[一、浏览器加载、白屏期间（发生什么） 1](#_Toc15357)

[二、CDN原理 1](#_Toc31814)

[三、Webpack配置首屏loading图 3](#_Toc6379)

[四、http1和http2 3](#_Toc22252)

[五、 骨架屏 4](#_Toc7115)

[六、Polyfill 4](#_Toc24693)

# 一、浏览器加载、白屏期间（发生什么）

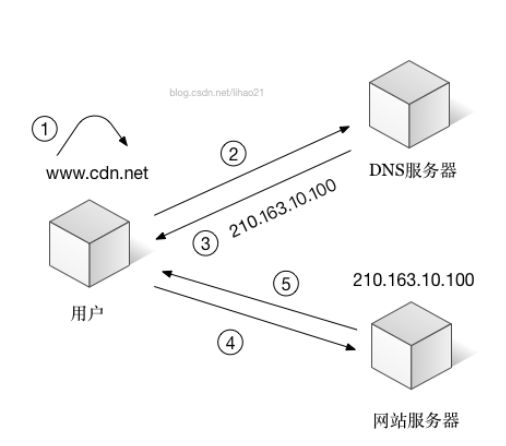
1、回车按下,浏览器解析网址,进行 DNS 查询,查询返回 IP,通过 IP 发出 HTTP(S) 请求。

2、服务器返回HTML,浏览器开始解析 HTML,此时触发请求 js 和 css 资源。

3、js 被加载,开始执行 js,调用各种函数创建 DOM 并渲染到根节点,直到第一个可见元素产生。

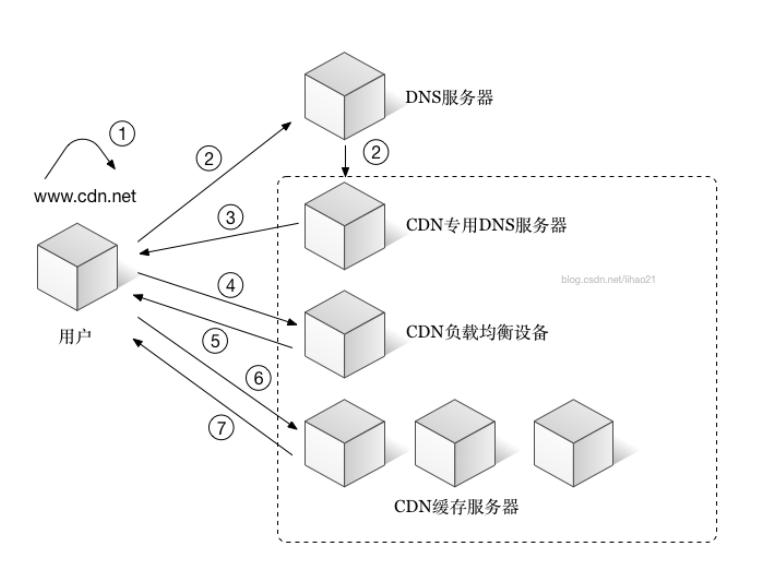
# 二、CDN原理

**未使用cdn，正常情况下：**



1. 用户在浏览器中输入要访问的域名。
2. 浏览器向DNS服务器请求对该域名的解析。
3. DNS服务器返回该域名的IP地址给浏览器。
4. 浏览器使用该IP地址向服务器请求内容。
5. 服务器将用户请求的内容返回给浏览器。

**使用了CDN：**



1、用户在浏览器中输入要访问的域名。

2、浏览器向DNS服务器请求对域名进行解析。由于CDN对域名解析进行了调整，DNS服务器会最终将域名的解析权交给CNAME指向的CDN专用DNS服务器。

3、CDN的DNS服务器将CDN的负载均衡设备IP地址返回给用户。

4、用户向CDN的负载均衡设备发起内容URL访问请求。

5、CDN负载均衡设备会为用户选择一台合适的缓存服务器提供服务。  
选择的依据包括：  
 根据用户IP地址，判断哪一台服务器距离用户最近；  
 根据用户所请求的URL中携带的内容名称，判断哪一台服务器上有用户所需内容；  
 查询各个服务器的负载情况，判断哪一台服务器的负载较小。  
 基于以上这些依据的综合分析之后，负载均衡设置会把缓存服务器的IP地址返回给用户。

6、用户向缓存服务器发出请求。

7、缓存服务器响应用户请求，将用户所需内容传送到用户。  
如果这台缓存服务器上并没有用户想要的内容，而负载均衡设备依然将它分配给了用户，那么这台服务器就要向它的上一级缓存服务器请求内容，直至追溯到网站的源服务器将内容拉取到本地。

# 三、Webpack配置首屏loading图

我们一般有一款 webpack 插件叫[html-webpack-plugin](https://github.com/jantimon/html-webpack-plugin" \t "https://segmentfault.com/a/_blank) ,在其中配置 html 就可以在文件中插入 loading 图。

const HtmlWebpackPlugin = require('html-webpack-plugin')

const loading = require('./render-loading') // 事先设计好的 loading 图
module.exports = {
entry: './src/index.js',
output: {
path: \_\_dirname + '/dist',
filename: 'index\_bundle.js'
},
plugins: [
new HtmlWebpackPlugin({
template: './src/index.html',
loading: loading
})
]
}

# 四、http1和http2

1. http2采用二进制分帧的方式进行通信,而 http1.x 是用文本,http2 的效率更高
2. http2 可以进行多路复用,即跟同一个域名通信,仅需要一个 TCP 建立请求通道,请求与响应可以同时基于此通道进行双向通信,而 http1.x 每次请求需要建立 TCP,多次请求需要多次连接,还有并发限制,十分耗时
3. http2 可以头部压缩,能够节省消息头占用的网络的流量,而HTTP/1.x每次请求，都会携带大量冗余头信息，浪费了很多带宽资源
4. http2可以进行服务端推送,我们平时解析 HTML 后碰到相关标签才会进而请求 css 和 js 资源,而 http2 可以直接将相关资源直接推送,无需请求,这大大减少了多次请求的耗时

# 骨架屏

React: antd 内置的骨架图[Skeleton](https://ant.design/components/skeleton-cn/" \t "https://segmentfault.com/a/_blank)方案  
Vue: [vue-skeleton-webpack-plugin](https://github.com/lavas-project/vue-skeleton-webpack-plugin" \t "https://segmentfault.com/a/_blank)

以 vue-cli 3 为例,我们可以直接在vue.config.js 中配置

//引入插件const SkeletonWebpackPlugin = require('vue-skeleton-webpack-plugin');
module.exports = {
// 额外配置参考官方文档
configureWebpack: (config)=>{
config.plugins.push(new SkeletonWebpackPlugin({
webpackConfig: {
entry: {
app: path.join(\_\_dirname, './src/Skeleton.js'),
},
},
minimize: true,
quiet: true,
}))
},
//这个是让骨架屏的css分离，直接作为内联style处理到html里，提高载入速度
css: {
extract: true,
sourceMap: false,
modules: false
}
}

# 六、Polyfill

Polyfill是一个js库，主要抚平不同浏览器之间对js实现的差异,用来为旧浏览器提供它没有原生支持的较新的功能。

比如，html5的storage(session,local), 不同浏览器，不同版本，有些支持，有些不支持。

Polyfill（Polyfill有很多，在GitHub上https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills），帮你把这些差异化抹平，不支持的变得支持了（典型做法是在IE浏览器中增加 window.XMLHttpRequest ，内部实现使用 ActiveXObject。）

**动态加载polyfill,避免不需要时也加载**

是否需要 polyfill 应该有客户端的浏览器自己决定,而不是开发者决定,但是我们在很长一段时间里都是开发者将各种 polyfill 打包,其实很多情况下导致用户加载了根本没有必要的代码.

解决这个问题的方法很简单,直接引入 <script src="https://cdn.polyfill.io/v2/polyfill.min.js"></script> 即可,而对于 Vue 开发者就更友好了,vue-cli 现在生成的模板就自带这个引用.

这个原理就是服务商通过识别不同浏览器的浏览器User Agent，使得服务器能够识别客户使用的操作系统及版本、CPU 类型、浏览器及版本、浏览器渲染引擎、浏览器语言、浏览器插件等，然后根据这个信息判断是否需要加载 polyfill,开发者在浏览器的 network 就可以查看User Agent。

# 七、路由级别拆解代码（按需加载路由）

我们在上文中已经通过SplitChunksPlugin将第三方库进行了抽离,但是在首屏加载过程中依然有很多冗余代码,比如我们的首页是个登录界面,那么其实用到的代码很简单

1. 框架的基础库例如 vue redux 等等
2. ui 框架的部分 form 组件和按钮组件等等
3. 一个简单的布局组件
4. 其它少量逻辑和样式

登录界面的代码是很少的,为什么不只加载登录界面的代码呢?  
这就需要我们进行对代码在路由级别的拆分,除了基础的框架和 UI 库之外,我们只需要加载当前页面的代码即可,这就有得用到Code Splitting技术进行代码分割,我们要做的其实很简单.  
我们得先给 babel 设置plugin-syntax-dynamic-import这个动态import 的插件,然后就可以就函数体内使用 import 了.

对于Vue 你可以这样引入路由

export default new Router({
routes: [
{
path: '/',
name: 'Home',
component: Home
},
{
path: '/login',
name: 'login',
component: () => import('@components/login')
}
]

**这样这个页面会被单独打包**

# 组件懒加载

其实组件分割的方法跟路由分割差不多,也是通过 lazy + Suspense 的方法进行组件懒加载

// 动态加载图表组件

const Chart = lazy(() => import(/\* webpackChunkName: 'chart' \*/'./charts'))
// 包含着图表的 modal 组件

const ModalEchart = (props) => (
<Modal
title="Basic Modal"
visible={props.visible}
onOk={props.handleOk}
onCancel={props.handleCancel}
>
<Chart />
</Modal>
)

# 组件预加载

继上一点，我们通过组件懒加载将页面的初始渲染的资源体积降低了下来,提高了加载性能,但是组件的性能又出现了问题,还是上一个 demo,我们把初始页面的 3.9m 的体积减少到了1.7m,页面的加载是迅速了,但是组件的加载却变慢了.

原因是其余的 2m 资源的压力全部压在了图表组件上(Echarts 的体积缘故),因此当我们点击菜单加载图表的时候会出现 1-2s 的 loading 延迟,

我们能不能提前把图表加载进来,避免图表渲染中加载时间过长的问题?这种提前加载的方法就是组件的预加载.

原理也很简单,就是在用户的鼠标还处于 hover 状态的时候就开始触发图表资源的加载,通常情况下当用户点击结束之后,加载也基本完成,这个时候图表会很顺利地渲染出来,不会出现延迟.

/\*\*
\* @param {\*} factory 懒加载的组件
\* @param {\*} next factory组件下面需要预加载的组件
\*/**function** **lazyWithPreload**(factory, next) {
**const** Component = lazy(factory);
Component.preload = next;
**return** Component;
}
...// 然后在组件的方法中触发预加载
**const** preloadChart = () => {
Modal.preload()
}