`this.cache` 对象中没有该 `key` 值的情况，如下：

```
/* 如果没有命中缓存，则将其设置进缓存 */
else {
    cache[key] = vnode
    keys.push(key)
    /* 如果配置了max并且缓存的长度超过了this.max，则从缓存中删除第一个 */
    if (this.max && keys.length > parseInt(this.max)) {
        pruneCacheEntry(cache, keys[0], keys, this._vnode)
    }
}
```

表明该组件还没有被缓存过，则以该组件的 `key` 为键，组件 `vnode` 为值，将其存入 `this.cache` 中，并且把 `key` 存入 `this.keys` 中

此时再判断 `this.keys` 中缓存组件的数量是否超过了设置的最大缓存数量值 `this.max` ，如果超过了，则把第一个缓存组件删掉

24.4. 思考题：缓存后如何获取数据

解决方案可以有以下两种：

- beforeRouteEnter
- actived

24.4.1. beforeRouteEnter

每次组件渲染的时候，都会执行 `beforeRouteEnter`

```
beforeRouteEnter(to, from, next){
    next(vm=>{
        console.log(vm)
        // 每次进入路由执行
        vm.getData()  // 获取数据
    })
},
```

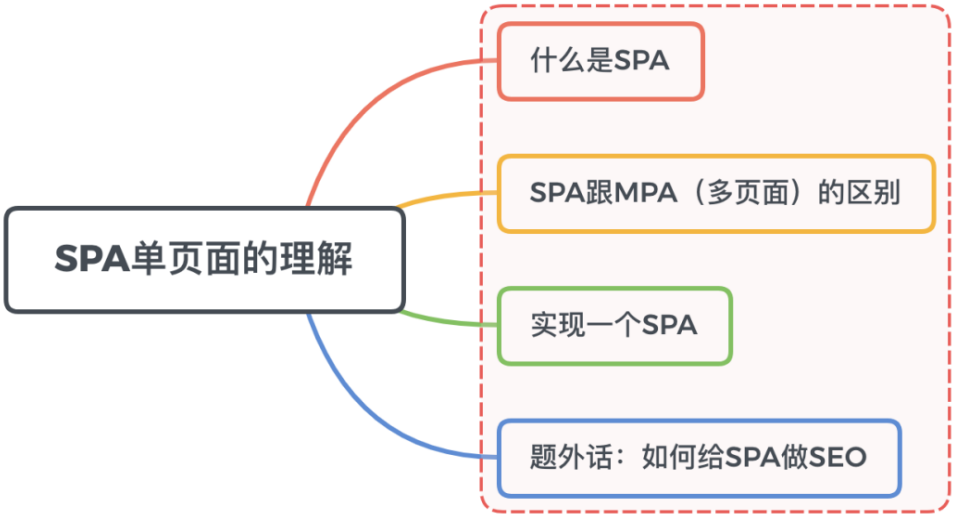
24.4.2. actived

在 `keep-alive` 缓存的组件被激活的时候，都会执行 `actived` 钩子

```Go
activated(){
    this.getData() // 获取数据
},
```

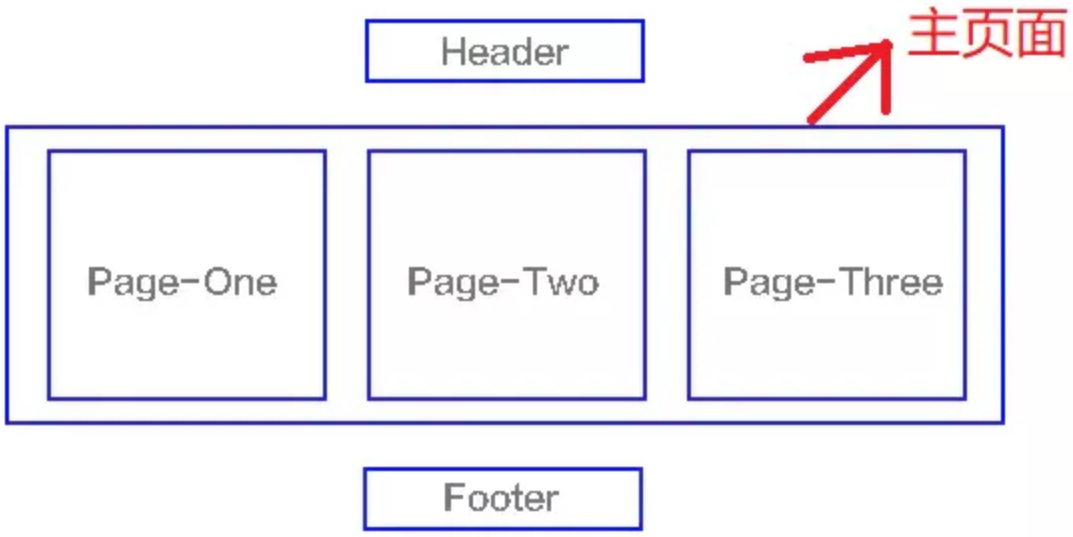
注意：服务器端渲染期间 `avtived` 不被调用

25. 你对SPA单页面的理解，它的优缺点分别是什么？如何实现SPA应用呢



25.1. 什么是SPA

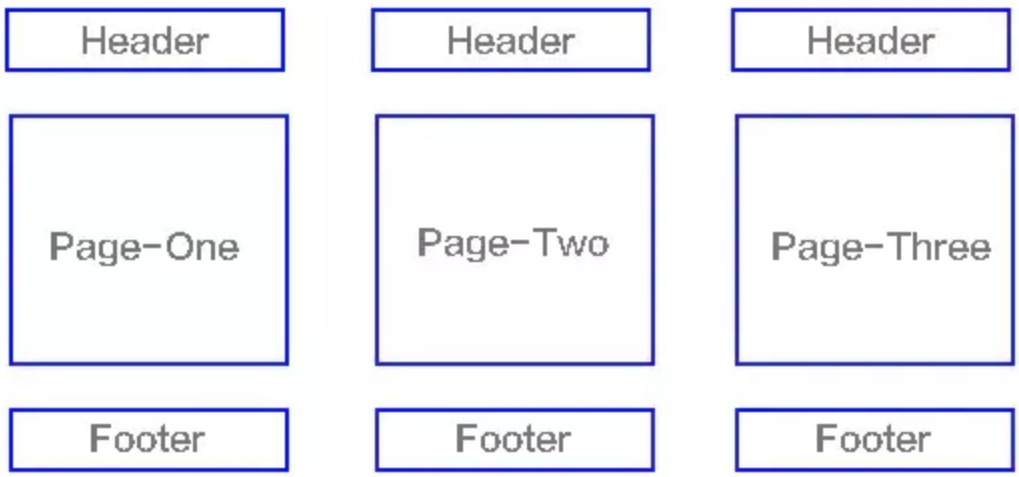
SPA（single-page application），翻译过来就是单页应用 `SPA` 是一种网络应用程序或网站的模型，它通过动态重写当前页面来与用户交互，这种方法避免了页面之间切换打断用户体验在单页应用中，所有必要的代码（ `HTML` 、 `JavaScript` 和 `CSS` ）都通过单个页面的加载而检索，或者根据需要（通常是为响应用户操作）动态装载适当的资源并添加到页面页面在任何时间点都不会重新加载，也不会将控制转移到其他页面举个例子来讲就是一个杯子，早上装的牛奶，中午装的是开水，晚上装的是茶，我们发现，变的始终是杯子里的内容，而杯子始终是那个杯子结构如下图



我们熟知的JS框架如 `react` , `vue` , `angular` , `ember` 都属于 `SPA`

25.2. SPA和MPA的区别

上面大家已经对单页面有所了解了，下面来讲讲多页应用MPA（MultiPage-page application），翻译过来就是多页应用在 `MPA` 中，每个页面都是一个主页面，都是独立的当我们在访问另一个页面的时候，都需要重新加载 `html` 、 `css` 、 `js` 文件，公共文件则根据需求按需加载如下图



25.2.1.1. 单页应用与多页应用的区别

	单页面应用（SPA）	多页面应用（MPA）
组成	一个主页面和多个页面片段	多个主页面

刷新方式	局部刷新	整页刷新
url模式	哈希模式	历史模式
SEO搜索引擎优化	难实现，可使用SSR方式改善	容易实现
数据传递	容易	通过url、cookie、localStorage等传递
页面切换	速度快，用户体验良好	切换加载资源，速度慢，用户体验差
维护成本	相对容易	相对复杂

25.2.1.2. 单页应用优缺点

优点：

- 具有桌面应用的即时性、网站的可移植性和可访问性
- 用户体验好、快，内容的改变不需要重新加载整个页面
- 良好的前后端分离，分工更明确

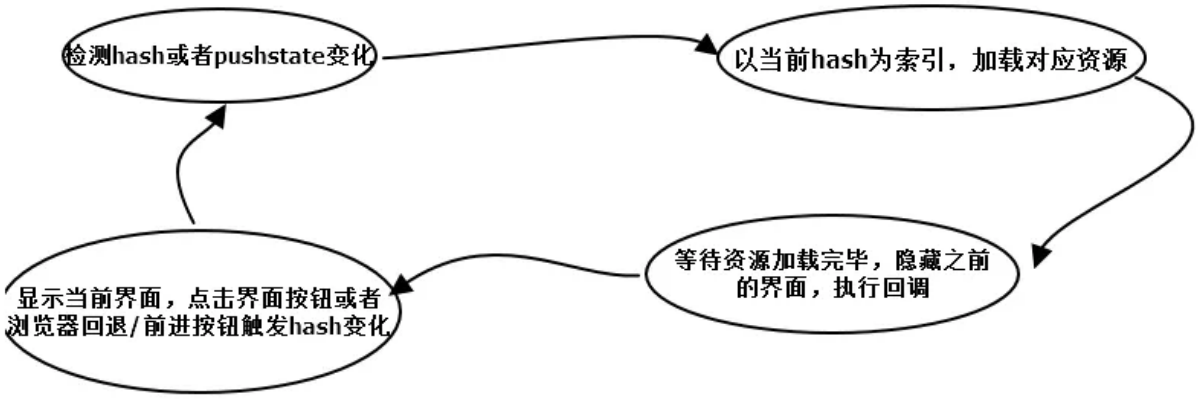
缺点：

- 不利于搜索引擎的抓取
- 首次渲染速度相对较慢

25.3. 实现一个SPA

25.3.1.1. 原理

1. 监听地址栏中 `hash` 变化驱动界面变化
2. 用 `pushsate` 记录浏览器的历史，驱动界面发送变化



25.3.1.2. 实现

25.3.1.2.1. `hash` 模式

核心通过监听 `url` 中的 `hash` 来进行路由跳转

JavaScript 复制代码

```javascript
// 定义 Router
class Router {
    constructor () {
        this.routes = {}; // 存放路由path及callback
        this.currentUrl = '';

        // 监听路由change调用相对应的路由回调
        window.addEventListener('load', this.refresh, false);
        window.addEventListener('hashchange', this.refresh, false);
    }

    route(path, callback){
        this.routes[path] = callback;
    }

    push(path) {
        this.routes[path] && this.routes[path]()
    }
}

// 使用 router
window.miniRouter = new Router();
miniRouter.route('/', () => console.log('page1'))
miniRouter.route('/page2', () => console.log('page2'))

miniRouter.push('/') // page1
miniRouter.push('/page2') // page2
```

25.3.1.2.2. history模式

`history` 模式核心借用 `HTML5 history api`， `api` 提供了丰富的 `router` 相关属性先了解一个几个相关的api

- `history.pushState` 浏览器历史纪录添加记录
- `history.replaceState` 修改浏览器历史纪录中当前纪录
- `history.popState` 当 `history` 发生变化时触发

JavaScript | 复制代码

```javascript
// 定义 Router
class Router {
    constructor () {
        this.routes = {};
        this.listerPopState()
    }

    init(path) {
        history.replaceState({path: path}, null, path);
        this.routes[path] && this.routes[path]();
    }

    route(path, callback){
        this.routes[path] = callback;
    }

    push(path) {
        history.pushState({path: path}, null, path);
        this.routes[path] && this.routes[path]();
    }

    listerPopState () {
        window.addEventListener('popstate' , e => {
            const path = e.state && e.state.path;
            this.routers[path] && this.routers[path]()
        })
    }
}

// 使用 Router

window.miniRouter = new Router();
miniRouter.route('/', ()=> console.log('page1'))
miniRouter.route('/page2', ()=> console.log('page2'))

// 跳转
miniRouter.push('/page2')  // page2
```

25.3.1. 如何给SPA做SEO

下面给出基于 `Vue` 的 `SPA` 如何实现 `SEO` 的三种方式

1. SSR服务端渲染

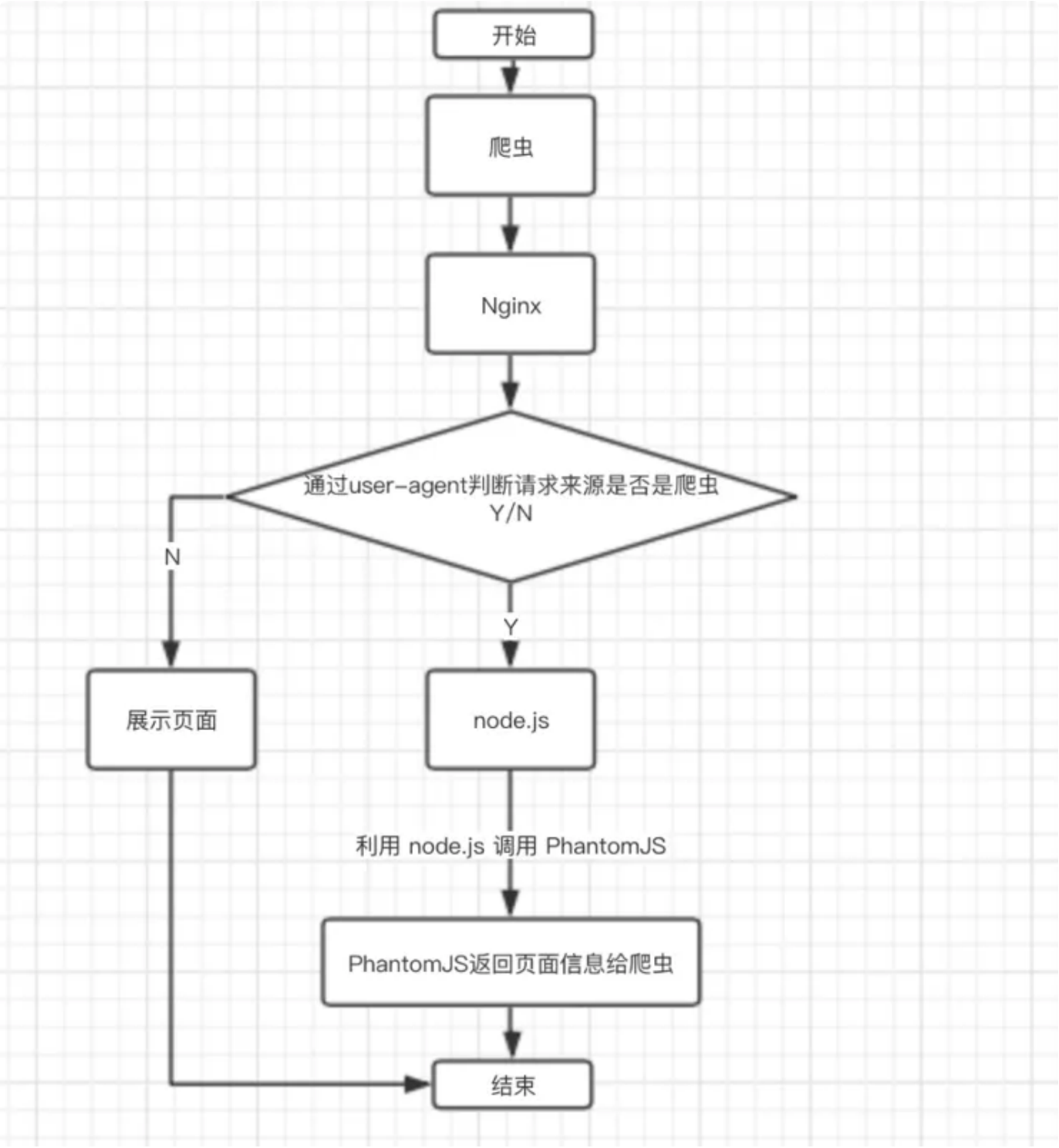
将组件或页面通过服务器生成html，再返回给浏览器，如 `nuxt.js`

2. 静态化

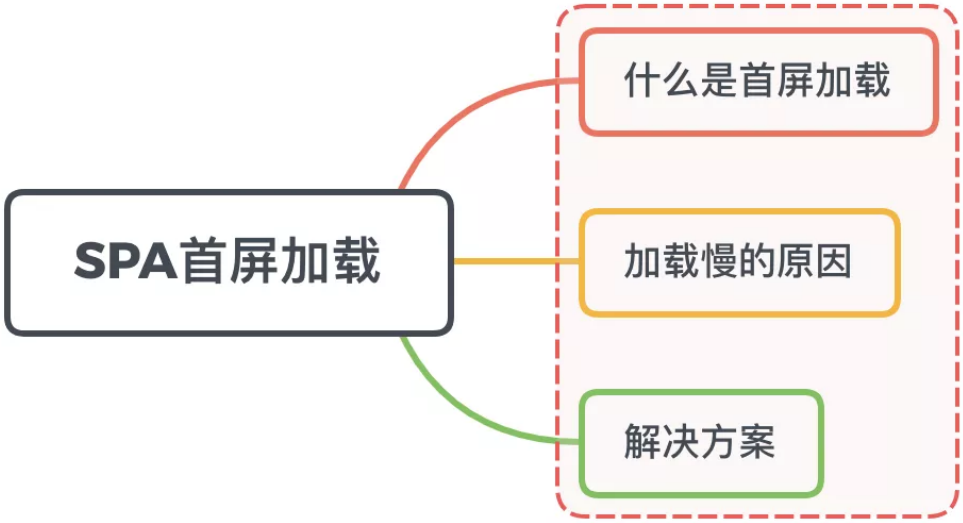
目前主流的静态化主要有两种： （1）一种是通过程序将动态页面抓取并保存为静态页面，这样的页面的实际存在于服务器的硬盘中（2）另外一种是通过WEB服务器的 `URL Rewrite` 的方式，它的原理是通过web服务器内部模块按一定规则将外部的URL请求转化为内部的文件地址，一句话来说就是把外部请求的静态地址转化为实际的动态页面地址，而静态页面实际是不存在的。这两种方法都达到了实现URL静态化的效果

3. 使用 `Phantomjs` 针对爬虫处理

原理是通过 `Nginx` 配置，判断访问来源是否为爬虫，如果是则搜索引擎的爬虫请求会转发到一个 `node server` ，再通过 `PhantomJS` 来解析完整的 `HTML` ，返回给爬虫。下面是大致流程图



26. SPA首屏加载速度慢的怎么解决?



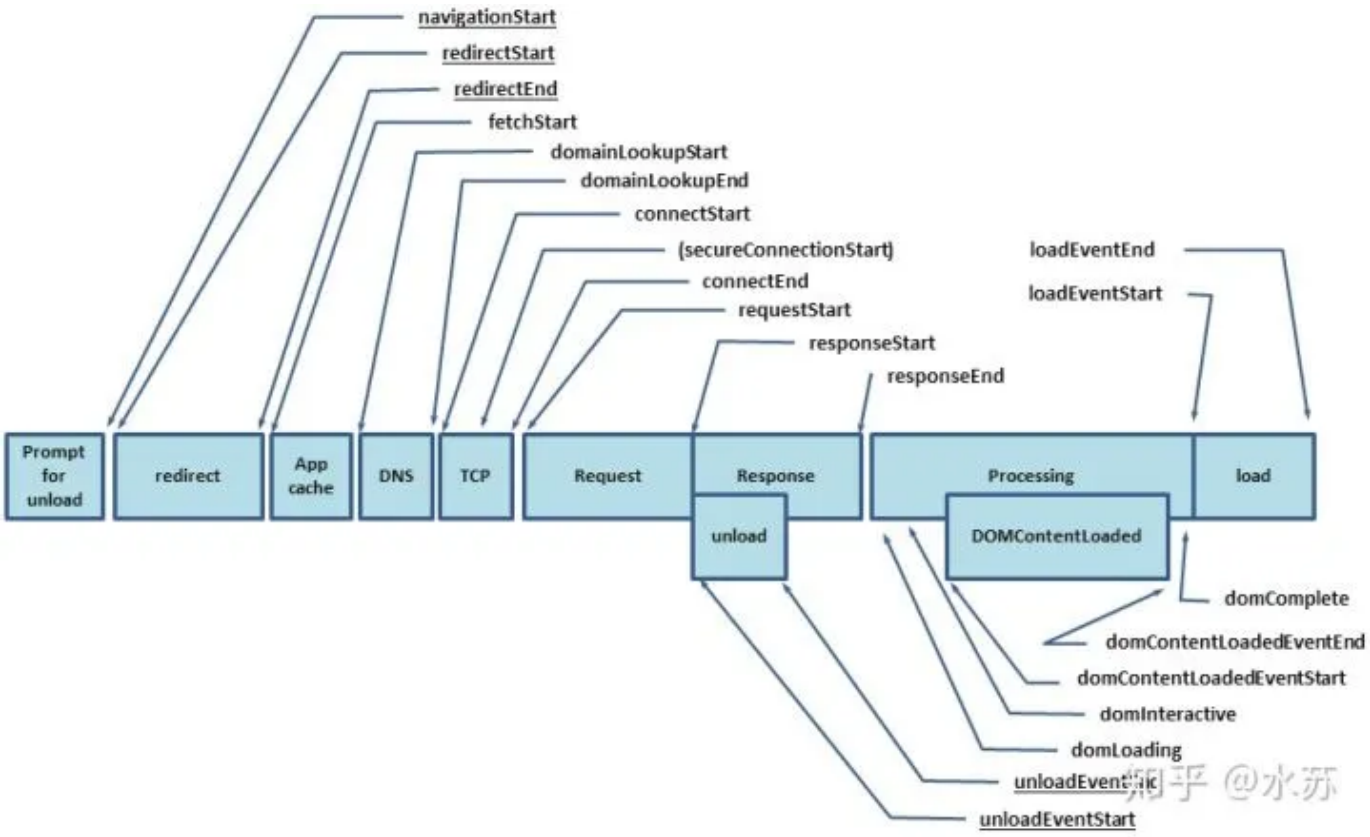
26.1. 什么是首屏加载

首屏时间（First Contentful Paint），指的是浏览器从响应用户输入网址地址，到首屏内容渲染完成的时间，此时整个网页不一定要全部渲染完成，但需要展示当前视窗需要的内容

首屏加载可以说是用户体验中**最重要**的环节

26.1.1. 关于计算首屏时间

利用 `performance.timing` 提供的数据：



通过 `DOMContentLoad` 或者 `performance` 来计算出首屏时间

JavaScript | 复制代码

```
// 方案一:
document.addEventListener('DOMContentLoaded', (event) => {
    console.log('first contentful painting');
});
// 方案二:
performance.getEntriesByName("first-contentful-paint")[0].startTime

// performance.getEntriesByName("first-contentful-paint")[0]
// 会返回一个 PerformancePaintTiming的实例，结构如下:
{
  name: "first-contentful-paint",
  entryType: "paint",
  startTime: 507.80000002123415,
  duration: 0,
};
```

26.2. 加载慢的原因

在页面渲染的过程，导致加载速度慢的因素可能如下:

- 网络延时问题
- 资源文件体积是否过大
- 资源是否重复发送请求去加载了
- 加载脚本的时候，渲染内容堵塞了

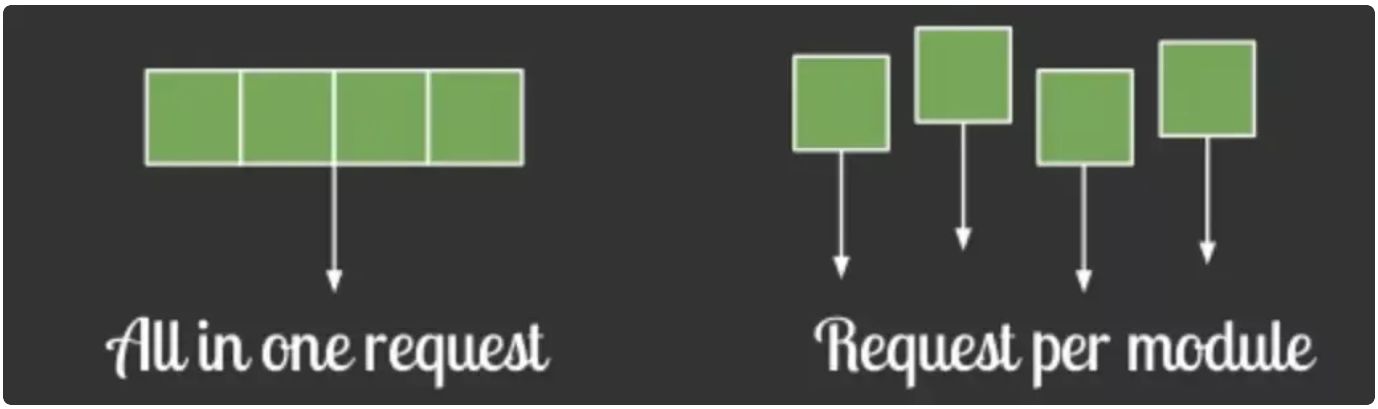
26.3. 解决方案

常见的几种SPA首屏优化方式

- 减小入口文件积
- 静态资源本地缓存
- UI框架按需加载
- 图片资源的压缩
- 组件重复打包
- 开启GZip压缩
- 使用SSR

26.3.1. 减小入口文件体积

常用的手段是路由懒加载，把不同路由对应的组件分割成不同的代码块，待路由被请求的时候会单独打包路由，使得入口文件变小，加载速度大大增加



在 `vue-router` 配置路由的时候，采用动态加载路由的形式

```JavaScript
routes:[
    path: 'Blogs',
    name: 'ShowBlogs',
    component: () => import('./components/ShowBlogs.vue')
]
```

以函数的形式加载路由，这样就可以把各自的路由文件分别打包，只有在解析给定的路由时，才会加载路由组件

26.3.2. 静态资源本地缓存

后端返回资源问题：

- 采用 `HTTP` 缓存，设置 `Cache-Control` ， `Last-Modified` ， `Etag` 等响应头
- 采用 `Service Worker` 离线缓存

前端合理利用 `localStorage`

26.3.3. UI框架按需加载

在日常使用 `UI` 框架，例如 `element-UI` 、或者 `antd` ，我们经常性直接引用整个 `UI` 库

```JavaScript
import ElementUI from 'element-ui'
Vue.use(ElementUI)
```

但实际上我用到的组件只有按钮，分页，表格，输入与警告 所以我们要按需引用

```JavaScript
import { Button, Input, Pagination, Table, TableColumn, MessageBox } from 'element-ui';
Vue.use(Button)
Vue.use(Input)
Vue.use(Pagination)
```

26.3.4. 组件重复打包

假设 `A.js` 文件是一个常用的库，现在有多个路由使用了 `A.js` 文件，这就造成了重复下载

解决方案：在 `webpack` 的 `config` 文件中，修改 `CommonsChunkPlugin` 的配置

```JavaScript
minChunks: 3
```

`minChunks` 为3表示会把使用3次及以上的包抽离出来，放进公共依赖文件，避免了重复加载组件

26.3.5. 图片资源的压缩

图片资源虽然不在编码过程中，但它却是对页面性能影响最大的因素

对于所有的图片资源，我们可以进行适当的压缩

对页面上使用到的 `icon` ，可以使用在线字体图标，或者雪碧图，将众多小图标合并到同一张图上，用以减轻 `http` 请求压力。

26.3.6. 开启GZip压缩

拆完包之后，我们再用 `gzip` 做一下压缩 安装 `compression-webpack-plugin`

```JavaScript
cnmp i compression-webpack-plugin -D
```

在 `vue.congig.js` 中引入并修改 `webpack` 配置

```JavaScript
const CompressionPlugin = require('compression-webpack-plugin')

configureWebpack: (config) => {
        if (process.env.NODE_ENV === 'production') {
            // 为生产环境修改配置...
            config.mode = 'production'
            return {
                plugins: [new CompressionPlugin({
                    test: /\.js$|\.html$|\.css/, //匹配文件名
                    threshold: 10240, //对超过10k的数据进行压缩
                    deleteOriginalAssets: false //是否删除原文件
                })]
            }
        }
```

在服务器我们也要做相应的配置 如果发送请求的浏览器支持 `gzip` ，就发送给它 `gzip` 格式的文件 我的服务器是用 `express` 框架搭建的 只要安装一下 `compression` 就能使用

```
Plain Text | 复制代码
```

```
const compression = require('compression')
app.use(compression())  // 在其他中间件使用之前调用
```

26.3.7. 使用SSR

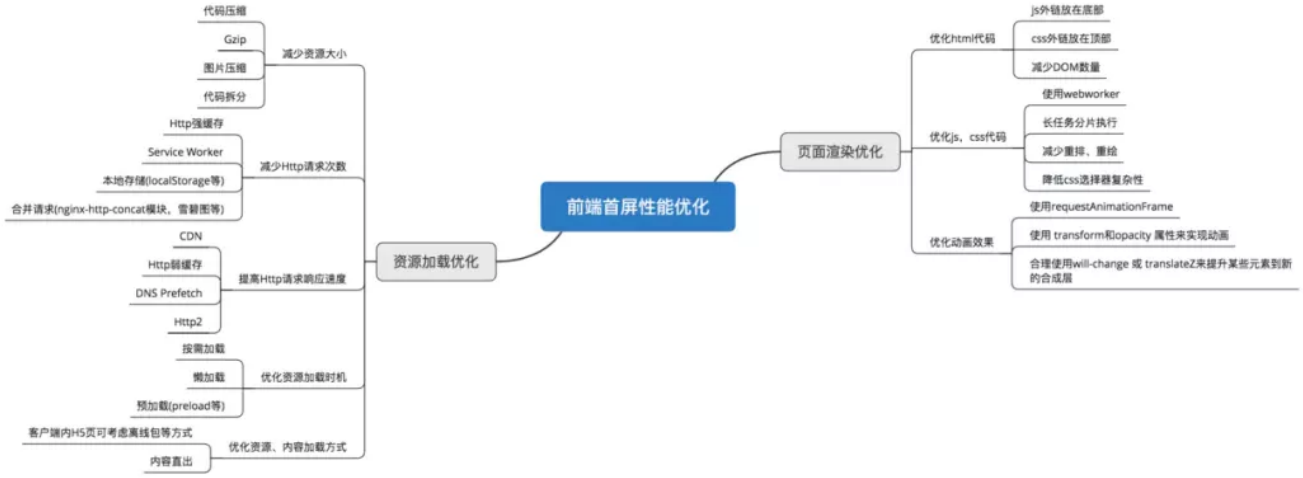
SSR（Server side ），也就是服务端渲染，组件或页面通过服务器生成html字符串，再发送到浏览器

从头搭建一个服务端渲染是很复杂的， `vue` 应用建议使用 `Nuxt.js` 实现服务端渲染

26.4. 小结

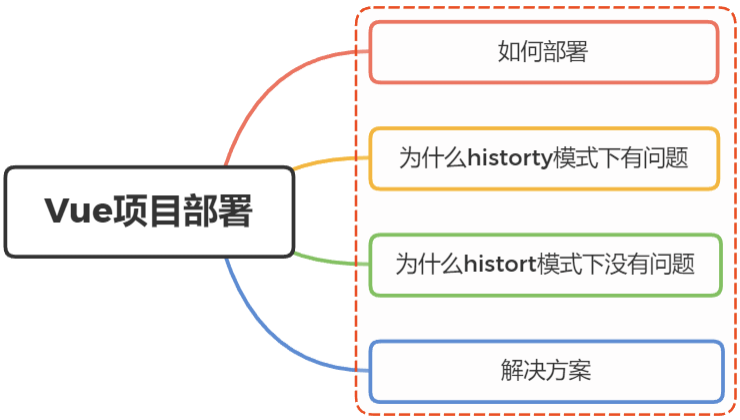
减少首屏渲染时间的方法有很多，总的来讲可以分成两大部分 ： 资源加载优化 和 页面渲染优化

下图是更为全面的首屏优化的方案



大家可以根据自己项目的情况选择各种方式进行首屏渲染的优化

27. vue项目本地开发完成后部署到服务器后报404是什么原因呢?



27.1. 如何部署

前后端分离开发模式下，前后端是独立布署的，前端只需要将最后的构建物上传至目标服务器的 `web` 容器指定的静态目录下即可

我们知道 `vue` 项目在构建后，是生成一系列的静态文件

常规布署我们只需要将这个目录上传至目标服务器即可

Bash | 复制代码

```
// scp 上传 user为主机登录用户，host为主机外网ip，xx为web容器静态资源路径
scp dist.zip user@host:/xx/xx/xx
```

让 `web` 容器跑起来，以 `nginx` 为例

Bash | 复制代码

```
server {
    listen  80;
    server_name  www.xxx.com;

    location / {
      index  /data/dist/index.html;
    }
}
```

配置完成记得重启 `nginx`

```bash
// 检查配置是否正确
nginx -t

// 平滑重启
nginx -s reload
```

操作完后就可以在浏览器输入域名进行访问了

当然上面只是提到最简单也是最直接的一种布署方式

什么自动化，镜像，容器，流水线布署，本质也是将这套逻辑抽象，隔离，用程序来代替重复性的劳动，本文不展开

27.2. 404问题

这是一个经典的问题，相信很多同学都有遇到过，那么你知道其真正的原因吗?

我们先还原一下场景:

- `vue` 项目在本地时运行正常，但部署到服务器中，刷新页面，出现了404错误

先定位一下，HTTP 404 错误意味着链接指向的资源不存在

问题在于为什么不存在？且为什么只有 `history` 模式下会出现这个问题?

27.2.1. 为什么history模式下有问题

`Vue` 是属于单页应用（single-page application）

而 `SPA` 是一种网络应用程序或网站的模型，所有用户交互是通过动态重写当前页面，前面我们也看到了，不管我们应用有多少页面，构建物都只会产出一个 `index.html`

现在，我们回头来看一下我们的 `nginx` 配置