在我们日常的移动端项目开发中，处理滚动列表是再常见不过的需求了。 以滴滴为例，可以是这样竖向滚动的列表，如图所示：

也可以是横向滚动的导航栏，如图所示：

> 可以打开“微信 —> 钱包—>滴滴出行”体验效果。

我们在实现这类滚动功能的时候，会用到我写的第三方库，better-scroll。

什么是 better-scroll

better-scroll 是一个移动端滚动的解决方案，它是基于 iscroll 的重写，它和 iscroll 的主要区别在这里。better-scroll 也很强大，不仅可以做普通的滚动列表，还可以做轮播图、picker 等等。

better-scroll 的滚动原理

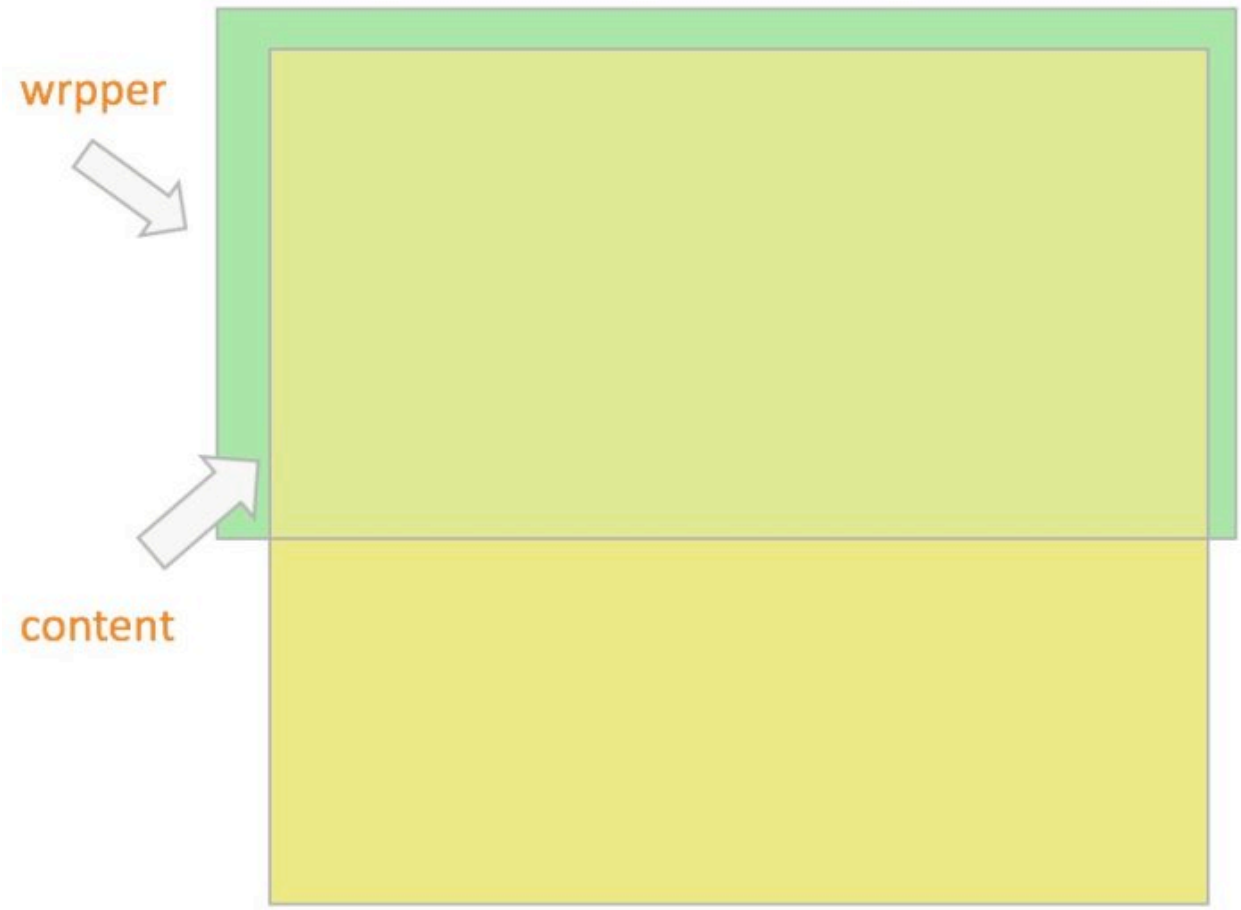
不少同学可能用过 better-scroll，我收到反馈最多的问题是：

> 我的 better-scroll 初始化了， 但是没法滚动。

不能滚动是现象，我们得搞清楚这其中的根本原因。在这之前，我们先来看一下浏览器的滚动原理： 浏览器的滚动条大家都会遇到，当页面内容的高度超过视口高度的时候，会出现纵向滚动条；当页面内容的宽度超过视口宽度的时候，会出现横向滚动条。也就是当我们的视口展示不下内容的时候，会通过滚动条的方式让用户滚动屏幕看到剩余的内容。
那么对于 better-scroll 也是一样的道理，我们先来看一下 better-scroll 常见的 html 结构：

```
<div class="wrapper">
  <ul class="content">
    <li>...</li>
    <li>...</li>
    ...
  </ul>
</div>
```

为了更加直观，我们再来看一张图：



绿色部分为 wrapper，也就是父容器，它会有固定的高度。黄色部分为 content，它是父容器的第一个子元素，它的高度会随着内容的大小而撑高。那么，当 content 的高度不超过父容器的高度，是不能滚动的，而它一旦超过了父容器的高度，我们就可以滚动内容区了，这就是 better-scroll 的滚动原理。
那么，我们怎么初始化 better-scroll 呢，如果是上述 html 结构，那么初始化代码如下：

```
import BScroll from 'better-scroll'
let wrapper = document.querySelector('.wrapper')
let scroll = new BScroll(wrapper, {})
```

better-scroll 对外暴露了一个 BScroll 的类，我们初始化只需要 new 一个类的实例即可。第一个参数就是我们 wrapper 的 DOM 对象，第二个是一些配置参数，具体参考 better-scroll 的文档。
better-scroll 的初始化时机很重要，因为它在初始化的时候，会计算父元素和子元素的高度和宽度，来决定是否可以纵向和横向滚动。因此，我们在初始化它的时候，必须确保父元素和子元素的内容已经正确渲染了。如果子元素或者父元素 DOM 结构发生改变的时候，必须重新调用 scroll.refresh() 方法重新计算来确保滚动效果的正常。所以同学们反馈的 better-scroll 不能滚动的原因多半是初始化 better-scroll 的时机不对，或者是当 DOM 结构发送变化的时候并没有重新计算 better-scroll。

better-scroll 遇见 Vue

相信很多同学对 Vue.js 都不陌生，当 better-scroll 遇见 Vue，会擦出怎样的火花呢?

如何在 Vue 中使用 better-scroll

很多同学开始接触使用 better-scroll 都是受到了我的一门教学课程——《Vue.js高仿饿了么外卖App》 的影响。在那门课程中，我们把 better-scroll 和 Vue 做了结合，实现了很多列表滚动的效果。在 Vue 中的使用方法如下：

```
<template>
  <div class="wrapper" ref="wrapper">
    <ul class="content">
      <li>...</li>
      <li>...</li>
      ...
    </ul>
  </div>
</template>
<script>
  import BScroll from 'better-scroll'
  export default {
    mounted() {
      this.$nextTick(() => {
        this.scroll = new Bscroll(this.$refs.wrapper, {})
      })
    }
  }
</script>
```

Vue.js 提供了我们一个获取 DOM 对象的接口—— vm.$refs$。在这里，我们通过了$this$.refs.wrapper访问到了这个 DOM 对象，并且我们在 mounted 这个钩子函数里，this.$nextTick 的回调函数中初始化 better-scroll 。因为这个时候，wrapper 的 DOM 已经渲染了，我们可以正确计算它以及它内层 content 的高度，以确保滚动正常。
这里的 this.
$nextTick$是一个异步函数，为了确保DOM已经渲染，感兴趣的同学可以了解一下它的[内部实现细节]($https://link.zhihu.com/?target=https$
nextTick 替换成 setTimeout(fn, 20) 也是可以的（20 ms 是一个经验值，每一个 Tick 约为 17 ms），对用户体验而言都是无感知的。

异步数据的处理

在我们的实际工作中，列表的数据往往都是异步获取的，因此我们初始化 better-scroll 的时机需要在数据获取后，代码如下：

```
<template>
  <div class="wrapper" ref="wrapper">
    <ul class="content">
      <li v-for="item in data">{{item}}</li>
    </ul>
  </div>
</template>
<script>
  import BScroll from 'better-scroll'
  export default {
    data() {
      return {
        data: []
      }
    },
    created() {
      requestData().then((res) => {
        this.data = res.data
        this.$nextTick(() => {
          this.scroll = new Bscroll(this.$refs.wrapper, {})
        })
      })
    }
  }
</script>
```

这里的 requestData 是伪代码，作用就是发起一个 http 请求从服务端获取数据，并且这个函数返回的是一个 promise（实际项目中我们可能会用 axios 或者 vue-resource）。我们获取到数据的后，需要通过异步的方式再去初始化 better-scroll，因为 Vue 是数据驱动的， Vue 数据发生变化（this.data = res.data）到页面重新渲染是一个异步的过程，我们的初始化时机是要在 DOM 重新渲染后，所以这里用到了 this.$nextTick，当然替换成 setTimeout(fn, 20) 也是可以的。
为什么这里在 created 这个钩子函数里请求数据而不是放到 mounted 的钩子函数里？因为 requestData 是发送一个网络请求，这是一个异步过程，当拿到响应数据的时候，Vue 的 DOM 早就已经渲染好了，但是数据改变 —> DOM 重新渲染仍然是一个异步过程，所以即使在我们拿到数据后，也要异步初始化 better-scroll。

数据的动态更新

我们在实际开发中，除了数据异步获取，还有一些场景可以动态更新列表中的数据，比如常见的下拉加载，上拉刷新等。比如我们用 better-scroll 配合 Vue 实现下拉加载功能，代码如下：

```
<template>
  <div class="wrapper" ref="wrapper">
    <ul class="content">
      <li v-for="item in data">{{item}}</li>
    </ul>
    <div class="loading-wrapper"></div>
  </div>
</template>
<script>
  import BScroll from 'better-scroll'
  export default {
    data() {
      return {
        data: []
      }
    },
    created() {
      this.loadData()
    },
    methods: {
      loadData() {
        requestData().then((res) => {
          this.data = res.data.concat(this.data)
          this.$nextTick(() => {
            if (!this.scroll) {
              this.scroll = new Bscroll(this.$refs.wrapper, {})
              this.scroll.on('touchend', (pos) => {
                // 下拉动作
                if (pos.y > 50) {
                  this.loadData()
                }
              })
            } else {
              this.scroll.refresh()
            }
          })
        })
      }
    }
  }
</script>
```

这段代码比之前稍微复杂一些, 当我们在滑动列表松开手指时候， better-scroll 会对外派发一个 touchend 事件，我们监听了这个事件，并且判断了 pos.y > 50（我们把这个行为定义成一次下拉的动作）。如果是下拉的话我们会重新请求数据，并且把新的数据和之前的 data 做一次 concat，也就更新了列表的数据，那么数据的改变就会映射到 DOM 的变化。需要注意的一点，这里我们对 this.scroll 做了判断，如果没有初始化过我们会通过 new BScroll 初始化，并且绑定一些事件，否则我们会调用 this.scroll.refresh 方法重新计算，来确保滚动效果的正常。
这里，我们就通过 better-scroll 配合 Vue，实现了列表的下拉刷新功能，上拉加载也是类似的套路，一切看上去都是 ok 的。但是，我们发现这里写了大量命令式的代码（这一点不是 Vue.js 推荐的），如果有很多类似滚动的组件，我们就需要写很多类似的命令式且重复性的代码，而且我们把数据请求和 better-scroll 也做了强耦合，这些对于一个追求编程逼格的人来说，就不 ok 了。

scroll 组件的抽象和封装

因此，我们有强烈的需求抽象出来一个 scroll 组件，类似小程序的 scroll-view 组件，方便开发者的使用。
首先，我们要考虑的是 scroll 组件本质上就是一个可以滚动的列表组件，至于列表的 DOM 结构，只需要满足 better-scroll 的 DOM 结构规范即可，具体用什么标签，有哪些辅助节点（比如下拉刷新上拉加载的 loading 层），这些都不是 scroll 组件需要关心的。因此， scroll 组件的 DOM 结构十分简单，如下所示：

```
<template>
  <div ref="wrapper">
    <slot></slot>
  </div>
</template>
```

这里我们用到了 Vue 的特殊元素——— slot 插槽，它可以满足我们灵活定制列表 DOM 结构的需求。接下来我们来看看 JS 部分：

```
<script type="text/ecmascript-6">
  import BScroll from 'better-scroll'

  export default {
    props: {
      /**
       * 1 滚动的时候会派发scroll事件，会截流。
       * 2 滚动的时候实时派发scroll事件，不会截流。
       * 3 除了实时派发scroll事件，在swipe的情况下仍然能实时派发scroll事件
       */
      probeType: {
        type: Number,
        default: 1
      },
      /**
       * 点击列表是否派发click事件
       */
      click: {
        type: Boolean,
        default: true
      },
      /**
       * 是否开启横向滚动
       */
      scrollX: {
        type: Boolean,
        default: false
      },
      /**
       * 是否派发滚动事件
       */
```

```
    listenScroll: {
      type: Boolean,
      default: false
    },
    /**
     * 列表的数据
     */
    data: {
      type: Array,
      default: null
    },
    /**
     * 是否派发滚动到底部的事件，用于上拉加载
     */
    pullup: {
      type: Boolean,
      default: false
    },
    /**
     * 是否派发顶部下拉的事件，用于下拉刷新
     */
    pulldown: {
      type: Boolean,
      default: false
    },
    /**
     * 是否派发列表滚动开始的事件
     */
    beforeScroll: {
      type: Boolean,
      default: false
    },
    /**
     * 当数据更新后，刷新scroll的延时。
     */
    refreshDelay: {
      type: Number,
      default: 20
    }
  },
  mounted() {
    // 保证在DOM渲染完毕后初始化better-scroll
    setTimeout(() => {
      this._initScroll()
    }, 20)
  },
  methods: {
    _initScroll() {
      if (!this.$refs.wrapper) {
        return
      }
      // better-scroll的初始化
      this.scroll = new BScroll(this.$refs.wrapper, {
        probeType: this.probeType,
        click: this.click,
        scrollX: this.scrollX
      })

      // 是否派发滚动事件
```

```
91        if (this.listenScroll) {
92          this.scroll.on('scroll', (pos) => {
93            this.$emit('scroll', pos)
94          })
95        }
96
97        // 是否派发滚动到底部事件，用于上拉加载
98        if (this.pullup) {
99          this.scroll.on('scrollEnd', () => {
100           // 滚动到底部
101           if (this.scroll.y <= (this.scroll.maxScrollY + 50)) {
102             this.$emit('scrollToEnd')
103           }
104         })
105       }
106
107       // 是否派发顶部下拉事件，用于下拉刷新
108       if (this.pulldown) {
109         this.scroll.on('touchend', (pos) => {
110           // 下拉动作
111           if (pos.y > 50) {
112             this.$emit('pulldown')
113           }
114         })
115       }
116
117       // 是否派发列表滚动开始的事件
118       if (this.beforeScroll) {
119         this.scroll.on('beforeScrollStart', () => {
120           this.$emit('beforeScroll')
121         })
122       }
123     },
124     disable() {
125       // 代理better-scroll的disable方法
126       this.scroll && this.scroll.disable()
127     },
128     enable() {
129       // 代理better-scroll的enable方法
130       this.scroll && this.scroll.enable()
131     },
132     refresh() {
133       // 代理better-scroll的refresh方法
134       this.scroll && this.scroll.refresh()
135     },
136     scrollTo() {
137       // 代理better-scroll的scrollTo方法
138       this.scroll && this.scroll.scrollTo.apply(this.scroll, arguments)
139     },
140     scrollToElement() {
141       // 代理better-scroll的scrollToElement方法
142       this.scroll && this.scroll.scrollToElement.apply(this.scroll, arguments)
143     }
144   },
145   watch: {
146     // 监听数据的变化，延时refreshDelay时间后调用refresh方法重新计算，保证滚动效果正常
147     data() {
148       setTimeout(() => {
149         this.refresh()
```

```
        }, this.refreshDelay)
      }
    }
  }
</script>
```

JS 部分实际上就是对 better-scroll 做一层 Vue 的封装，通过 props 的形式，把一些对 better-scroll 定制化的控制权交给父组件；通过 methods 暴露的一些方法对 better-scroll 的方法做一层代理；通过 watch 传入的 data，当 data 发生改变的时候，在适当的时机调用 refresh 方法重新计算 better-scroll 确保滚动效果正常，这里之所以要有一个 refreshDelay 的设置是考虑到如果我们对列表操作用到了 transition-group 做动画效果，那么 DOM 的渲染完毕时间就是在动画完成之后。
有了这一层 scroll 组件的封装，我们来修改刚刚最复杂的代码（假设我们已经全局注册了 scroll 组件）。

```
<template>
  <scroll class="wrapper"
          :data="data"
          :pulldown="pulldown"
          @pulldown="loadData">
    <ul class="content">
      <li v-for="item in data">{{item}}</li>
    </ul>
    <div class="loading-wrapper"></div>
  </scroll>
</template>
<script>
  import BScroll from 'better-scroll'
  export default {
    data() {
      return {
        data: [],
        pulldown: true
      }
    },
    created() {
      this.loadData()
    },
    methods: {
      loadData() {
        requestData().then((res) => {
          this.data = res.data.concat(this.data)
        })
      }
    }
  }
</script>
```

可以很明显的看到我们的 JS 部分精简了非常多的代码，没有对 better-scroll 再做命令式的操作了，同时把数据请求和 better-scroll 也做了剥离，父组件只需要把数据 data 通过 prop 传给 scroll 组件，就可以保证 scroll 组件的滚动效果。同时，如果想实现下拉刷新的功能，只需要通过 prop 把 pulldown 设置为 true，并且监听 pulldown 的事件去做一些数据获取并更新的动作即可，整个逻辑也是非常清晰的。

插件 Vue 化引发的一些思考

这篇文章我不仅仅是要教会大家封装一个 scroll 组件，还想传递一些把第三方插件（原生 JS 实现）Vue 化的思考过程。很多学习 Vue.js 的同学可能还停留在 “XX 效果如何用 Vue.js 实现” 的程度，其实把插件 Vue 化有两点很关键，一个是对插件本身的实现原理很了解，另一个是对 Vue.js 的特性很了解。对插件本身的实现原理了解需要的是一个思考和钻研的过程，这个过程可能困难，但是收获也是巨大的；而对 Vue.js 的特性的了解，是需要大家对 Vue.js 多多使用，学会从平时的项目中积累和总结，也要善于查阅 Vue.js 的官方文档，关注一些 Vue.js 的升级等。

所以，我们拒绝伸手党，但也不是鼓励大家什么时候都要去造轮子，当我们在使用一些现成插件的同时，也希望大家能多多思考，去探索一下现象背后的本质，把 “XX 效果如何用 Vue.js 实现” 这句话从问号变成句号。