2024前端高频—Node&Webpack&性能优化篇

1\. Node是什么？

Node.js是一个基于 Chrome V8 引擎的**JavaScript运行环境**，这个环境就好比是服务器上的浏览器（虽然不是很恰当），但正是因为有了它才使得 js 变成了一门后台语言。

2\. Node解决了哪些问题?

Node在处理高并发,I/O密集场景有明显的性能优势

- 高并发,是指在同一时间并发访问服务器
- I/O密集指的是文件操作、网络操作、数据库,相对的有CPU密集,CPU密集指的是逻辑处理运算、压缩、解压、加密、解密

> Web主要场景就是接收客户端的请求读取静态资源和渲染界面,所以Node非常适合Web应用的开发。

3\. Node 的应用场景

一般来说，node 主要应用于以下几个方面：

- 自动化构建等工具
- 中间层
- 小项目

第一点对于前端同学来说应该是重中之重了，什么工程化、自动构建工具就是用 node 写出来的，它是前端的一大分水岭之一

4\. 请介绍一下Node事件循环的流程

- 在进程启动时，Node便会创建一个类似于while(true)的循环，每执行一次循环体的过程我们成为Tick。

- 每个Tick的过程就是查看是否有事件待处理。如果有就取出事件及其相关的回调函数。然后进入下一个循环，如果不再有事件处理，就退出进程。

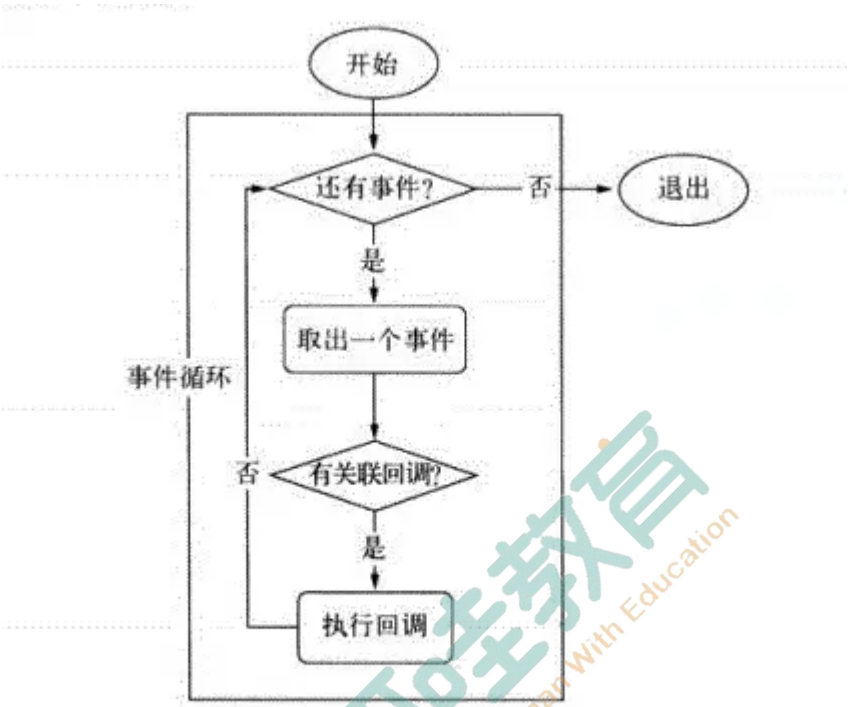


图3-11 Tick流程图

4.2 浏览器和 Node.js 中的事件循环机制有什么区别？

Node.js中宏任务分成了几种类型，并且放在了不同的task queue(事件队列)里。不同的task queue在执行顺序上也有区别，微任务放在了每个task queue的末尾：

- `setTimeout/setInterval` 属于 timers 类型；
- `setImmediate` 属于 check 类型；
- socket 的 close 事件属于 close callbacks 类型；
- 其他 MacroTask 都属于 poll 类型。
- `process.nextTick` 本质上属于 MicroTask，但是它先于所有其他 MicroTask 执行；
- 所有 MicroTask 的执行时机在不同类型的 MacroTask 切换后。
- idle/prepare 仅供内部调用，我们可以忽略。
- pending callbacks 不太常见，我们也可以忽略。

5\. 在每个tick的过程中，如何判断是否有事件需要处理呢？

- 每个事件循环中有一个或者多个观察者，而判断是否有事件需要处理的过程就是向这些观察者询问是否有要处理的事件。
- 在Node中，事件主要来源于网络请求、文件的I/O等，这些事件对应的观察者有文件I/O观察者，网络I/O的观察者。
- 事件循环是一个典型的生产者/消费者模型。异步I/O，网络请求等则是事件的生产者，源源不断为Node提供不同类型的事件，这些事件被传递到对应的观察者那里，事件循环则从观察者那里取出事件并处理。
- 在windows下，这个循环基于IOCP创建，在*nix下则基于多线程创建

6\. 请描述一下整个异步I/O的流程

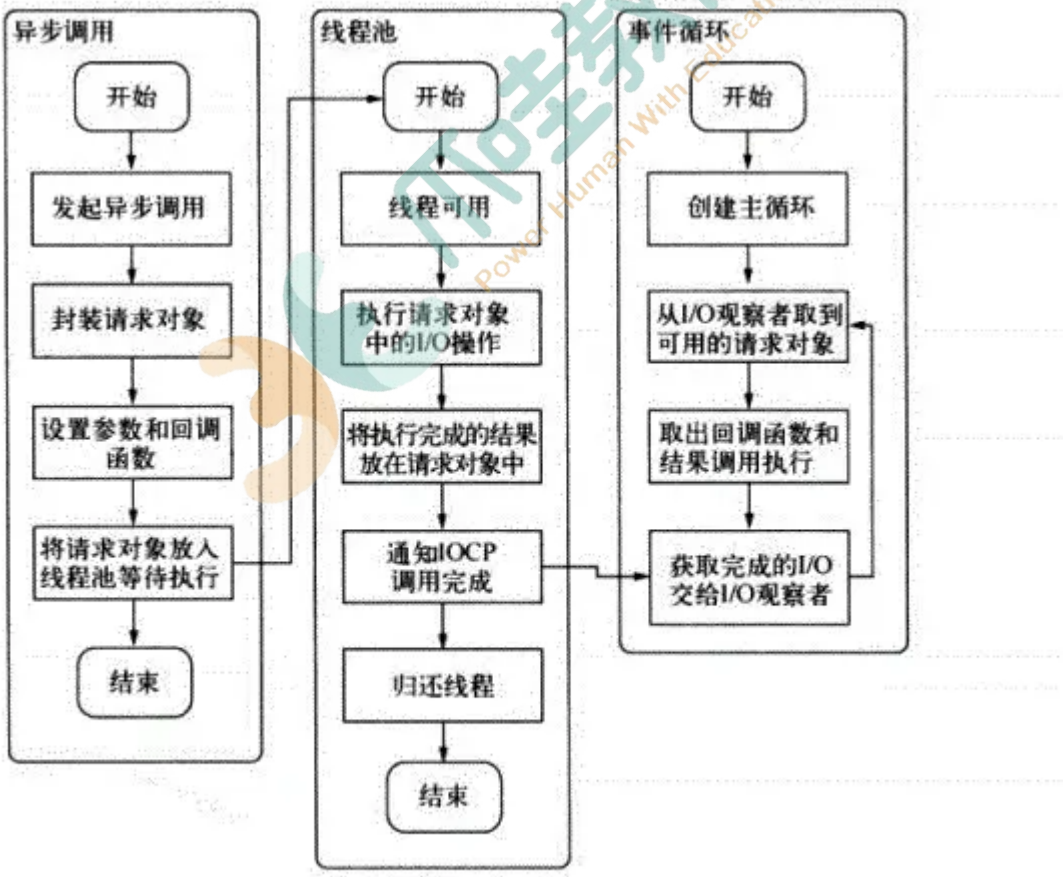


图3-13　整个异步I/O的流程

7\. Express

7.1 Express框架介绍

1.Express是NodeJS开发中一个非常重量级的第三方框架，它对于NodeJS服务端就相当于Jquery对于HTML客户端。

- **如果连Express都不会用，基本上都不好意思跟别人说你会NodeJS**

2.Express官网是这样介绍自己的:基于 Node.js 平台，快速、开放、极简的 web 开发框架。

- *Express一个非常重要的亮点就是它没有改变nodejs已有的特性，而是在它的基础上进行了拓展*
    - **也就是说，使用Express你既可以使用nodejs原生的任何API，也能使用Express的API**

7.2 Express三大核心功能

- 1.托管静态资源
    - 第二天讲的nodejs实现静态服务器功能在express中只需要一行代码
- 2.路由
    - express自带路由功能,让Node服务端开发变得极其简单
    - express支持链式语法，可以让代码看起来更加简洁
- ==3.中间件==
    - Express最为核心的技术和思想，万物皆中间件
        - 中间件虽然理解起来有点困难，但是使用起来非常方便，类似于`bootstrap插件`.

8\. Node模块化规范

8.1 什么是模块化? 模块化有什么好处?

- 1.什么叫模块化？（模块 == js文件）
    - `一个js文件`可以`引入另一个js文件中的数据`, 这种开发方式就叫做模块化开发

- 2.模块化开发好处？
  - （1）将功能分离出来
    - 一个js文件只负责处理一个功能，这样做的好处是业务逻辑清晰，便于维护
  - （2）按需导入
    - 用到哪一个功能，就导入哪一个js文件。模块化开发是`渐进式框架`的共同特征。
  - （3）避免变量污染
    - 一个js文件(模块),都是一个独立的作用域。 互不干扰，不用考虑变量名冲突问题。

8.2 模块化语法规范介绍

任何一个语法，都需要遵循一定的规范。不同的平台需要遵循的规范不同。一般情况下，模块化语法规范主要有三种。

- CommonJS 规范：nodejs 默认支持的
  - 文档：javascript.ruanyifeng.com/nodejs/modu...
- ES6规范：前端的ES6语法支持的规范
  - 文档：es6.ruanyifeng.com/#docs/modul...
- CMD 和 AMD 模块化规范 ： ES6语法之前的模块化语法规范，现在已经几乎不用了

8.3 CommonJS规范实现模块化语法

- CommonJS规范只有两句话
  - 1.模块必须要使用 `require()` 导入
  - 2.模块必须要使用 `module.exports` 导出
    - 这两个语法都是nodejs环境自带的全局语法
- exports与module.exports区别
  - (1)如果分不清两者区别,就使用 `module.exports`
  - (2)exports不能去修改堆地址，只能往堆地址里面加数据
    - 错误写法： exports = { name:'张三' }

        - 这样写是修改堆地址
    - 正确写法: exports.name = '张三'
        - 这样写没有修改堆地址，而是往堆地址中加数据

8.4 模块缓存机制

- 1.当一个模块第一次会加载时，nodejs会执行里面的js代码，并且导出模块
- 2.nodejs会将导出的模块放入缓存中
- 3.当重复导入一个模块的时候，nodejs会先从缓存中读取模块。 如果缓存中有，就从缓存读取。缓存没有重复步骤1

8.5 nodejs三种模块及require()加载原理

- 1.nodejs有三种模块
    - 第一种： 核心模块、内置模块
        - nodejs作者写的，这些模块js文件会随着安装nodejs的时候一起安装。因此我们可以直接使用，而无需下载。
            - 例如： fs 、path、http都是核心模块
    - 第二种：第三方模块
        - npm官网上面的模块，这些都是大佬写的模块。需要下载后使用
            - 例如： express、cors、bod-parser
    - 第三种：自定义模块
        - 我们自己写的js文件
- 2.require('文件路径')加载原理
    - (1)自定义模块：必须要写文件路径， `require()` 会得到这个模块里面的 `module.exports` 对象
    - (2)第三方模块 ：写模块名。 nodejs会自动从你的node_module文件夹里面去找这个模块的名字，然后执行模块里面的 `index.js` 代码，得到里面的 `module.exports`
        - 如果当前目录没有module.exports就会从上级目录找，以此类推。一直找到你的磁盘根目录。 还找不到就会报错提示 模块不存在（有点类似于变量作用域就近原则）

- (3)核心模块：写模块名。 nodejs会自动从你的node安装包路径里面去找。

8.6 require和import的区别

1. 导入`require` 导出 `exports/module.exports` 是 `CommonJS` 的标准，通常适用范围如 `Node.js`
2. import/export `是`ES6` 的标准，通常适用范围如 React
3. `require` 是**赋值过程**并且是**运行时才执行**，也就是**_同步加载_**
4. `require` 可以理解为一个全局方法，因为它是一个方法所以意味着可以在任何地方执行。
5. `import` 是**解构过程**并且是**编译时执行**，理解为**_异步加载_**
6. `import` 会提升到整个模块的头部，具有置顶性，但是建议写在文件的顶部。

9\. 说说前端渲染和后端渲染，以及他们的优缺点

前端渲染

指的是后端返回JSON数据，前端利用预先写的html模板，循环读取JSON数据，拼接字符串（ES6的模板字符串特性大大减少了拼接字符串的的成本），并插入页面。

后端渲染

前端请求，后端用后台模板引擎直接生成html，前端接收到数据之后，直接插入页面。

区别

	前端渲染	后端渲染
页面呈现速度	主要受限于带宽和客户端机器的好坏，优化的好，可以逐步动态展开内容，感觉上会更快一点	快，受限于用户的带宽
流量消耗	多一点点（一个前端框架大概50KB）	少一点点（可以省去前端框架部分的代码）
可维护性	好，前后端分离，各施其职，代码一目明了	差（前后端东西放一起，不利于维护）
SEO友好度	差，大量使用Ajax，多数浏览器不能抓取Ajax数据	好
编码效率	高，前后端各自只做自己擅长的东西，后端最后只输出接口，不用管页面呈现，只要前后端人员能力不错，效率不会低	低（这个跟不同的团队不同，可能不对）

2\. 什么是CND

2.1 CDN的概念

CDN（Content Delivery Network，**内容分发网络）是指一种通过互联网互相连接的电脑网络系统**，利用最靠近每位用户的服务器，更快、更可靠地将音乐、图片、视频、应用程序及其他文件发送给用户，来提供高性能、可扩展性及低成本的网络内容传递给用户。

2.2 CDN的作用

CDN一般会用来托管Web资源（包括文本、图片和脚本等），可供下载的资源（媒体文件、软件、文档等），应用程序（门户网站等）。**使用CDN来加速这些资源的访问。**

2.3 CDN的使用场景

使用第三方的CDN服务： 如果想要开源一些项目，可以使用第三方的CDN服务

使用CDN进行静态资源的缓存： 将自己网站的静态资源放在CDN上，比如js、css、图片等。可以将整个项目放在CDN上，完成一键部署。

直播传送： 直播本质上是使用流媒体进行传送，CDN也是支持流媒体传送的，所以直播完全可以使用CDN来提高访问速度。CDN在处理流媒体的时候与处理普通静态文件有所不同，普通文件如果在边缘节点没有找到的话，就会去上一层接着寻找，但是流媒体本身数据量就非常大，如果使用回源的方式，必然会带来性能问题，所以流媒体一般采用的都是主动推送的方式来进行。

3.什么是 懒加载(图片)

3.1 懒加载的概念

懒加载也叫做延迟加载、按需加载，指的是在长网页中延迟加载图片数据，是一种较好的网页性能优化的方式。在比较长的网页或应用中，如果图片很多，所有的图片都被加载出来，而用户只能看到可视窗口的那一部分图片数据，这样就浪费了性能。

如果使用图片的懒加载就可以解决以上问题。在滚动屏幕之前，可视化区域之外的图片不会进行加载，在滚动屏幕时才加载。这样使得网页的加载速度更快，减少了服务器的负载。懒加载适用于图片较多，页面列表较长（长列表）的场景中。

3.2 懒加载的特点

减少无用资源的加载：使用懒加载明显减少了服务器的压力和流量，同时也减小了浏览器的负担。

提升用户体验: 如果同时加载较多图片，可能需要等待的时间较长，这样影响了用户体验，而使用懒加载就能大大的提高用户体验。

防止加载过多图片而影响其他资源文件的加载 ：会影响网站应用的正常使用。

3.3 懒加载的实现原理

图片的加载是由`src`引起的，当对`src`赋值时，浏览器就会请求图片资源。根据这个原理，我们使用HTML5 的`data-xxx`属性来储存图片的路径，在需要加载图片的时候，将`data-xxx`中图片的路径赋值给`src`，这样就实现了图片的按需加载，即懒加载。

注意：`data-xxx` 中的`xxx`可以自定义，这里我们使用`data-src`来定义。

懒加载的实现重点在于确定用户需要加载哪张图片，在浏览器中，可视区域内的资源就是用户需要的资源。所以当图片出现在可视区域时，获取图片的真实地址并赋值给图片即可。

3.32 Vue3实现图片懒加载

导入vueuse插件，使用vueuse封装的useIntersectionObserver监听对应的DOM元素，通过里面的isIntersecting属性的布尔值判断来设置img的src

可以封装一个v-lazy的自定义指令来控制img的src

```
app.directive('lazy', {
      mounted: (el: HTMLImageElement, { value }) => {

        const { stop } = useIntersectionObserver(el, ([{ isIn
tersecting }]) => {
          if (isIntersecting) {

            stop()

            el.src = value
            el.onerror = () => {

              el.src = defaultImg
            }
```

```
          }
        })
      }
    })
```

3.33 列表数据懒加载 (利用hooks抽取)

在hooks里封装通用的数据懒加载api

```
export function useLazyData(callBack: () => void) {

  const target = ref(null)
  const { stop } = useIntersectionObserver(

    target,


    ([{ isIntersecting }]) => {

      if (isIntersecting) {
        stop()
        callBack()
      }
    }
  )
  return target
}
```

在组件中使用

```
import useStore from '@/store'
import { useLazyData } from '@/utils/hooks';
const { home } = useStore()
```

```
const target = useLazyData(() => home.getHotList())
```

3.4 懒加载与预加载的区别

这两种方式都是提高网页性能的方式，两者主要区别是一个是提前加载，一个是迟缓甚至不加载。懒加载对服务器前端有一定的缓解压力作用，预加载则会增加服务器前端压力。

- **懒加载也叫延迟加载，指的是在长网页中延迟加载图片的时机，当用户需要访问时，再去加载**，这样可以提高网站的首屏加载速度，提升用户的体验，并且可以减少服务器的压力。它适用于图片很多，页面很长的电商网站的场景。
- **预加载指的是将所需的资源提前请求加载到本地，这样后面在需要用到时就直接从缓存取资源。** 通过预加载能够减少用户的等待时间，提高用户的体验。我了解的预加载的最常用的方式是使用 js 中的 image 对象，通过为 image 对象来设置 scr 属性，来实现图片的预加载。

4\. 什么是回流(重排)与重绘

4.1 什么是回流(重排)，哪些操作会导致回流

当渲染树中部分或者全部元素的尺寸、结构或者属性发生变化时，浏览器会重新渲染部分或者全部文档的过程就称为**回流**。

下面这些操作会导致回流：

- 页面的首次渲染
- 浏览器的窗口大小发生变化
- 元素的内容发生变化
- 元素的尺寸或者位置发生变化
- 元素的字体大小发生变化
- 激活CSS伪类
- 查询某些属性或者调用某些方法
- 添加或者删除可见的DOM元素

在触发回流（重排）的时候，由于浏览器渲染页面是基于流式布局的，所以当触发回流时，会导致周围的DOM元素重新排列，它的影响范围有两种：

- 全局范围：从根节点开始，对整个渲染树进行重新布局
- 局部范围：对渲染树的某部分或者一个渲染对象进行重新布局

4.2 什么是重绘，哪些操作会导致重绘

当页面中某些元素的样式发生变化，但是不会影响其在文档流中的位置时，浏览器就会对元素进行重新绘制，这个过程就是**重绘**。

下面这些操作会导致重绘：

- color、background 相关属性：background-color、background-image 等
- outline 相关属性：outline-color、outline-width 、text-decoration
- border-radius、visibility、box-shadow

注意： **当触发回流时，一定会触发重绘，但是重绘不一定会引发回流。**

4.3 如何避免回流与重绘？

减少回流与重绘的措施：

- 操作DOM时，尽量在低层级的DOM节点进行操作
- 不要使用`table`布局， 一个小的改动可能会使整个`table`进行重新布局
- 使用CSS的表达式
- 不要频繁操作元素的样式，对于静态页面，可以修改类名，而不是样式。
- 使用absolute或者fixed，使元素脱离文档流，这样他们发生变化就不会影响其他元素
- 避免频繁操作DOM，可以创建一个文档片段`documentFragment`，在它上面应用所有DOM操作，最后再把它添加到文档中
- 将元素先设置`display: none`，操作结束后再把它显示出来。因为在display属性为none的元素上进行的DOM操作不会引发回流和重绘。
- 将DOM的多个读操作（或者写操作）放在一起，而不是读写操作穿插着写。这得益于**浏览器的渲染队列机制**。

4.4 浏览器的渲染队列

浏览器针对页面的回流与重绘，进行了自身的优化——**渲染队列**

浏览器会将所有的回流、重绘的操作放在一个队列中，当队列中的操作到了一定的数量或者到了一定的时间间隔，浏览器就会对队列进行批处理。这样就会让多次的回流、重绘变成一次回流重绘。

4.5 如何优化动画？

对于如何优化动画，我们知道，一般情况下，动画需要频繁的操作DOM，就就会导致页面的性能问题，我们可以将动画的`position`属性设置为`absolute`或者`fixed`，将动画脱离文档流，这样他的回流就不会影响到页面了。

4.6 documentFragment (文档碎片)是什么？用它跟直接操作 DOM 的区别是什么？

当我们把一个 DocumentFragment 节点插入文档树时，插入的不是 DocumentFragment 自身，而是它的所有子孙节点。在频繁的DOM操作时，我们就可以将DOM元素插入DocumentFragment，之后一次性的将所有的子孙节点插入文档中。DocumentFragment不是真实 DOM 树的一部分，它的变化**不会触发 DOM 树的重新渲染**，这样就**大大提高了页面的性能**。

假如有 10000 个元素需要添加到页面上，你觉得怎么操作性能最好（考察`文档碎片`）

```
<script>
     /* console.time('耗时')
    for (let i = 1
      document.body.innerHTML = document.body.innerHTML + `<d
iv>${i}</div>`
    }
    console.timeEnd('耗时') // 1586.053955078125 ms */

    /* console.time('耗时')
    let str = ''
    for (let i = 1
      str += `<div>${i}</div>`
    }
    document.body.innerHTML = str
    console.timeEnd('耗时') // 2.5810546875 ms */
```

```
    /* console.time('耗时')
    const arr = []
    for (let i = 1
      arr.push(`<div>${i}</div>`)
    }
    document.body.innerHTML = arr.join('')
    console.timeEnd('耗时') // 2.883056640625 ms */

    /* console.time('耗时')
    for (let i = 1
      const oDiv = document.createElement('div')
      // 更灵活
      oDiv.innerHTML = i
      oDiv.onclick = function () { }
      oDiv.style.backgroundColor = 'red'
      document.body.appendChild(oDiv)
    }
    console.timeEnd('耗时') // 7.409912109375 ms */

    console.time('耗时')
    // 篮子，“文档碎片”
    const oFrag = document.createDocumentFragment()
    for (let i = 1
      const oDiv = document.createElement('div')
      oDiv.innerHTML = i
      oFrag.appendChild(oDiv)
    }
    document.body.appendChild(oFrag)
    console.timeEnd('耗时') // 13.442138671875 ms
</script>
```

5\. 什么是节流与防抖

5.1 对节流与防抖的理解

- 函数防抖是指**事件被触发 n 秒后再执行回调，如果在这 n 秒内事件又被触发，则重新计时。** 这可以使用在一些点击请求的事件上，避免因为用户的多次点击向后端发送多次请求。
- 函数节流是指规定一个单位时间，**在这个单位时间内，只能有一次触发事件的回调函数执行**，如果在同一个单位时间内某事件被触发多次，只有一次能生效。节流可以使用在 scroll 函数的事件监听上，通过事件节流来降低事件调用的频率。

5.2 适用场景

防抖函数的应用场景：

- 按钮提交场景：防止多次提交按钮，**只执行最后提交的一次**
- 服务端验证场景：表单验证需要服务端配合，只执行一段连续的输入事件的最后一次，还有搜索联想词功能类似生存环境请用lodash.debounce

节流函数的适用场景：

- 拖拽场景：**固定时间内只执行一次**，防止超高频次触发位置变动
- 缩放场景：监控浏览器resize
- 动画场景：避免短时间内多次触发动画引起性能问题

5.3 代码实现

```
function debounce(fn, date) {
  let timer
  return function (...arg) {
    timer && clearTimeout(timer)
    timer = setTimeout(() => {

      fn.apply(this, arg)
    }, date)
  }
}
```

```
function debounce(fn, data) {
  let timer = +new Date()
  return function (...arg) {
    let newTimer = +new Date()
    if (newTimer - timer >= data) {
      fn.apply(this, arg)
      timer = +new Date()
    }
  }
}

box.addEventListener('click', debounce(function (e) {
  if (e.target.tagName === 'BUTTON') {
    console.log(111);
  }
}, 2000))
```

6\. 如何对项目中的图片进行优化？

1. 不用图片。很多时候会使用到很多修饰类图片，其实这类修饰图片完全可以用 CSS 去代替。
2. 对于移动端来说，屏幕宽度就那么点，完全没有必要去加载原图浪费带宽。一般图片都用 CDN 加载，可以计算出适配屏幕的宽度，然后去请求相应裁剪好的图片。
3. 小图使用 base64 格式
4. 将多个图标文件整合到一张图片中（雪碧图）
5. 选择正确的图片格式：
    - 对于能够显示 WebP 格式的浏览器尽量使用 WebP 格式。因为 WebP 格式具有更好的图像数据压缩算法，能带来更小的图片体积，而且拥有肉眼识别无差异的图像质量，缺点就是兼容性并不好
    - 小图使用 PNG，其实对于大部分图标这类图片，完全可以使用 SVG 代替

- 照片使用 JPEG

7\. webpack知识总结

7.1.什么是webpack

- *静态模块打包工具*

7.2.webpack作用

- 分析、压缩、打包代码

7.3.webpack好处

- 减少文件体积、减少文件数量
- 提高网页加载速度

7.4.webpack工作流程

- 1.初始化参数：从配置文件读取与合并参数，得出最终的参数
- 2.开始编译：用上一步得到的参数初始化 Compiler 对象，加载所有配置的插件，开始执行编译
- 3.确定入口：根据配置中的 entry 找出所有的入口文件
- 4.编译模块：从入口文件出发，调用所有配置的 Loader 对模块进行翻译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理
- 5.完成模块编译：在经过第4步使用 Loader 翻译完所有模块后，得到了每个模块被翻译后的最终内容以及它们之间的依赖关系
- 6.输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的 Chunk，再把每个 Chunk 转换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会
- 7。输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名，把文件内容写入到文件系统。

7.5.什么是loader，什么是Plugin

- `loader`直译为"加载器"。webpack将一切文件视为模块，但是webpack原生是只能解析js文件，如果想将其他文件也打包的话，就会用到loader。 所以loader的作用是让webpack拥有了加载和解析非JavaScript文件的能力。
    - 说人话： loader就是用于解析文件的 （类似War3的游戏地图）
        - 例如：css-loader 、style-loader、image-loader
- `Plugin`直译为"插件"。Plugin可以扩展webpack的功能，让webpack具有更多的灵活性。 在 webpack 运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在合适的时机通过 webpack 提供的 API 改变输出结果。
    - 说人话：插件就是拓展功能的 （类似游戏的作弊器）
        - 例如：html-webpack-plugin,
- 个人见解： 广义来说,loader属于插件的一种。
    - 插件范围很广 ： 只要不是webapck原生的功能，都可以理解为插件
    - loader ： 一种特殊的插件，主要是用在webpack编译环节，帮我们编译各种文件的

7.6 有哪些常见的Loader？你用过哪些Loader？

`raw-loader`：加载文件原始内容（utf-8）

`file-loader`：把文件输出到一个文件夹中，在代码中通过相对 URL 去引用输出的文件 (处理图片和字体)

`url-loader`：与 file-loader 类似，区别是用户可以设置一个阈值，大于阈值会交给 file-loader 处理，小于阈值时返回文件 base64 形式编码 (处理图片和字体)

`source-map-loader`：加载额外的 Source Map 文件，以方便断点调试

`svg-inline-loader`：将压缩后的 SVG 内容注入代码中

`image-loader`：加载并且压缩图片文件

`json-loader` 加载 JSON 文件（默认包含）

`handlebars-loader`: 将 Handlebars 模版编译成函数并返回

`babel-loader`：把 ES6 转换成 ES5

`ts-loader`: 将 TypeScript 转换成 JavaScript

`awesome-typescript-loader`：将 TypeScript 转换成 JavaScript，性能优于 ts-loader

`sass-loader`：将SCSS/SASS代码转换成CSS

`css-loader`：加载 CSS，支持模块化、压缩、文件导入等特性

`style-loader`：把 CSS 代码注入到 JavaScript 中，通过 DOM 操作去加载 CSS

`postcss-loader`：扩展 CSS 语法，使用下一代 CSS，可以配合 autoprefixer 插件自动补齐 CSS3 前缀

`eslint-loader`：通过 ESLint 检查 JavaScript 代码

`tslint-loader`：通过 TSLint检查 TypeScript 代码

7.7 有哪些常见的Plugin？你用过哪些Plugin？

`define-plugin`：定义环境变量 (Webpack4 之后指定 mode 会自动配置)

`ignore-plugin`：忽略部分文件

`html-webpack-plugin`：简化 HTML 文件创建 (依赖于 html-loader)

`web-webpack-plugin`：可方便地为单页应用输出 HTML，比 html-webpack-plugin 好用

`uglifyjs-webpack-plugin`：不支持 ES6 压缩 (Webpack4 以前)

`terser-webpack-plugin`: 支持压缩 ES6 (Webpack4)

`webpack-parallel-uglify-plugin`: 多进程执行代码压缩，提升构建速度

`mini-css-extract-plugin`: 分离样式文件，CSS 提取为独立文件，支持按需加载 (替代 extract-text-webpack-plugin)

`serviceworker-webpack-plugin`：为网页应用增加离线缓存功能

`clean-webpack-plugin`: 目录清理

`ModuleConcatenationPlugin`: 开启 Scope Hoisting

`speed-measure-webpack-plugin`: 可以看到每个 Loader 和 Plugin 执行耗时 (整个打包耗时、每个 Plugin 和 Loader 耗时)

`webpack-bundle-analyzer`: 可视化 Webpack 输出文件的体积 (业务组件、依赖第三方模块)

7.8 那你再说一说Loader和Plugin的区别？

`Loader` 本质就是一个函数，在该函数中对接收到的内容进行转换，返回转换后的结果。因为 Webpack 只认识 JavaScript，所以 Loader 就成了翻译官，对其他类型的资源进行转译的预处理工作。

`Plugin` 就是插件，基于事件流框架 `Tapable`，插件可以扩展 Webpack 的功能，在 Webpack 运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在合适的时机通过 Webpack 提供的 API 改变输出结果。

`Loader` 在 module.rules 中配置，作为模块的解析规则，类型为数组。每一项都是一个 Object，内部包含了 test(类型文件)、loader、options (参数)等属性。

`Plugin` 在 plugins 中单独配置，类型为数组，每一项是一个 Plugin 的实例，参数都通过构造函数传入

7.9 说一下 Webpack 的热更新原理吧

`Webpack` 的热更新又称热替换（`Hot Module Replacement`），缩写为 `HMR`。 这个机制可以做到不用刷新浏览器而将新变更的模块替换掉旧的模块。

HMR的核心就是客户端从服务端拉去更新后的文件，准确的说是 chunk diff (chunk 需要更新的部分)，实际上 WDS 与浏览器之间维护了一个 `Websocket`，当本地资源发生变化时，WDS 会向浏览器推送更新，并带上构建时的 hash，让客户端与上一次资源进行对比。客户端对比出差异后会向 WDS 发起 `Ajax` 请求来获取更改内容(文件列表、hash)，这样客户端就可以再借助这些信息继续向 WDS 发起 `jsonp` 请求获取该chunk的增量更新。

后续的部分(拿到增量更新之后如何处理？哪些状态该保留？哪些又需要更新？)由 `HotModulePlugin` 来完成，提供了相关 API 以供开发者针对自身场景进行处理，像 `react-hot-loader` 和 `vue-loader` 都是借助这些 API 实现 HMR。

7.10 代码分割的本质是什么？有什么意义呢？

代码分割的本质其实就是在 `源代码直接上线` 和 `打包成唯一脚本main.bundle.js` 这两种极端方案之间的一种更适合实际场景的中间状态。

「用可接受的服务器性能压力增加来换取更好的用户体验。」

源代码直接上线：虽然过程可控，但是http请求多，性能开销大。

打包成唯一脚本：一把梭完自己爽，服务器压力小，但是页面空白期长，用户体验不好。

8\. Webpack优化

8.1 如何提高webpack的打包速度?

（1）优化 Loader

对于 Loader 来说，影响打包效率首当其冲必属 Babel 了。因为 Babel 会将代码转为字符串生成 AST，然后对 AST 继续进行转变最后再生成新的代码，项目越大，**转换代码越多，效率就越低**。

（2）HappyPack

受限于 Node 是单线程运行的，所以 Webpack 在打包的过程中也是单线程的，特别是在执行 Loader 的时候，长时间编译的任务很多，这样就会导致等待的情况。

HappyPack 可以将 Loader 的同步执行转换为并行的，这样就能充分利用系统资源来加快打包效率了

（3）DllPlugin

DllPlugin 可以将特定的类库提前打包然后引入。这种方式可以极大的减少打包类库的次数，只有当类库更新版本才有需要重新打包，并且也实现了将公共代码抽离成单独文件的优化方案。

（4）代码压缩

在 Webpack3 中，一般使用 `UglifyJS` 来压缩代码，但是这个是单线程运行的，为了加快效率，可以使用 `webpack-parallel-uglify-plugin` 来并行运行 `UglifyJS`，从而提高效率。

在 Webpack4 中，不需要以上这些操作了，只需要**将** `mode` **设置为** `production` 就可以默认开启以上功能。代码压缩也是我们必做的性能优化方案，当然我们不止可以压缩 JS 代码，还可以压缩 HTML、CSS 代码，并且在压缩 JS 代码的过程中，我们还可以通过配置实现比如删除 `console.log` 这类代码的功能。

8.2 如何减少 Webpack 打包体积

（1）按需加载

在开发 SPA 项目的时候，项目中都会存在很多路由页面。如果将这些页面全部打包进一个 JS 文件的话，虽然将多个请求合并了，但是同样也加载了很多并不需要的代码，耗费了更长的时间。那么为了首页能更快地呈现给用户，希望首页能加载的文件体积越小越好，**这时候就可以使用按需加载，将每个路由页面单独打包为一个文件**。当然不仅仅路由可以按需加载，对于 `loadash` 这种大型类库同样可以使用这个功能。

（2）Scope Hoisting

Scope Hoisting 会分析出模块之间的依赖关系，尽可能的把打包出来的模块合并到一个函数中去。

（3）Tree Shaking

Tree Shaking 可以实现删除项目中未被引用的代码。可以通过在启动webpack时追加参数 --optimize-minimize 来实现

8.3 如何用webpack来优化前端性能？

用webpack优化前端性能是指优化webpack的输出结果，让打包的最终结果在浏览器运行快速高效。

- **压缩代码**：删除多余的代码、注释、简化代码的写法等等方式。可以利用webpack的 UglifyJsPlugin 和 ParallelUglifyPlugin 来压缩JS文件， 利用 cssnano （css-loader?minimize）来压缩css
- **利用CDN加速**: 在构建过程中，将引用的静态资源路径修改为CDN上对应的路径。可以利用webpack对于 output 参数和各loader的 publicPath 参数来修改资源路径
- **Tree Shaking**: 将代码中永远不会走到的片段删除掉。可以通过在启动webpack时追加参数 --optimize-minimize 来实现
- **Code Splitting (自动):** 将代码按路由维度或者组件分块(chunk),这样做到按需加载,同时可以充分利用浏览器缓存
- **提取公共第三方库**: SplitChunksPlugin插件来进行公共模块抽取,利用浏览器缓存可以长期缓存这些无需频繁变动的公共代码

8.4 如何提高webpack的构建速度？

- 多入口情况下，使用 CommonsChunkPlugin 来提取公共代码
- 通过 **externals** 配置来提取常用库
- 利用 DllPlugin 和 DllReferencePlugin 预编译资源模块 通过 DllPlugin 来对那些我们引用但是绝对不会修改的npm包来进行预编译，再通过 DllReferencePlugin 将预编译的模块加载进来。
- 使用 Happypack 实现多线程加速编译
- 使用 webpack-uglify-parallel 来提升 uglifyPlugin 的压缩速度。 原理上 webpack-uglify-parallel 采用了多核并行压缩来提升压缩速度
- 使用 Tree-shaking 和 Scope Hoisting 来剔除多余代码

8.5 什么是长缓存？在Webpack中如何做到长缓存优化？

1、什么是长缓存

浏览器在用户访问页面的时候，为了加快加载速度，会对用户访问的静态资源进行存储，但是每一次代码升级或者更新，都需要浏览器去下载新的代码，最方便的更新方式就是引入新的文件名称，只下载新的代码块，不加载旧的代码块，这就是长缓存。

2、具体实现

在Webpack中，可以在output给出输出的文件制定chunkhash，并且分离经常更新的代码和框架代码，通NameModulesPlugin或者HashedModulesPlugin使再次打包文件名不变

8.6 怎么实现Webpack的按需加载

在Webpack中，import不仅仅是ES6module的模块导入方式，还是一个类似require的函数，我们可以通过import('module')的方式引入一个模块，import()返回的是一个Promise对象；使用import（）方式就可以实现 Webpack的按需加载

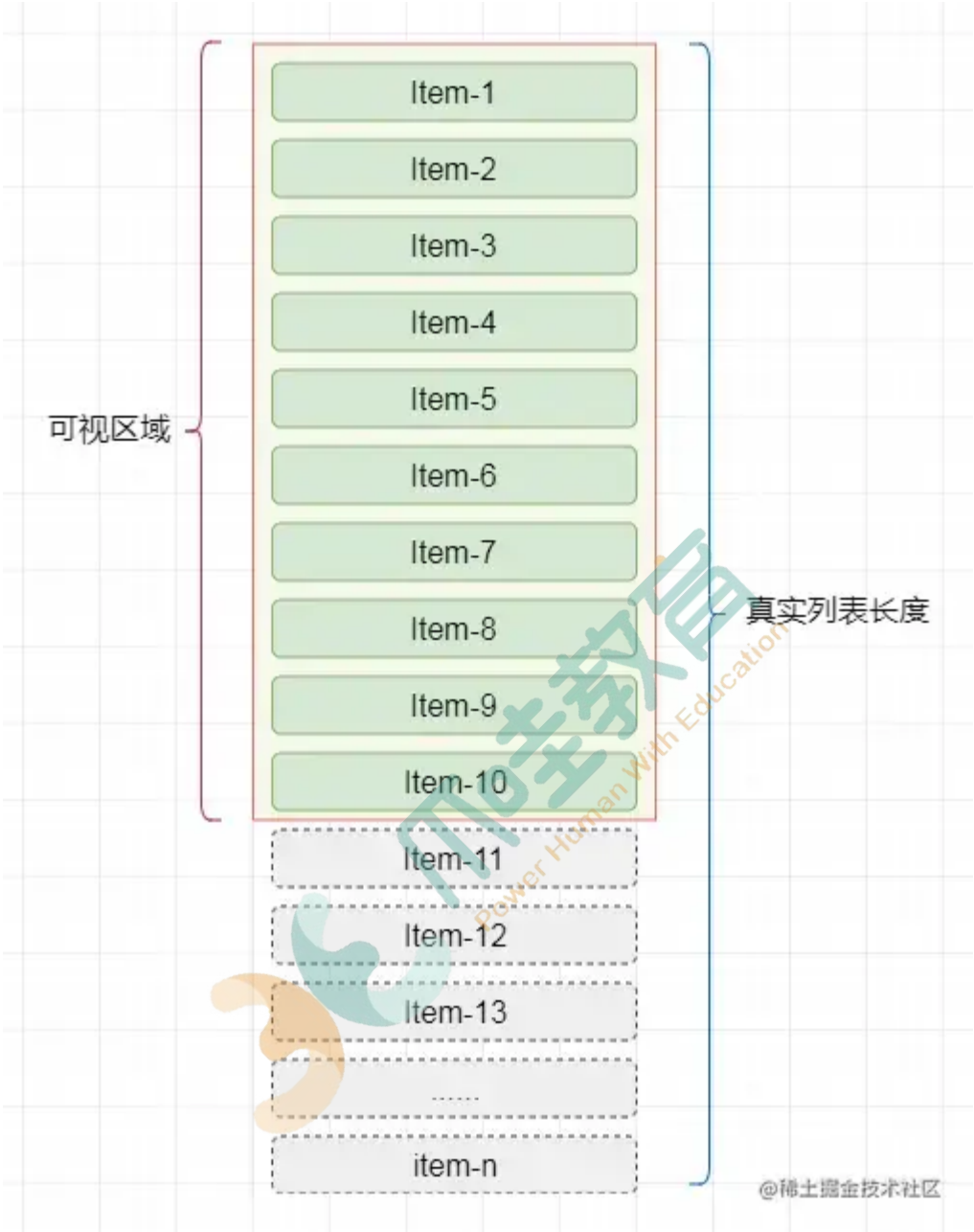
8.7 什么是神奇注释？

在import（）里可以添加一些注释，如定义该chunk的名称，要过滤的文件，指定引入的文件等等，这类带有特殊功能的注释被称为神器注释。

9\. 什么是虚拟列表

`虚拟列表` 其实是按需显示的一种实现，即**只对 `可见区域` 进行渲染，对 `非可见区域` 中的数据不渲染或部分渲染的技术，从而达到极高的渲染性能。**

假设有1万条记录需要同时渲染，我们屏幕的 `可见区域` 的高度为 `500px`,而列表项的高度为 `50px`，则此时我们在屏幕中最多只能看到10个列表项，那么在首次渲染的时候，我们只需加载10条即可。

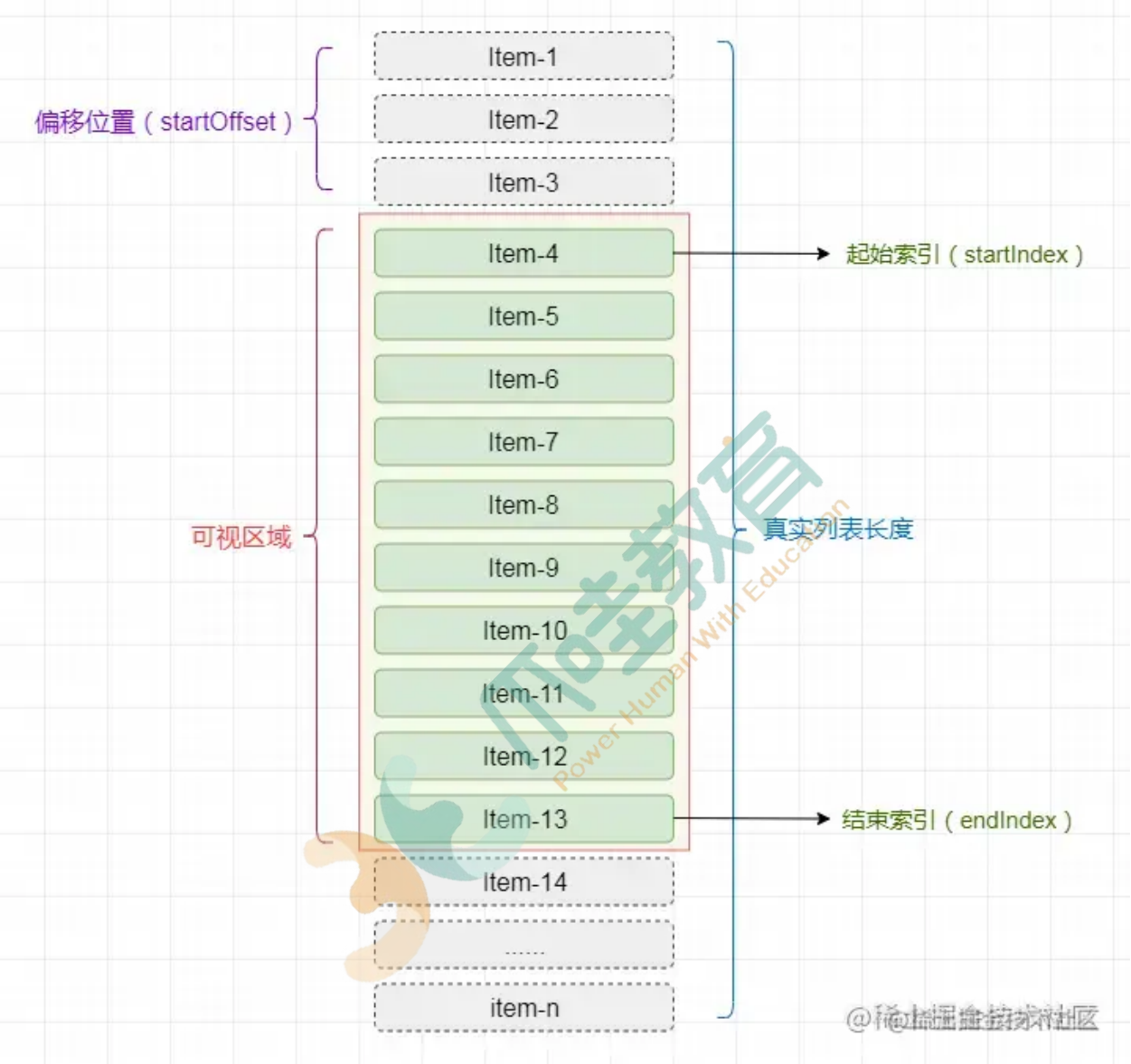


虚拟列表如何实现

虚拟列表的实现，实际上就是在首屏加载的时候，只加载`可视区域`内需要的列表项，当滚动发生时，动态通过计算获得`可视区域`内的列表项，并将`非可视区域`内存在的列表项删除。

- 计算当前`可视区域`起始数据索引(`startIndex`)
- 计算当前`可视区域`结束数据索引(`endIndex`)
- 计算当前`可视区域的`数据，并渲染到页面中

- 计算`startIndex`对应的数据在整个列表中的偏移位置`startOffset`并设置到列表上



由于只是对`可视区域`内的列表项进行渲染，所以为了保持列表容器的高度并可正常的触发滚动，将Html结构设计成如下结构：

```
<div class="infinite-list-container">
    <div class="infinite-list-phantom"></div>
    <div class="infinite-list">
```

```
    </div>
</div>
```

- `infinite-list-container` 为 `可视区域` 的容器
- `infinite-list-phantom` 为容器内的占位，高度为总列表高度，用于形成滚动条
- `infinite-list` 为列表项的 `渲染区域`

接着，监听 `infinite-list-container` 的 `scroll` 事件，获取滚动位置 `scrollTop`

- 假定 `可视区域` 高度固定，称之为 `screenHeight`
- 假定 `列表每项` 高度固定，称之为 `itemSize`
- 假定 `列表数据` 称之为 `listData`
- 假定 `当前滚动位置` 称之为 `scrollTop`

则可推算出：

- 列表总高度 `listHeight` = listData.length * itemSize
- 可显示的列表项数 `visibleCount` = Math.ceil(screenHeight / itemSize)
- 数据的起始索引 `startIndex` = Math.floor(scrollTop / itemSize)
- 数据的结束索引 `endIndex` = startIndex + visibleCount
- 列表显示数据为 `visibleData` = listData.slice(startIndex,endIndex)

当滚动后，由于 `渲染区域` 相对于 `可视区域` 已经发生了偏移，此时我需要获取一个偏移量 `startOffset`，通过样式控制将 `渲染区域` 偏移至 `可视区域` 中。

- 偏移量 `startOffset` = scrollTop - (scrollTop % itemSize); 用来做transform的数据

监听scoll事件的优化

我们使用 `监听scroll事件` 的方式来触发可视区域中数据的更新，当滚动发生后，scroll事件会频繁触发，很多时候会造成 `重复计算` 的问题，从性能上来说无疑存在浪费的情况。

可以使用 `IntersectionObserver` 替换监听scroll事件，`IntersectionObserver` 可以监听目标元素是否出现在可视区域内，在监听的回调事件中执行可视区域数据的更新，并且

`IntersectionObserver`的监听回调是异步触发，不随着目标元素的滚动而触发，性能消耗极低。

10\. 前端工程化

Babel的原理是什么？

Babel 的主要工作是对代码进行转译。 (解决兼容, 解析执行一部分代码)

```
let a = 1 + 1    =>  var a = 2
```

转译分为三阶段：

- 解析（Parse），将代码解析生成抽象语法树 AST，也就是词法分析与语法分析的过程
- 转换（Transform），对语法树进行变换方面的一系列操作。通过 babel-traverse，进行遍历并作添加、更新、删除等操作
- 生成（Generate），通过 babel-generator 将变换后的 AST 转换为 JS 代码

我们可以通过 AST Explorer 工具来查看 Babel 具体生成的 AST 节点。

11\. 什么是单点登录？

单点登录（Single Sign On），简称为 SSO，是目前比较流行的企业业务整合的解决方案之一

SSO的定义是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统

SSO 一般都需要一个独立的认证中心（passport），子系统的登录均得通过`passport`，子系统本身将不参与登录操作

当一个系统成功登录以后，`passport`将会颁发一个令牌给各个子系统，子系统可以拿着令牌会获取各自的受保护资源，为了减少频繁认证，各个子系统在被`passport`授权以后，会建立一个局部会话，在一定时间内可以无需再次向`passport`发起认证

12\. 大文件上传如何做分片上传、断点继传？

分片上传

分片上传，就是将所要上传的文件，按照一定的大小，将整个文件分隔成多个数据块（Part）来进行分片上传

如下图

上传完之后再由服务端对所有上传的文件进行汇总整合成原始的文件

大致流程如下：

1. 将需要上传的文件按照一定的分割规则，分割成相同大小的数据块；
2. 初始化一个分片上传任务，返回本次分片上传唯一标识；
3. 然后借助 http 的可并发性，同时上传多个切片
4. 发送完成后，服务端根据判断数据上传是否完整，如果完整，则进行数据块合成得到原始文件

断点续传

断点续传指的是在下载或上传时，将下载或上传任务人为的划分为几个部分

每一个部分采用一个线程进行上传或下载，如果碰到网络故障，可以从已经上传或下载的部分开始继续上传下载未完成的部分，而没有必要从头开始上传下载。用户可以节省时间，提高速度

一般实现方式有两种：

- 服务器端返回，告知从哪开始
- 浏览器端自行处理

上传过程中将文件在服务器写为临时文件，等全部写完了（文件上传完），将此临时文件重命名为正式文件即可

如果中途上传中断过，下次上传的时候根据当前临时文件大小，作为在客户端读取文件的偏移量，从此位置继续读取文件数据块，上传到服务器从此偏移量继续写入文件即可

使用场景

- 大文件加速上传：当文件大小超过预期大小时，使用分片上传可实现并行上传多个Part， 以加快上传速度
- 网络环境较差：建议使用分片上传。当出现上传失败的时候，仅需重传失败的Part
- 流式上传：可以在需要上传的文件大小还不确定的情况下开始上传。这种场景在视频监控等行业应用中比较常见

13\. npm run dev的时候webpack做了什么事情

执行**npm run dev**时候最先执行的**build/dev-server.js**文件，该文件主要完成下面几件事情：

1、检查node和npm的**版本**、引入相关**插件**和**配置**

2、webpack对源码进行**编译打包**并返回compiler对象

3、**创建express服务器**

4、**配置开发中间件**（webpack-dev-middleware）和**热重载中间件**（webpack-hot-middleware）

5、**挂载代理服务和中间件**

6、配置静态资源

7、启动服务器监听特定端口（8080）

8、自动打开浏览器并打开特定网址（localhost:8080）