

面试官: 你了解前端路由吗?

2018 年 04 月 09 日

面试官系列(3): 前端路由的实现

往期

- [面试官系列\(1\): 如何实现深克隆](#)
- [面试官系列\(2\): Event Bus的实现](#)

文章目录

1. 基于hash的前端路由实现
2. 基于hash的前端路由升级
3. 基于H5 History的前端路由实现

前言

前端路由是现代SPA应用必备的功能,每个现代前端框架都有对应的实现,例如vue-router、react-router。

我们不想探究vue-router或者react-router们的实现,因为不管是哪种路由无外乎用兼容性更好的hash实现或者是H5 History实现,与框架几个只需要做相应的封装即可。

提前声明: 我们没有对传入的参数进行及时判断而规避错误,也没有考虑兼容性问题,仅仅对核心方法进行了实现.

1.hash路由

hash路由一个明显的标志是带有#,我们主要是通过监听url中的hash变化来进行路由跳转。

hash的优势就是兼容性更好,在老版IE中都有运行,问题在于url中一直存在#不够美观,而且hash路由更像是Hack而非标准,相信随着发展更加标准化的History API会逐步蚕食掉hash路由的市场。



1.1 初始化class

我们用Class关键字初始化一个路由.

```
class Routers {
  constructor() {
    // 以键值对的形式储存路由
    this.routes = {};
    // 当前路由的URL
    this.currentUrl = '';
  }
}
```

1.2 实现路由hash储存与执行

在初始化完毕后我们需要思考两个问题:

1. 将路由的hash以及对应的callback函数储存
2. 触发路由hash变化后,执行对应的callback函数

```
class Routers {
  constructor() {
    this.routes = {};
    this.currentUrl = '';
  }
  // 将path路径与对应的callback函数储存
  route(path, callback) {
    this.routes[path] = callback || function() {};
  }
  // 刷新
  refresh() {
    // 获取当前URL中的hash路径
    this.currentUrl = location.hash.slice(1) || '/';
    // 执行当前hash路径的callback函数
    this.routes[this.currentUrl]();
  }
}
```

1.3 监听对应事件

那么我们只需要在实例化Class的时候监听上面的事件即可.

```
class Routers {
  constructor() {
    this.routes = {};
    this.currentUrl = '';
    this.refresh = this.refresh.bind(this);
    window.addEventListener('load', this.refresh, false);
    window.addEventListener('hashchange', this.refresh, false);
  }

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

  refresh() {
    this.currentUrl = location.hash.slice(1) || '/';
    this.routes[this.currentUrl]();
  }
}
```

对应效果如下:



完整示例

点击这里 [hash router](#) by 寻找海蓝 (@xiaomuzhu) on [CodePen](#).

2.增加回退功能

上一节我们只实现了简单的路由功能,没有我们常用的回退与前进功能,所以我们需要进行改造。

2.1 实现后退功能

我们在需要创建一个数组history来储存过往的hash路由例如/blue,并且创建一个指针currentIndex来随着后退和前进功能移动来指向不同的hash路由。

```
class Routers {
  constructor() {
    // 储存hash与callback键值对
    this.routes = {};
    // 当前hash
    this.currentUrl = '';
    // 记录出现过的hash
    this.history = [];
    // 作为指针,默认指向this.history的末尾,根据后退前进指向history中不同的hash
    this.currentIndex = this.history.length - 1;
    this.refresh = this.refresh.bind(this);
    this.backOff = this.backOff.bind(this);
    window.addEventListener('load', this.refresh, false);
    window.addEventListener('hashchange', this.refresh, false);
  }

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

  refresh() {
    this.currentUrl = location.hash.slice(1) || '/';
    // 将当前hash路由推入数组储存
    this.history.push(this.currentUrl);
    // 指针向前移动
    this.currentIndex++;
    this.routes[this.currentUrl]();
  }

  // 后退功能
  backOff() {
    // 如果指针小于0的话就不存在对应hash路由了,因此锁定指针为0即可
    this.currentIndex <= 0
      ? (this.currentIndex = 0)
      : (this.currentIndex = this.currentIndex - 1);
    // 随着后退,location.hash也应该随之变化
    location.hash = `#${this.history[this.currentIndex]}`;
    // 执行指针目前指向hash路由对应的callback
    this.routes[this.history[this.currentIndex]]();
  }
}
```

我们看起来实现的不错,可是出现了Bug,在后退的时候我们往往需要点击两下。

点击查看Bug示例 [hash router](#) by 寻找海蓝 (@xiaomuzhu) on [CodePen](#).

问题在于,我们每次在后退都会执行相应的callback,这会触发refresh()执行,因此每次我们后退,history中都会被push新的路由hash,currentIndex也会向前移动,这显然不是我们想要的。

```
refresh() {  
  this.currentUrl = location.hash.slice(1) || '/';  
  // 将当前hash路由推入数组储存  
  this.history.push(this.currentUrl);  
  // 指针向前移动  
  this.currentIndex++;  
  this.routes[this.currentUrl]();  
}
```

如图所示,我们每次点击后退,对应的指针位置和数组被打印出来



Log Entry	Index	History Array
hash.js:76	3	["/green", "/blue", "/", "/green", "/blue"]
hash.js:76	3	["/green", "/blue", "/", "/green", "/blue", "/green"]
hash.js:76	2	["/green", "/blue", "/", "/green", "/blue", "/green"]
hash.js:76	2	["/green", "/blue", "/", "/green", "/blue", "/green", "/"]
hash.js:76	1	["/green", "/blue", "/", "/green", "/blue", "/green", "/"]

2.2 完整实现hash Router

我们必须做一个判断,如果是后退的话,我们只需要执行回调函数,不需要添加数组和移动指针。

```
class Routers {  
  constructor() {  
    // 储存hash与callback键值对  
    this.routes = {};  
    // 当前hash  
    this.currentUrl = '';  
    // 记录出现过的hash
```

```

this.history = [];
// 作为指针,默认指向this.history的末尾,根据后退前进指向history中不同的hash
this.currentIndex = this.history.length - 1;
this.refresh = this.refresh.bind(this);
this.backOff = this.backOff.bind(this);
// 默认不是后退操作
this.isBack = false;
window.addEventListener('load', this.refresh, false);
window.addEventListener('hashchange', this.refresh, false);
}

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

refresh() {
  this.currentUrl = location.hash.slice(1) || '/';
  if (!this.isBack) {
    // 如果不是后退操作,且当前指针小于数组总长度,直接截取指针之前的部分储存下来
    // 此操作来避免当点击后退按钮之后,再进行正常跳转,指针会停留在原地,而数组添加新ha
    // 避免再次造成指针的不匹配,我们直接截取指针之前的数组
    // 此操作同时与浏览器自带后退功能的行为保持一致
    if (this.currentIndex < this.history.length - 1)
      this.history = this.history.slice(0, this.currentIndex + 1);
    this.history.push(this.currentUrl);
    this.currentIndex++;
  }
  this.routes[this.currentUrl]();
  console.log('指针:', this.currentIndex, 'history:', this.history);
  this.isBack = false;
}

// 后退功能
backOff() {
  // 后退操作设置为true
  this.isBack = true;
  this.currentIndex <= 0
    ? (this.currentIndex = 0)
    : (this.currentIndex = this.currentIndex - 1);
  location.hash = `#${this.history[this.currentIndex]}`;
  this.routes[this.history[this.currentIndex]]();
}
}

```

查看完整示例 [Hash Router](#) by 寻找海蓝 (@xiaomuzhu) on [CodePen](#).

前进的部分就不实现了,思路我们已经讲得比较清楚了,可以看出来,hash路由这种方式确实有点繁琐,所以HTML5标准提供了History API供我们使用。

3. HTML5新路由方案

3.1 History API

我们可以直接在浏览器中查询出History API的方法和属性。



当然,我们常用的方法其实是有限的,如果想全面了解可以去MDN查询[History API的资料](#)。

我们只简单看一下常用的API

```
window.history.back(); // 后退
window.history.forward(); // 前进
window.history.go(-3); // 后退三个页面
```

`history.pushState`用于在浏览历史中添加历史记录,但是并不触发跳转,此方法接受三个参数,依次为:

`state`:一个与指定网址相关的状态对象, `popstate`事件触发时,该对象会传入回调函数。如果不需要这个对象,此处可以填`null`。

`title`: 新页面的标题, 但是所有浏览器目前都忽略这个值, 因此这里可以填`null`。

`url`: 新的网址, 必须与当前页面处在同一个域。浏览器的地址栏将显示这个网址。

`history.replaceState`方法的参数与`pushState`方法一模一样, 区别是它修改浏览历史中当前纪录, 而非添加记录, 同样不触发跳转。

`popstate`事件, 每当同一个文档的浏览历史 (即`history`对象) 出现变化时, 就会触发`popstate`事件。

需要注意的是, 仅仅调用`pushState`方法或`replaceState`方法, 并不会触发该事件, 只有用户点击浏览器倒退按钮和前进按钮, 或者使用 JavaScript 调用`back`、`forward`、`go`方法时才会触发。

另外, 该事件只针对同一个文档, 如果浏览历史的切换, 导致加载不同的文档, 该事件也不会触发。

以上API介绍选自[history对象](#), 可以点击查看完整版, 我们不想占用过多篇幅来介绍API。

3.2 新标准下路由的实现

上一节我们介绍了新标准的History API, 相比于我们在Hash 路由实现的那些操作, 很显然新标准让我们的实现更加方便和可读。

所以一个mini路由实现起来其实很简单

```
class Routers {
  constructor() {
    this.routes = {};
    // 在初始化时监听popstate事件
    this._bindPopState();
  }
  // 初始化路由
  init(path) {
    history.replaceState({path: path}, null, path);
    this.routes[path] && this.routes[path]();
  }
  // 将路径和对应回调函数加入hashMap储存
```



```

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

// 触发路由对应回调
go(path) {
  history.pushState({path: path}, null, path);
  this.routes[path] && this.routes[path]();
}
// 监听popstate事件
_bindPopState() {
  window.addEventListener('popstate', e => {
    const path = e.state && e.state.path;
    this.routes[path] && this.routes[path]();
  });
}
}

```

点击查看H5路由 [H5 Router](#) by 寻找海蓝 (@xiaomuzhu) on [CodePen](#).

小结

我们大致探究了前端路由的两种实现方法,在没有兼容性要求的情况下显然符合标准的History API实现的路由是更好的选择。

想更深入了解前端路由实现可以阅读[vue-router代码](#), 除去开发模式代码、注释和类型检测代码, 核心代码并不多, 适合阅读。

下期预告

下期准备一篇关于双向绑定的话题,因为许多人只知道Object.defineProperty,禁不住深究:

1. 同是数据劫持,与Proxy相比有何优劣?
2. 除了数据劫持可以实现双向绑定还有没有其他方法?
3. 其他方法(例如脏检测、Observable模式、数据模型等)与数据劫持相比优劣如何?

由于涉及的框架和知识点过多,我开了一个头已经小2000字了,,在考虑要不要

分上下篇发出来,不过我相信它解决你对双向绑定所有的疑问。