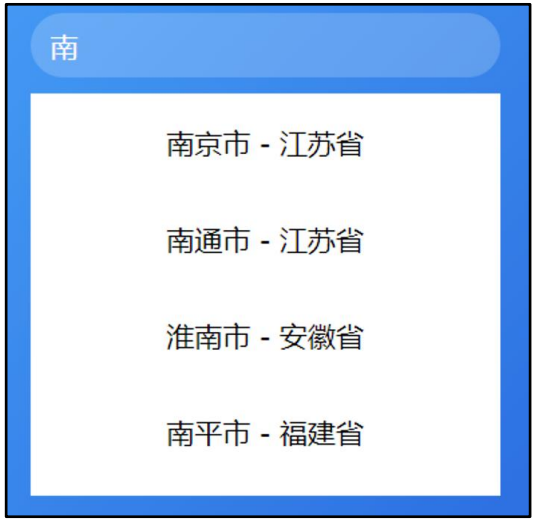
1. 开启定时器,保存定时器id
2. 清除已开启的定时器

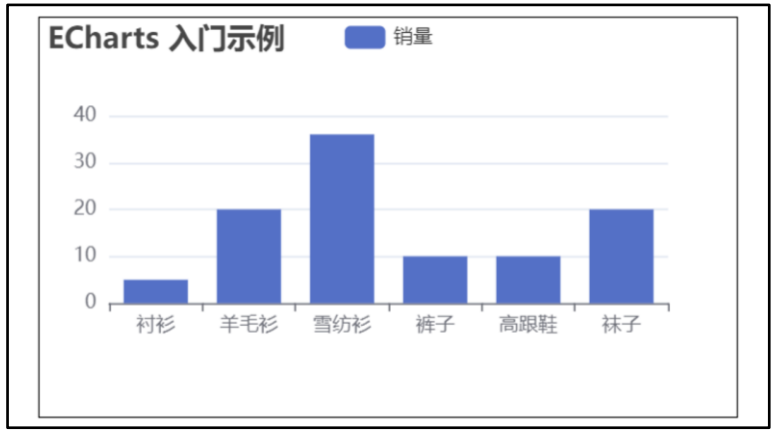




防抖工具函数debounce

1. lodash工具库中的debounce
2. debounce的实现原理





lodash工具库中的debounce

```
_.debounce(func, [wait=0], [options=])
```

（原函数，防抖延时时间，其他配置）→ 防抖新函数

小建议： 如果项目使用了lodash优先使用它的debounce方法，它可以支持 this 和 参数

debounce的实现原理

1. 返回防抖动的新函数
2. 原函数中的this可以正常使用
3. 原函数中的参数可以正常使用

```
const func = function (e) {
  console.log('e:', e)
  renderCity(this.value)
}

function debounce(func, wait = 0) {
  // TODO









}

const deFunc = debounce(func, 500)

document.querySelector('.search-city').addEventListener('input', deFunc)
```

debounce的实现原理

1. 返回防抖动的新函数
2. 原函数中的this可以正常使用
3. 原函数中的参数可以正常使用

```
const func = function (e) {
  console.log('e:', e)
  renderCity(this.value)
}

function debounce(func, wait = 0) {
  let timeId
  return function () {
    clearTimeout(timeId)
    timeId = setTimeout(function () {
      func()
    }, wait)
  }

}

const deFunc = debounce(func, 500)

document.querySelector('.search-city').addEventListener('input', deFunc)
```

debounce的实现原理

1. 返回防抖动的新函数
2. 原函数中的this可以正常使用
3. 原函数中的参数可以正常使用

```
const func = function (e) {
  console.log('e:', e)
  renderCity(this.value)
}

function debounce(func, wait = 0) {
  let timeId
  return function () {
    let _this = this
    clearTimeout(timeId)
    timeId = setTimeout(function () {
      func.apply(_this)
    }, wait)
  }
}

const deFunc = debounce(func, 500)

document.querySelector('.search-city').addEventListener('input', deFunc)
```

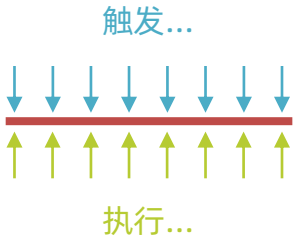
debounce的实现原理

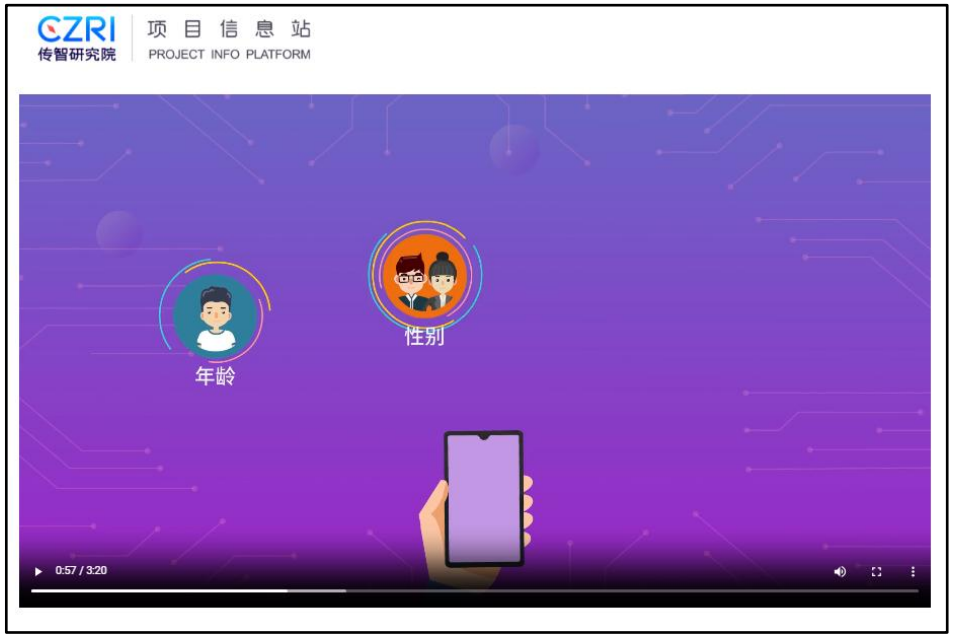
1. 返回防抖动的新函数
2. 原函数中的this可以正常使用
3. 原函数中的参数可以正常使用

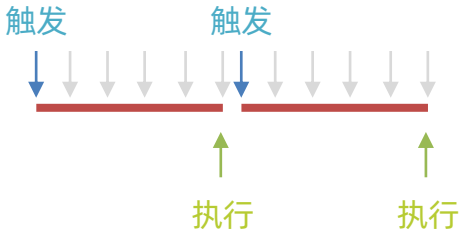
```
const func = function (e) {
  console.log('e:', e)
  renderCity(this.value)
}

function debounce(func, wait = 0) {
  let timeId
  return function (...args) {
    let _this = this
    clearTimeout(timeId)
    timeId = setTimeout(function () {
      func.apply(_this, args)
    }, wait)
  }
}

const deFunc = debounce(func, 500)

document.querySelector('.search-city').addEventListener('input', deFunc)
```

节流

常见的前端性能优化方案，它可以防止高频触发事件造成的性能浪费







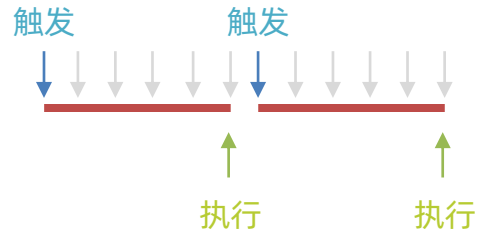
适用场景:在触发频率高的事件中，执行耗费性能操作，连续触发，单位时间内只有一次生效

节流

在触发频率高的事件中，执行耗费性能操作，连续触发，单位时间内只有一次生效

核心步骤：

1. 开启定时器,并保存 id
2. 判断是否已开启定时器
3. 定时器执行时，id设置为空

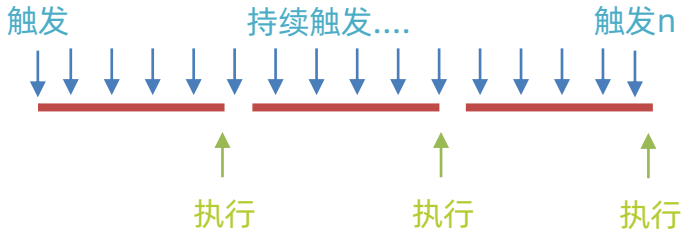


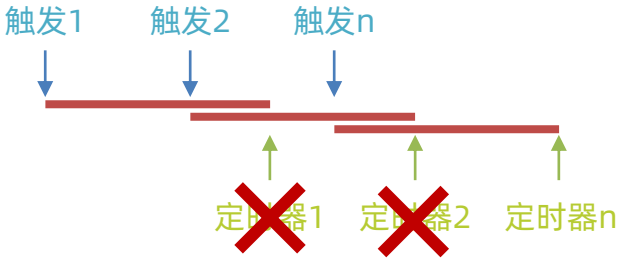
防抖

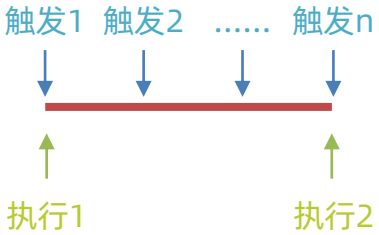
连续事件触发时,单位时间内只有一次生效

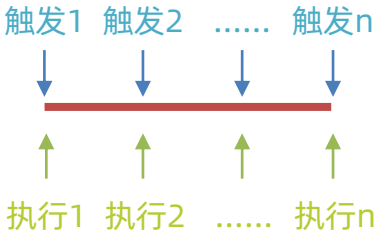
核心步骤：

1. 开启定时器,保存定时器id
2. 清除已开启的定时器









节流工具函数throttle

1. lodash工具库中的throttle
2. throttle的实现原理

lodash工具库中的throttle

```
_.throttle(func, [wait=0], [options=])
```

（原函数，节流时间，其他配置）→ 节流新函数

options.leading: 节流开始前是否调用,默认为true

小建议: 如果项目使用了lodash优先使用它的throttle方法，它可以支持 this 和 参数

节流工具函数throttle

1. lodash工具库中的throttle
2. throttle的实现原理

throttle的实现原理

1. 返回节流的新函数
2. 原函数中的this可以正常使用
3. 原函数中的参数可以正常使用

```
const func = function (e) {
  console.log('timeupdate触发')
  console.log('e:', e)
  localStorage.setItem('currentTime', this.currentTime)
}

function throttle(func, wait = 0) {
  // TODO









}

const throttleFn = throttle(func, 1000)

video.addEventListener('timeupdate', throttleFn)
```

throttle的实现原理

1. 返回节流的新函数
2. 原函数中的this可以正常使用
3. 原函数中的参数可以正常使用

```
const func = function (e) {
  console.log('timeupdate触发')
  console.log('e:', e)
  localStorage.setItem('currentTime', this.currentTime)
}

function throttle(func, wait = 0) {
  let timeId
  return function () {
    if (timeId !== undefined) {
      return
    }
    timeId = setTimeout(() => {
      func()
      timeId = undefined
    }, wait)
  }

}

const throttleFn = throttle(func, 1000)

video.addEventListener('timeupdate', throttleFn)
```

throttle的实现原理

1. 返回节流的新函数
2. 原函数中的this可以正常使用
3. 原函数中的参数可以正常使用

```
const func = function (e) {
  console.log('timeupdate触发')
  console.log('e:', e)
  localStorage.setItem('currentTime', this.currentTime)
}

function throttle(func, wait = 0) {
  let timeId
  return function () {
    if (timeId !== undefined) {
      return
    }
    const _this = this
    timeId = setTimeout(() => {
      func.apply(_this)
      timeId = undefined
    }, wait)
  }
}

const throttleFn = throttle(func, 1000)

video.addEventListener('timeupdate', throttleFn)
```

throttle的实现原理

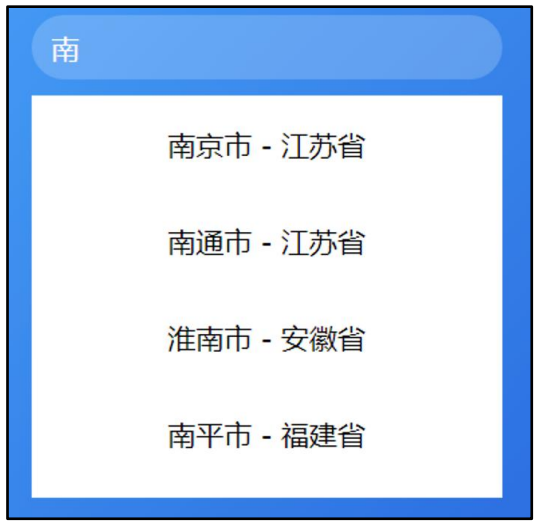
1. 返回节流的新函数
2. 原函数中的this可以正常使用
3. 原函数中的参数可以正常使用

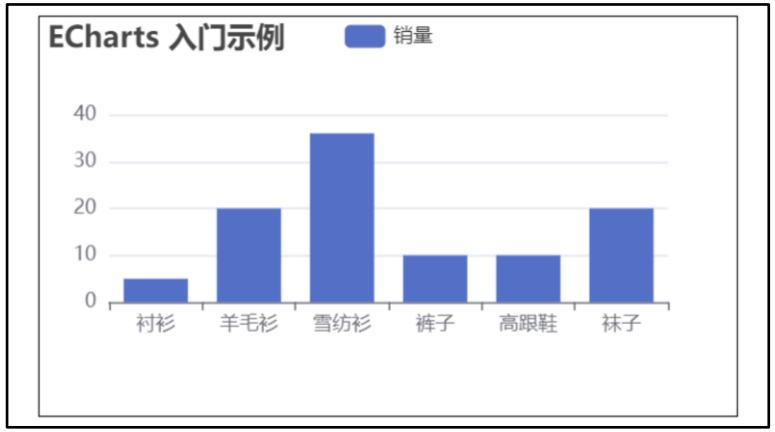
```
const func = function (e) {
  console.log('timeupdate触发')
  console.log('e:', e)
  localStorage.setItem('currentTime', this.currentTime)
}

function throttle(func, wait = 0) {
  let timeId
  return function (...args) {
    if (timeId !== undefined) {
      return
    }
    const _this = this
    timeId = setTimeout(() => {
      func.apply(_this, args)
      timeId = undefined
    }, wait)
  }
}

const throttleFn = throttle(func, 1000)

video.addEventListener('timeupdate', throttleFn)
```

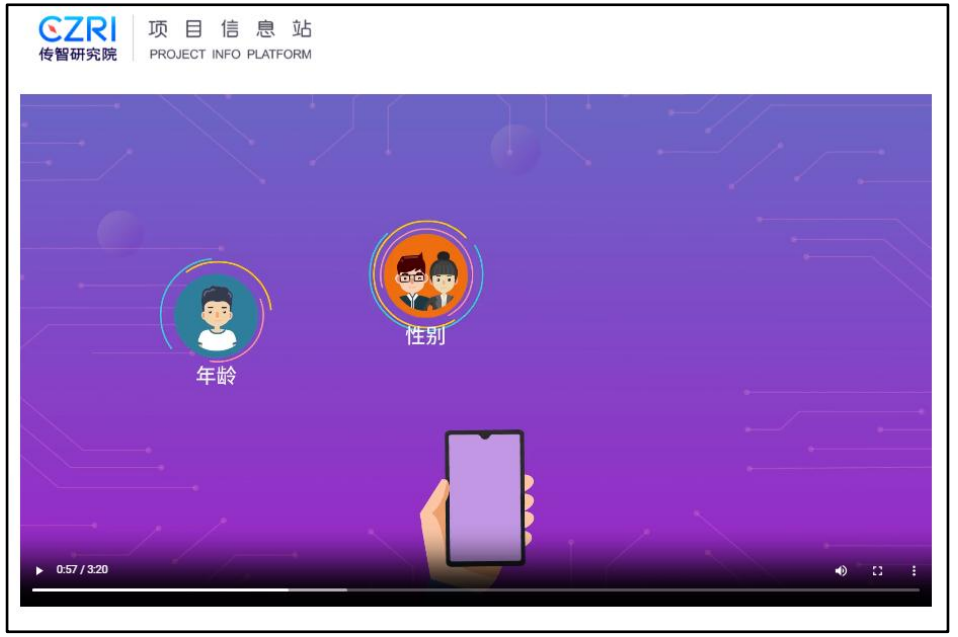
防抖和节流

防抖: 常见的前端性能优化方案，它可以防止JS高频渲染页面时出现的视觉抖动(卡顿)

节流: 常见的前端性能优化方案，它可以防止高频触发事件造成的性能浪费





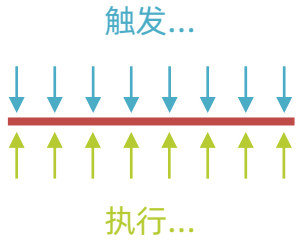


防抖和节流的适用场景

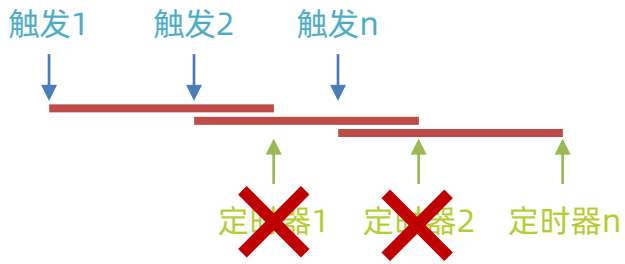
防抖：在触发频率高的事件中，执行耗费性能操作，连续操作之后只有最后一次生效

节流：在触发频率高的事件中，执行耗费性能操作，连续触发，单位时间内只有一次生效

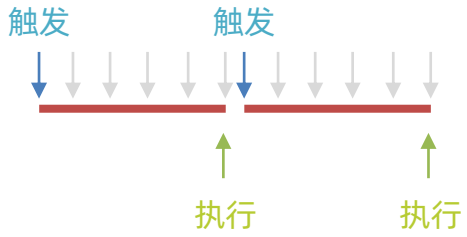
未优化



防抖



节流



lodash工具库中的防抖(debounce)和节流(throttle)

```
_.debounce(func, [wait=0], [options=])
```

（原函数，防抖延时时间，其他配置）→ 防抖新函数

```
_.throttle(func, [wait=0], [options=])
```

（原函数，节流时间，其他配置）→ 节流新函数

options.leading: 节流开始前是否调用,默认为true

小建议: 如果项目使用了lodash优先使用它的debounce或throttle方法，它可以支持 this 和 参数

手写实现debounce和throttle函数

1. 返回 防抖 or 节流 的新函数
2. 原函数中的this可以正常使用
3. 原函数中的参数可以正常使用

```
// 防抖工具函数
function debounce(func, wait = 0) {
  let timeId
  return function (...args) {
    let _this = this
    clearTimeout(timeId)
    timeId = setTimeout(function () {
      func.apply(_this, args)
    }, wait)
  }
}

// 节流工具函数
function throttle(func, wait = 0) {
  let timeId
  return function (...args) {
    if (timeId !== undefined) {
      return
    }
    const _this = this
    timeId = setTimeout(() => {
      func.apply(_this, args)
      timeId = undefined
    }, wait)
  }
}
```