

9个移动端触摸事件优化,响应速度提升95%

原创 story JavaScript 2025年01月03日 09:52 湖北

移动端触摸事件的处理直接影响着应用的响应速度和交互体验，分享下这几年使用频率较高的几个优化策略，可以将触摸事件的响应速度显著提高，让你的应用如丝般顺滑！

1. 使用 passive: true 优化滚动性能

在移动端，滚动是最常见的交互行为之一。默认情况下，浏览器需要等待 touchstart 和 touchmove 事件的处理完成，才能执行滚动，这会导致明显的延迟。通过添加 passive 标志，我们可以告诉浏览器无需等待事件处理即可执行滚动：

```
window.addEventListener('touchstart', function(e) {  
    // 处理触摸事件  
}, { passive: true });  
  
window.addEventListener('touchmove', function(e) {  
    // 处理触摸移动  
}, { passive: true });
```

这个优化可以让滚动响应提升很大，特别是在长列表场景下效果显著。

2. 实现事件委托优化性能

对于包含大量子元素的列表，如果每个元素都单独绑定事件处理器，会导致内存占用过高和性能下降。使用事件委托可以显著改善这个问题：

```
// 优化前  
listItems.forEach(item => {  
    item.addEventListener('touchstart', handleTouch);  
});  
  
// 优化后  
parentElement.addEventListener('touchstart', function(e) {  
    const targetItem = e.target.closest('.list-item');  
    if (targetItem) {  
        handleTouch(e, targetItem);  
    }  
});
```

事件委托可以减少 90% 以上的事件监听器，同时提供相同的功能。

3. 防止滚动穿透

在移动端弹窗场景中，经常会遇到滚动穿透问题。当用户在弹窗上滑动时，背景内容也会跟着滚动，这严重影响了用户体验。以下是一个完整的解决方案：

```
const body = document.body;
let scrollTop = 0;

function preventScroll() {
  scrollTop = window.pageYOffset;
  body.style.overflow = 'hidden';
  body.style.position = 'fixed';
  body.style.top = `-${scrollTop}px`;
  body.style.width = '100%';
}

function allowScroll() {
  body.style.removeProperty('overflow');
  body.style.removeProperty('position');
  body.style.removeProperty('top');
  body.style.removeProperty('width');
  window.scrollTo(0, scrollTop);
}
```

4. 使用 requestAnimationFrame 优化动画效果

在处理触摸移动相关的动画时，使用 requestAnimationFrame 可以确保动画流畅且性能优异：

```

let ticking = false;
let lastKnownScrollPosition = 0;

document.addEventListener('scroll', function(e) {
    lastKnownScrollPosition = window.scrollY;

    if (!ticking) {
        window.requestAnimationFrame(function() {
            doSomething(lastKnownScrollPosition);
            ticking = false;
        });

        ticking = true;
    }
});

```

5. 触摸事件节流优化

对于频繁触发的触摸事件，使用节流可以显著减少事件处理次数，提升性能：

```

function throttle(func, limit) {
    let inThrottle;
    return function(...args) {
        if (!inThrottle) {
            func.apply(this, args);
            inThrottle = true;
            setTimeout(() => inThrottle = false, limit);
        }
    }
}

const throttledTouchMove = throttle(function(e) {
    // 处理触摸移动
}, 16); // 约60fps

element.addEventListener('touchmove', throttledTouchMove);

```

6. 使用硬件加速提升渲染性能

通过触发硬件加速，可以让滚动和动画更加流畅：

```
.smooth-scroll {  
  transform: translateZ(0);  
  will-change: transform;  
  backface-visibility: hidden;  
}
```

但需要注意的是，过度使用硬件加速可能导致内存占用增加，应当谨慎使用。

7. 优化点击延迟

移动端浏览器存在 300ms 的点击延迟，这是为了区分双击和单击。我们可以通过以下方式消除这个延迟：

```
// 方案一：使用 FastClick 库  
if ('addEventListener' in document) {  
  document.addEventListener('DOMContentLoaded', function() {  
    FastClick.attach(document.body);  
  }, false);  
}  
  
// 方案二：使用 CSS 属性（推荐）  
html {  
  touch-action: manipulation;  
}
```

8. 实现自定义触摸手势

对于复杂的手势交互，可以实现自定义手势识别器：

```

class TouchGestureDetector {
  constructor(element) {
    this.element = element;
    this.startX = 0;
    this.startY = 0;
    this.currentX = 0;
    this.currentY = 0;
    this.touching = false;

    this.onTouchStart = this.onTouchStart.bind(this);
    this.onTouchMove = this.onTouchMove.bind(this);
    this.onTouchEnd = this.onTouchEnd.bind(this);

    this.init();
  }

  init() {
    this.element.addEventListener('touchstart', this.onTouchStart);
    this.element.addEventListener('touchmove', this.onTouchMove);
    this.element.addEventListener('touchend', this.onTouchEnd);
  }

  onTouchStart(e) {
    this.touching = true;
    this.startX = e.touches[0].clientX;
    this.startY = e.touches[0].clientY;
  }

  onTouchMove(e) {
    if (!this.touching) return;

    this.currentX = e.touches[0].clientX;
    this.currentY = e.touches[0].clientY;

    const deltaX = this.currentX - this.startX;
    const deltaY = this.currentY - this.startY;

    // 判断手势方向和距离
    if (Math.abs(deltaX) > Math.abs(deltaY)) {
      // 水平手势
      if (deltaX > 0) {
        this.onSwipeRight(deltaX);
      } else {
        this.onSwipeLeft(-deltaX);
      }
    }
  }

  onTouchEnd() {
    this.touching = false;
  }
}

```

9. 使用 IntersectionObserver 优化滚动加载

在滚动加载场景中，使用 IntersectionObserver 可以提供更好的性能：

```
const observer = new IntersectionObserver((entries) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      loadMoreContent();
      observer.unobserve(entry.target);
    }
  });
}, {
  root: null,
  rootMargin: '100px',
  threshold: 0.1
});

observer.observe(document.querySelector('.load-more-trigger'));
```

优化是一个持续的过程，应该根据具体的应用场景和性能瓶颈来选择合适的优化策略，欢迎补充。



JavaScript

为 JavaScript 爱好人员提供：Web 前端相关技术教程、JavaScript、Node.js、Deno、V...
137篇原创内容

公众号