# 你真的了解前端路由么

## 什么是路由？

路由这个概念最先是后端出现的。在以前用模板引擎开发页面时，经常会看到这样

http://hometown.xxx.edu.cn/bbs/forum.php

有时还会有带`.asp`或`.html`的路径，这就是所谓的 SSR(Server Side Render)，通过服务端渲染，直接返回页面。

其响应过程是这样的

1.浏览器发出请求

2.服务器监听到 80 端口（或 443）有请求过来，并解析 url 路径

3.根据服务器的路由配置，返回相应信息（可以是 html 字串，也可以是 json 数据，图片等）

4.浏览器根据数据包的`Content-Type`来决定如何解析数据简单来说路由就是用来跟后端服务器进行交互的一种方式，通过不同的路径，来请求不同的资源，请求不同的页面是路由的其中一种功能。

## 前端路由的诞生

前端路由的诞生的缘由前端路由的出现要从 ajax 开始，Ajax，全称 Asynchronous JavaScript And XML，是浏览器用来实现异步加载的一种技术方案。在 90s 年代初，大多数的网页都是通过直接返回 HTML 的，用户的每次更新操作都需要重新刷新页面。及其影响交互体验，随着网络的发展，迫切需要一种方案来改善这种情况。1996，微软首先提出 iframe 标签，iframe 带来了异步加载和请求元素的概念，随后在 1998 年，微软的 Outloook Web App 团队提出 Ajax 的基本概念（XMLHttpRequest 的前身），并在 IE5 通过 ActiveX 来实现了这项技术。在微软实现这个概念后，其他浏览器比如 Mozilia，Safari，Opera 相继以 XMLHttpRequest 来实现 Ajax。（ 兼容问题从此出现）不过在 IE7 发布时，微软选择了妥协，兼容了 XMLHttpRequest 的实现。有了 Ajax 后，用户交互就不用每次都刷新页面，体验带来了极大的提升。但真正让这项技术发扬光大的，还是后来的

Google Map，它的出现向人们展现了 Ajax 的真正魅力，释放了众多开发人员的想象力，让其不仅仅局限于简单的数据和页面交互，为后来异步交互体验方式的繁荣发展带来了根基。而异步交互体验的更高级版本就是 SPA —— 单页应用。

单页应用不仅仅是在页面交互是无刷新的，连页面跳转都是无刷新的，为了实现单页应用，所以就有了前端路由。单页应用的概念是伴随着 MVVM 出现的。最早由微软提出，然后他们在浏览器端用 `Knockoutjs` 实现。但这项技术的强大之处并未当时的开发者体会到，可能是因为 `Knockoutjs` 实现过于复杂，导致没有大面积的扩散。同样，这次接力的选手依然是 Google。Google 通过 Angularjs 将 MVVM 及单页应用发扬光大，让前端开发者能够开发出更加大型的应用，职能变得更大了随后都是就是前端圈开始得到了爆发式的发展，陆续出现了很多优秀的框架。

前端路由就是把不同路由对应不同的内容或页面的任务交给前端来做，之前是通过服务端根据 url 不同返回不同的页面来实现。

利用 H5 的 history.pushState 和 history.replaceState ，这两个 history 新增的 api，为前端操控浏览器历史栈提供了可能性

这两个 Api 都会操作浏览器的历史栈，而不会引起页面的刷新。

不同的是，pushState 会增加一条新的历史记录，而 replaceState 则会替换当前的历史记录。

优点和缺点：

优点： 用户体验好，不需要每次向服务器发送请求请求页面数据，响应快

缺点：使用浏览器的前进，后退键的时候会重新发送请求，没有合理地利用缓存，

## 前端路由解决方案

目前前端路由方案主要有以下几种

hash：可能是大多数人了解的模式，主要是基于锚点的原理实现。简单易用

history：即使用 html5 标准中的 history api 通过监听 popstate 事件来对 dom 进行操作。每次路由变化都会引起重定向

memory：这种实现是在内存中维护一个堆栈用于管理访问历史的方式，比较复杂。在早起移动端使用比较多。实现麻烦，问题也较多。现在很少有使用。RN 在使用这种路由模式

static：主要用于 ssr。需要后端去管理路由

前端路由解决的问题
根据路由变化显示不同的页面，完成页面切换
通过 query 传参
前端路由各种实现方案的对比
hash 路由 优缺点

优点：
实现简单，兼容性好（兼容到 ie8）
绝大多数前端框架均提供了给予 hash 的路由实现
不需要服务器端进行任何设置和开发
除了资源加载和 ajax 请求以外，不会发起其他请求
缺点：
对于部分需要重定向的操作，后端无法获取 hash 部分内容，导致后台无法取得 url 中的数据，典型的例子就是微信公众号的 oauth 验证
服务器端无法准确跟踪前端路由信息
对于需要锚点功能的需求会与目前路由机制冲突

**Hash 值的由来**

历史：

1、基于 Ajax 的 Web 应用最为明显的特征在于使用了浏览器内部原生支持的 XMLHttpRequest 对象与后台服务器进行数据通。

2、由于这种通信方式不需要页面的刷新动作，因而无论与后台发生了多少次通信，浏览器的 URL 会一直保持在初始地址不变。

3、这随之而来的一个问题便是不断变化的页面状态信息无法记录到浏览器的历史记录堆栈中，从而使得用户无法通过浏览器的前进 / 后退按钮在不同状态页面间进行切换。

解决思路

浏览器能够支持在用户访问过的页面间进行前进 / 后退的操作，依赖于内部维持的 history 对象。

出于安全性的考虑，浏览器并不允许 JavaScript 脚本对该对象进行增删改之类写操作，

而只是可以通过 history. back/forward() 等方法进行访问。既然在页面状态发生变化时，

无法通过脚本直接去影响浏览器的历史信息，那么只有通过 URL 的变化来触发浏览器增加一条新的历史记录。

这也就是说需要将 Ajax 应用的不同页面状态与 URL 进行一种一对

一的映射，并且能够在回退或前进到某一 URL 之时，
应用本身能够在页面无刷新的情况下跳转到正确的页面状态。如何对 Ajax 应用的初始 URL 进行改变，
而同时这种变化的切换又不会引起页面的重新加载呢？答案只有一个，那就是借助用于页面内资源片段定位目的
的“片段标识符”（fragment identifier），即 URL 中“#”符号后的字符串（hash string）。当浏览器向
服务器端请求资源时，片段标识符并不会连同 base URL 一同发往服务器端，而只是在得到服务器返回的结果
之后帮助浏览器快速定位到被相应的锚点（anchor）所标识的资源片段，即使无法找个对应的锚点，浏览器也并
不会报错。正是基于浏览器的这一特性，构建片段标识符与页面状态之间的映射关系成为了解决此类问题的基础。

History 模式路由优缺点
优点

对于重定向过程中不会丢失 url 中的参数。后端可以拿到这部分数据
绝大多数前段框架均提供了 history 模式的路由实现
后端可以准确跟踪路由信息
可以使用 history.state 来获取当前 url 对应的状态信息
缺点

兼容性不如 hash 路由(只兼容到 IE10)
需要后端支持，每次返回 html 文档

memory 路由 优缺点
优点

不存在兼容性问题，路由保存在内存中
不需要服务器端提供支持
缺点

目前很少有前端路由模块提供对 memory 路由的实现(react-router 提供了 memory 实现)
自己实现难度较大，且工作量也很大

对于前进后退操作的路由管理非常麻烦，尤其是 android 设备的 backbutton
static 路由 优缺点（该路由方式主要用于 ssr。不做比较。）
如何选择合适的前端路由方案？以下建议作为参考：

hash 模式适用场景：
兼容 IE8
没有重定向传参需求(第三方认证 oauth)
没有锚点跳跃需求
后端不需要跟踪前端路由信息
hybrid app 需要将前端资源打包在应用内，因为 html 的域在 file://下，所以不能发生重定向

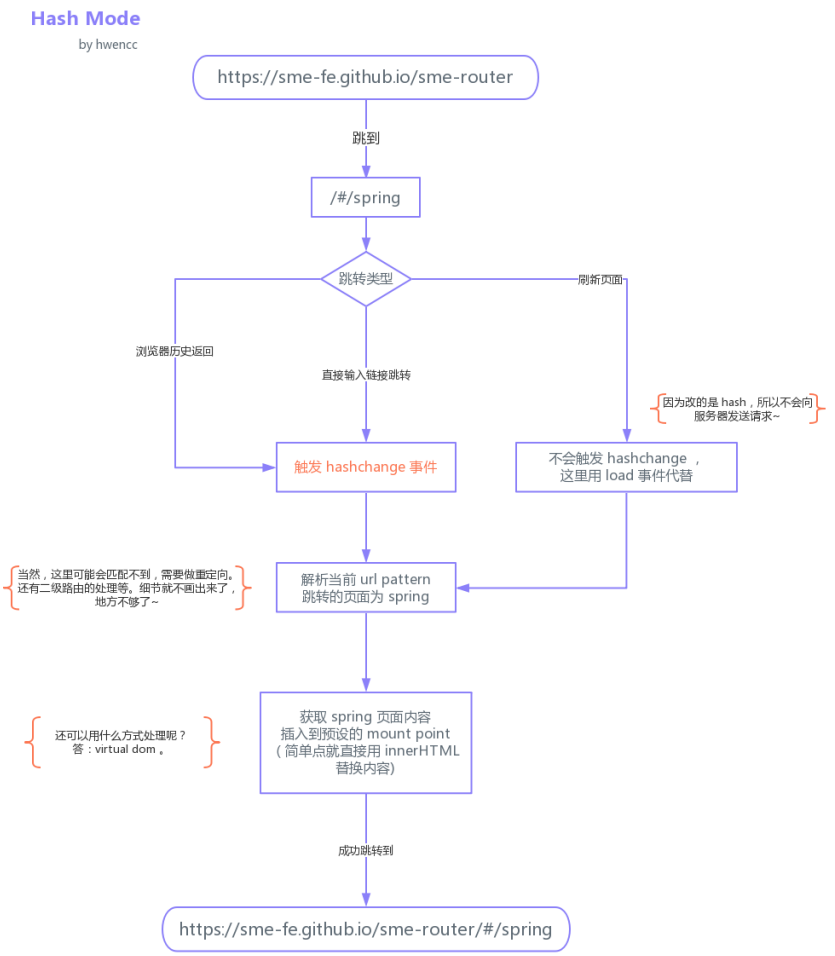
history 模式适用场景：
页面内锚点需求
需要重定向传参
同构直出
后端跟踪路由信息
附加路由信息(history.state)获取路由状态

memory 模式适用场景：
ie8 以下兼容
React Native

## 从 vue-router 来看前端路由实现原理

前端路由的实现其实很简单。
本质上就是检测 url 的变化，截获 url 地址，然后解析来匹配路由规则。
但是这样有人就会问：url 每次变化都会刷新页面啊？页面都刷新了，JavaScript 怎么检测和截获 url？
在 2014 年之前，大家是通过 hash 来实现路由，url hash 就是类似于 http://element.eleme.io/#/zh-CN/component/installation
这种 #。后面 hash 值的变化，并不会导致浏览器向服务器发出请求，浏览器不发出请求，也就不会刷新页面。另外每次 hash 值的变化，还会触发 hashchange 这个事件，通过这个事件我们就可以知道 hash 值发生了哪些变化。

让我们来整理思路，假如我们要用 hash 的模式实现一个路由，那么流程应该是这样的。



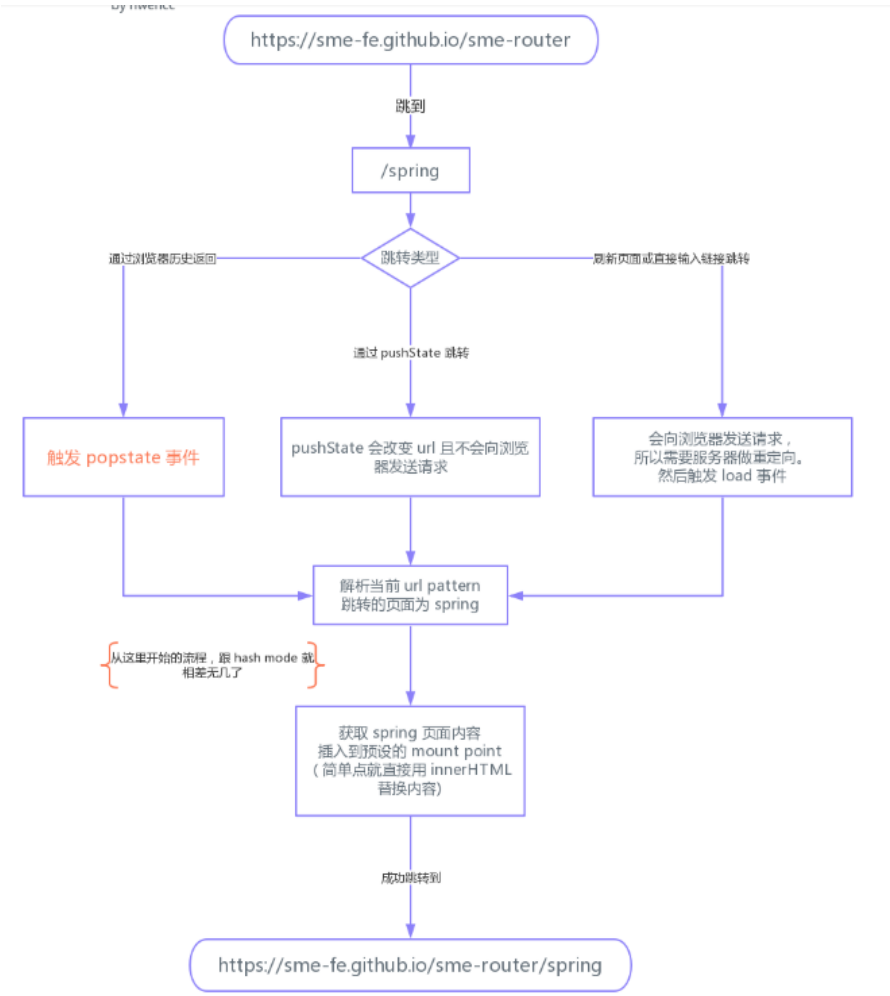

hash 的实现相对来说要简单方便些，而且不用服务器来支持。
另外我们可以参考参考 vue-router 这一部分的实现。

```
/**
 * 添加 url hash 变化的监听器
 */
```

```
setupListeners () {
  const router = this.router

  /**
   * 每当 hash 变化时就解析路径
   * 匹配路由
   */

  window.addEventListener('hashchange', () => {
    const current = this.current
    /**
     * transitionTo:
     * 匹配路由
     * 并通过路由配置，把新的页面 render 到 ui-view 的节点
     */

    this.transitionTo(getHash(), route => {

   replaceHash(route.fullPath)

    })

  })
}
```

检测到 hash 的变化后，就可以通过替换 DOM 的方式来实现页面的更换。

14 年后，因为 HTML5 标准发布。多了两个 API，pushState 和 replaceState，通过这两个 API 可以改变 url 地址且不会发送请求。同时还有 onpopstate 事件。通过这些就能用另一种方式来实现前端路由了，但原理都是跟 hash 实现相同的。用了 HTML5 的实现，单页路由的 url 就不会多出一个#，变得更加美观。但因为没有 # 号，所以当用户刷新页面之类的操作时，浏览器还是会给服务器发送请求。为了避免出现这种情况，所以这个实现需要服务器的支持，需要把所有路由都重定向到根页面。
同样，我们来理清下思路，这样写起代码才更得心应手




以下是 Vue-router 的部分实现源码，可以发现实现的思路大体也是相同的

```
export class HTML5History extends History {
  constructor (router, base) {
    super(router, base)
    /**
     * 原理还是跟 hash 实现一样
     * 通过监听 popstate 事件
     * 匹配路由，然后更新页面 DOM
     */
    window.addEventListener('popstate', e => {
      const current = this.current
```

```
        // Avoiding first `popstate` event dispatched in some browsers but first
        // history route not updated since async guard at the same time.
        const location = getLocation(this.base)
        if (this.current === START && location === initLocation) {
          return
        }

        this.transitionTo(location, route => {
          if (supportsScroll) {
            handleScroll(router, route, current, true)
          }
        })
      })
    }

    go (n) {
      window.history.go(n)
    }

    push (location, onComplete, onAbort) {
      const { current: fromRoute } = this
      this.transitionTo(location, route => {
        // 使用 pushState 更新 url，不会导致浏览器发送请求，从而不会刷新页面
        pushState(cleanPath(this.base + route.fullPath))
        onComplete && onComplete(route)
      }, onAbort)
    }

    replace (location, onComplete, onAbort) {
      const { current: fromRoute } = this
      this.transitionTo(location, route => {
        // replaceState 跟 pushState 的区别在于，不会记录到历史栈
        replaceState(cleanPath(this.base + route.fullPath))
        onComplete && onComplete(route)
      }, onAbort)
```

```
    }
}
```