30 天精通 RxJS (19)：實務範例 - 簡易 Auto Complete 實作

2017年1月1日。6 分钟阅读

> 今天我们要做一个RxJS 的经典范例- 自动完成(Auto Complete)，自动完成在实务上的应用非常广泛，几乎随处可见这样的功能，只要是跟表单、搜寻相关的都会看到。虽然是个很常见的功能，但多数的工程师都只是直接套套件来完成，很少有人会自己从头到尾把完整的逻辑写一次。

如果有自己实作过Auto Complete 功能的工程师，应该就会知道这个功能在实作的过程中很多细节会让程式码变的非常复杂，像是要如何取消上一次发送出去的request、要如何优化请求次数...等等，这些小细节都会让程式码变的非常复杂且很难维护。

就让我们一起来用RxJS 来实作这个功能吧！

需求分析

首先我们会有一个搜寻框(input#search)，当我们在上面打字并停顿超过100 毫秒就发送HTTP Request 来取得建议选项并显示在收寻框下方(ul#suggest-list)，如果使用者在前一次发送的请求还没有回来就打了下一个字，此时前一个发送的请求就要舍弃掉，当建议选项显示之后可以用滑鼠点击取建议选项代搜寻框的文字。

上面的叙述可以拆分成以下几个步骤

- 准备input#search 以及ul#suggest-list 的HTML 与CSS
- 在input#search 输入文字时，等待100 毫秒再无输入，就发送HTTP Request
- 当Response 还没回来时，使用者又输入了下一个文字就舍弃前一次的并再发送一次新的Request
- 接受到Response 之后显示建议选项
- 滑鼠点击后取代input#search 的文字

基本的HTML 跟CSS 笔者已经帮大家完成，大家可以直接到下面的连结接着实作：

- JSBin

先让我们看一下HTML，首先在HTML 里有一个input(#search)，这个input(#search) 就是要用来输入的栏位，它下方有一个ul(#suggest-list)，则是放建议选项的地方

CSS的部分可以不用看，JS的部分已经写好了要发送API的url跟方法 `getSuggestList` ，接着就开始实作自动完成的效果吧！

```
const searchInput = document.getElementById('search');
const suggestList = document.getElementById('suggest-list');
```

第二步，建立所需的 Observable

这里我们要监听收寻栏位的input 事件，以及建议选项的点击事件

```
const keyword = Rx.Observable.fromEvent(searchInput, 'input');
const selectItem = Rx.Observable.fromEvent(suggestList, 'click');
```

第三步，撰写程式逻辑

每当使用者输入文字就要发送HTTP request，并且有新的值被输入后就舍弃前一次发送的，所以这里用switchMap

```
keyword.switchMap(e => getSuggestList(e.target.value))
```

这里我们先试着订阅，看一下API 会回传什么样的资料

```
keyword
    .switchMap(e => getSuggestList(e.target.value))
    .subscribe(console.log)
```

在search 栏位乱打几个字

▼Array[4]
0: "aaa"
▼1: Array[5]
0: "AAA (團體)"
1: "AAA 10th ANNIVERSARY Documentary ～Road of 10th ANNIVERSARY～"
2: "AAA"
3: "AAA (电子游戏产业)"
4: "AAA 2nd Anniversary Live -5th ATTACK 070922- 日本武道館"
length: 5
▶__proto__: Array[0]
▼2: Array[5]
0: "AAA（日語：トリプル・エー，英語：Triple A，簡稱「トリエー」），取自英文「Attack All Around」的頭文字母，意指「全方位出擊」，是日本原五男三女8人組、現五男二女7人組表
1: "『AAA 10th ANNIVERSARY Documentary ～Road of 10th ANNIVERSARY～』是日本團體AAA於2016年3月30日發售的第18個影像作品。"
2: "AAA是一個三個字母的縮寫，可能指以下其中一個意思："
3: "AAA（讀作「3A」）在電子遊戲產業中是指，高開發預算、高宣傳投入、並獲得專業評論一致高度評價的遊戲。"
4: "『2nd Anniversary Live -5th ATTACK 070922- 日本武道館』為、日本音樂團體AAA在2008年1月9日發售的第五個影像作品。首次以DVD跟HD DVD兩種型態發售。"
length: 5
▶__proto__: Array[0]
▼3: Array[5]
0: "https://zh.wikipedia.org/wiki/AAA_(%E5%9C%98%E9%AB%94)"
1: "https://zh.wikipedia.org/wiki/AAA_10th_ANNIVERSARY_Documentary_%EF%BD%9ERoad_of_10th_ANNIVERSARY%EF%BD%9E"
2: "https://zh.wikipedia.org/wiki/AAA"
3: "https://zh.wikipedia.org/wiki/AAA_(%E7%94%B5%E5%AD%90%E6%B8%B8%E6%88%8F%E4%BA%A7%E4%B8%9A)"
4: "https://zh.wikipedia.org/wiki/AAA_2nd_Anniversary_Live_-5th_ATTACK_070922-_%E6%97%A5%E6%9C%AC%E6%AD%A6%E9%81%93%E9%A4%A8"

```
keyword
    .switchMap(
        e => getSuggestList(e.target.value),
        (e, res) => res[1]
    )
    .subscribe(console.log)
```

这时再输入文字就可以看到确实是我们要的返回值

```
runner-3.40.3.min.js:1
▶ ["AAA (團體)", "AAA 10th ANNIVERSARY Documentary ～Road of 10th ANNIVERSARY～", "AAA", "AAA (电子游戏产业)", "AAA 2nd Anniversary Live -5th ATTACK 070922- 日本武道館"]
```

写一个render 方法，把阵列转成li 并写入suggestList

```
const render = (suggestArr = []) => {
    suggestList.innerHTML = suggestArr
                            .map(item => '<li>'+ item +'</li>')
                            .join('');
}
```

这时我们就可用render 方法把取得的阵列传入

```
const render = (suggestArr = []) => {
    suggestList.innerHTML = suggestArr
                            .map(item => '<li>'+ item +'</li>')
                            .join('');
}

keyword
  .switchMap(
    e => getSuggestList(e.target.value),
    (e, res) => res[1]
  )
  .subscribe(list => render(list))
```

如此一来我们打字就能看到结果出现在input 下方了





只是目前还不能点选，先让我们来做点选的功能，这里点选的功能我们需要用到delegation event 的小技巧，利用ul 的click 事件，来塞选是否点到了li，如下

```
selectItem
  .filter(e => e.target.matches('li'))
```

上面我们利用DOM 物件的matches 方法(里面的字串放css 的selector)来过滤出有点击到li 的事件，再用map 转出我们要的值并写入input。

```
selectItem
  .filter(e => e.target.matches('li'))
  .map(e => e.target.innerText)
  .subscribe(text => searchInput.value = text)
```

现在我们就能点击建议清单了，但是点击后清单没有消失，这里我们要在点击后重新redner，所以把上面的程式码改一下

```
selectItem
```

```
      render();
  })
```

这样一来我们就完成最基本的功能了，大家可以到 这里 看初步的完成品。

还记得我们前面说每次打完字要等待100毫秒在发送request吗？这样能避免过多的request发送，可以降低server的负载也会有比较好的使用者体验，要做到这件事很简单只要加上 `debounceTime(100)` 就完成了

```
keyword
  .debounceTime(100)
  .switchMap(
    e => getSuggestList(e.target.value),
    (e, res) => res[1]
  )
  .subscribe(list => render(list))
```

当然这个数值可以依照需求或是请UX 针对这个细节作调整。

这样我们就完成所有功能了，大家可以到 这里 查看结果。

今日小结

我们用了不到30 行的程式码就完成了auto complete 的基本功能，当我们能够自己从头到尾的完成这样的功能，在面对各种不同的需求，我们就能很方便的针对需求作调整，而不会受到套件的牵制！比如说我们希望使用者打了2 个字以上在发送request，这时我们只要加上一行filter 就可以了

```
keyword
  .filter(e => e.target.value.length > 2)
  .debounceTime(100)
  .switchMap(
    e => getSuggestList(e.target.value),
    (e, res) => res[1]
  )
  .subscribe(list => render(list))
```

又或者网站的使用量很大，可能API在最大的时候会回传失败，主管希望可以在API失败的时候重

```
  .debounceTime(100)
  .switchMap(
    e => Rx.Observable.from(getSuggestList(e.target.value))
                      .retry(3),
    (e, res) => res[1]
  )
  .subscribe(list => render(list))
```

大家会发现我们的灵活度变的非常高，又同时兼顾了程式码的可读性，短短的几行程式码就完成了一个复杂的需求，这就是RxJS 的魅力啊～

标签

JavaScript RxJS 可观察的 运算符 RxJS 30天

上一页 下一个