# 历史

该**History**界面允许操纵浏览器会话历史记录，即在当前页面所加载的选项卡或框架中访问的页面。

参考：https://developer.mozilla.org/en-US/docs/Web/API/History

hash 路由：监听 url 中 hash 的变化，然后渲染不同的内容，这种路由不向服务器发送请求，不需要服务端的支持；

history 路由：监听 url 中的路径变化，需要客户端和服务端共同的支持；

1. **1. hash 路由**
2. 当页面中的 hash 发生变化时，会触发hashchange事件，因此我们可以监听这个事件，来判断路由是否发生了变化。
3. window.addEventListener(
4. 'hashchange',
5. **function** (event) {
6. **const** oldURL **=** event.oldURL; *// 上一个URL*
7. **const** newURL **=** event.newURL; *// 当前的URL*
8. console.log(newURL, oldURL);
9. },
10. **false**
11. );
12. **1.1 实现的过程**
13. 对 oldURL 和 newURL 进行拆分后，就能获取到更详细的 hash 值。我们这里从创建一个 HashRouter 的 class 开始一步步写起：
14. **class** HashRouter {
15. currentUrl **=** ''; *// 当前的URL*
16. handlers **=** {};
17. getHashPath(url) {
18. **const** index **=** url.indexOf('#');
19. **if** (index **>=** 0) {
20. **return** url.slice(index **+** 1);
21. }
22. **return** '/';
23. }
24. }
25. 事件hashchange只会在 hash 发生变化时才能触发，而第一次进入到页面时并不会触发这个事件，因此我们还需要监听load事件。这里要注意的是，两个事件的 event 是不一样的：hashchange 事件中的 event 对象有 oldURL 和 newURL 两个属性，但 load 事件中的 event 没有这两个属性，不过我们可以通过 location.hash 来获取到当前的 hash 路由：
26. **class** HashRouter {
27. currentUrl **=** ''; *// 当前的URL*
28. handlers **=** {};
29. constructor() {
30. **this**.refresh **=** **this**.refresh.bind(**this**);
31. window.addEventListener('load', **this**.refresh, **false**);
32. window.addEventListener('hashchange', **this**.refresh, **false**);
33. }
34. getHashPath(url) {
35. **const** index **=** url.indexOf('#');
36. **if** (index **>=** 0) {
37. **return** url.slice(index **+** 1);
38. }
39. **return** '/';
40. }
41. refresh(event) {
42. **let** curURL **=** '',
43. oldURL **=** **null**;
44. **if** (event.newURL) {
45. oldURL **=** **this**.getHashPath(event.oldURL **||** '');
46. curURL **=** **this**.getHashPath(event.newURL **||** '');
47. } **else** {
48. curURL **=** **this**.getHashPath(window.location.hash);
49. }
50. **this**.currentUrl **=** curURL;
51. }
52. }
53. 到这里已经可以实现获取当前的 hash 路由，但路由发生变化时，我们的页面应该进行切换，因此我们需要监听这个变化：
54. **class** HashRouter {
55. currentUrl **=** ''; *// 当前的URL*
56. handlers **=** {};
57. *// 暂时省略上面的代码*
58. refresh(event) {
59. *// 当hash路由发生变化时，则触发change事件*
60. **this**.emit('change', curURL, oldURL);
61. }
62. on(evName, listener) {
63. **this**.handlers[evName] **=** listener;
64. }
65. emit(evName, ...args) {
66. **const** handler **=** **this**.handlers[evName];
67. **if** (handler) {
68. handler(...args);
69. }
70. }
71. }
72. **const** router **=** **new** HashRouter();
73. rouer.on('change', (curUrl, lastUrl) => {
74. console.log('当前的hash:', curUrl);
75. console.log('上一个hash:', lastUrl);
76. });
77. **1.2 调用的方式**
78. 到这里，我们把基本的功能已经完成了。来配合一个例子就更形象了：
79. *// 先定义几个路由*
80. **const** routes **=** [
81. {
82. path**:** '/',
83. name**:** 'home',
84. component**:** **<**Home **/>**,
85. },
86. {
87. path**:** '/about',
88. name**:** 'about',
89. component**:** **<**About **/>**,
90. },
91. {
92. path**:** '\*',
93. name**:** '404',
94. component**:** **<**NotFound404 **/>**,
95. },
96. ];
97. **const** router **=** **new** HashRouter();
98. *// 监听change事件*
99. router.on('change', (currentUrl, lastUrl) => {
100. **let** route **=** **null**;
101. *// 匹配路由*
102. **for** (**let** i **=** 0, len **=** routes.length; i **<** len; i**++**) {
103. **const** item **=** routes[i];
104. **if** (currentUrl **===** item.path) {
105. route **=** item;
106. **break**;
107. }
108. }
109. *// 若没有匹配到，则使用最后一个路由*
110. **if** (**!**route) {
111. route **=** routes[routes.length **-** 1];
112. }
113. *// 渲染当前的组件*
114. ReactDOM.render(route.component, document.getElementById('app'));
115. });

### 2. history 路由

在 history 路由中，我们一定会使用window.history中的方法，常见的操作有：

* back()：后退到上一个路由；
* forward()：前进到下一个路由，如果有的话；
* go(number)：进入到任意一个路由，正数为前进，负数为后退；
* pushState(obj, title, url)：前进到指定的 URL，不刷新页面；
* replaceState(obj, title, url)：用 url 替换当前的路由，不刷新页面；

调用这几种方式时，都会只是修改了当前页面的 URL，页面的内容没有任何的变化。但前 3 个方法只是路由历史记录的前进或者后退，无法跳转到指定的 URL；而pushState和replaceState可以跳转到指定的 URL。如果有面试官问起这个问题“如何仅修改页面的 URL，而不发送请求”，那么答案就是这 5 种方法。

如果服务端没有新更新的 url 时，一刷新浏览器就会报错，因为刷新浏览器后，是真实地向服务器发送了一个 http 的网页请求。因此若要使用 history 路由，需要服务端的支持。

### 2.1 应用的场景

pushState 和 replaceState 两个方法跟 location.href 和 location.replace 两个方法有什么区别呢？应用的场景有哪些呢？

1. location.href 和 location.replace 切换时要向服务器发送请求，而 pushState 和 replace 仅修改 url，除非主动发起请求；
2. 仅切换 url 而不发送请求的特性，可以在前端渲染中使用，例如首页是服务端渲染，二级页面采用前端渲染；
3. 可以添加路由切换的动画；
4. 在浏览器中使用类似抖音的这种场景时，用户滑动切换视频时，可以静默修改对应的 URL，当用户刷新页面时，还能停留在当前视频。

### 2.2 无法监听路由的变化

当我们用 history 的路由时，必然要能监听到路由的变化才行。全局有个popstate事件，别看这个事件名称中有个 state 关键词，但pushState和replaceState被调用时，是不会触发触发 popstate 事件的，只有上面列举的前 3 个方法会触发。可以点击【[popState 不会触发 popstate 事件](https://link.zhihu.com/?target=https%3A//www.xiabingbao.com/demos/20200410/history-pushstate-popstate.html)】查看。

针对这种情况，我们可以使用window.dispatchEvent添加事件：

**const** listener **=** **function** (type) {

**var** orig **=** history[type];

**return** **function** () {

**var** rv **=** orig.apply(**this**, arguments);

**var** e **=** **new** Event(type);

e.arguments **=** arguments;

window.dispatchEvent(e);

**return** rv;

};

};

window.history.pushState **=** listener('pushState');

window.history.replaceState **=** listener('replaceState');

然后就可以添加对这两个方法的监听了：

window.addEventListener('pushState', **this**.refresh, **false**);

window.addEventListener('replaceState', **this**.refresh, **false**);

### 2.3 完整的代码

完整的代码如下：

**class** HistoryRouter {

currentUrl **=** '';

handlers **=** {};

constructor() {

**this**.refresh **=** **this**.refresh.bind(**this**);

**this**.addStateListener();

window.addEventListener('load', **this**.refresh, **false**);

window.addEventListener('popstate', **this**.refresh, **false**);

window.addEventListener('pushState', **this**.refresh, **false**);

window.addEventListener('replaceState', **this**.refresh, **false**);

}

addStateListener() {

**const** listener **=** **function** (type) {

**var** orig **=** history[type];

**return** **function** () {

**var** rv **=** orig.apply(**this**, arguments);

**var** e **=** **new** Event(type);

e.arguments **=** arguments;

window.dispatchEvent(e);

**return** rv;

};

};

window.history.pushState **=** listener('pushState');

window.history.replaceState **=** listener('replaceState');

}

refresh(event) {

**this**.currentUrl **=** location.pathname;

**this**.emit('change', location.pathname);

document.querySelector('#app span').innerHTML **=** location.pathname;

}

on(evName, listener) {

**this**.handlers[evName] **=** listener;

}

emit(evName, ...args) {

**const** handler **=** **this**.handlers[evName];

**if** (handler) {

handler(...args);

}

}

}

**const** router **=** **new** HistoryRouter();

router.on('change', **function** (curUrl) {

console.log(curUrl);

});