那些年我们错过的响应式编程

阅读目录

- 原文链接 ： The introduction to Reactive Programming you've been missing
- 作者 ： @andrestaltz
- 译者 ： yaoqinwei
- 校对者: bboyfeiyu、chaossss
- 状态 ： 完成

相信你们在学习响应式编程这个新技术的时候都会充满了好奇，特别是 的一些变体，例如：Rx系列、Bacon.js、RAC等等……

在缺乏优秀资料的前提下，响应式编程的学习过程将满是荆棘。起初，试图寻找一些教程，却只找到少量的实践指南，而且它们讲的都非常浅显，从来没人接受响应式编程建立一个完整知识体系的挑战。此外，官方文档通常也不能很好地帮助你理解函数，因为它们通常看起来很绕，不信请看这里：

> Rx.Observable.prototype.flatMapLatest(selector, [thisArg])
>
> 根据元素下标，将可观察序列中每个元素一一映射到一个新的可观察序列当中，然后...%……%&￥#@@……&**(晕了)

天呐，这简直太绕了！

我读过两本相关的书，一本只是在给你描绘响应式编程的伟大景象，而另一本却只是深入到如何使用响应式库而已。我在不断的构建项目过程中把响应式编程了解的透彻了一些，最后以这种艰难的方式学完了响应式编程。在我工作公司的一个实际项目中我会用到它，当我遇到问题时，还可以得到同事的支持。

学习过程中最难的部分是如何以响应式的方式来思考，更多的意味着要摒弃那些老旧的命令式和状态式的典型编程习惯，并且强迫自己的大脑以不同的范式来运作。我还没有在网络上找到任何一个教程是从这个层面来剖析的，我觉得这个世界非常值得拥有一个优秀的实践教程来教你如何以响应式编程的方式来思考，方便引导你开始学习响应式编程。然后看各种库文档才可以给你更多的指引。希望这篇文章能够帮助你快速地进入响应式编程的世界。

"什么是响应式编程?"

网络上有一大堆糟糕的解释和定义，如Wikipedia上通常都是些非常笼统和理论性的解释，而Stackoverflow上的一些规范的回答显然也不适合新手来参考，Reactive Manifesto看起来也只像是拿给你的PM或者老板看的东西，微软的Rx术语"Rx = Observables + LINQ + Schedulers" 也显得太过沉重，而且充满了太多微软式的东西，反而给我们带来更多疑惑。相对于你使用的MV*框架以及你钟爱的编程语言，"Reactive" 和"Propagation of change"这样的术语并没有传达任何有意义的概念。当然，我的view框架能够从model做出反应，我的改变当然也会传播，如果没有这些，我的界面根本就没有东西可渲染。

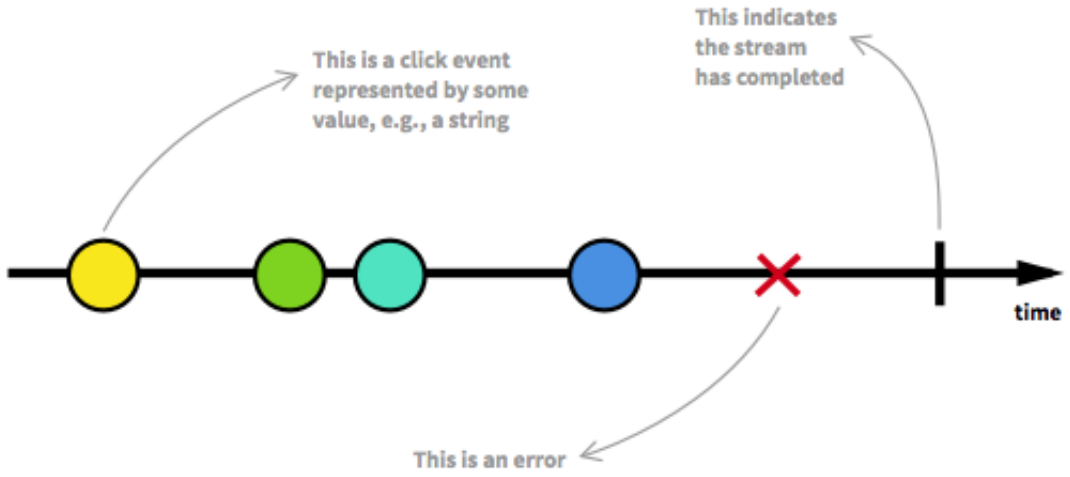
所以，不要再扯这些废话了。

响应式编程就是与异步数据流交互的编程范式

一方面，这已经不是什么新事物了。事件总线(Event Buses)或一些典型的点击事件本质上就是一个异步事件流(asynchronous event stream)，这样你就可以观察它的变化并使其做出一些反应(do some side effects)。响应式是这样的一个思路：除了点击和悬停(hover)的事件外，你可以给任何事物创建数据流。数据流无处不在，任何东西都可以成为一个数据流，例如变量、用户输入、属性、缓存、数据结构等等。举个栗子，你可以把你的微博订阅功能想象成跟点击事件一样的数据流，你可以监听这样的数据流，并做出相应的反应。

最重要的是，你会拥有一些令人惊艳的函数去结合、创建和过滤任何一组数据流。 这就是"函数式编程"的魔力所在。一个数据流可以作为另一个数据流的输入，甚至多个数据流也可以作为另一个数据流的输入。你可以合并两个数据流，也可以过滤一个数据流得到另一个只包含你感兴趣的事件的数据流，还可以映射一个数据流的值到一个新的数据流里。

数据流是整个响应式编程体系中的核心，要想学习响应式编程，当然要先走进数据流一探究竟了。那现在就让我们先从熟悉的"点击一个按钮"的事件流开始



一个数据流是一个按时间排序的即将发生的事件(Ongoing events ordered in time)的序列。如上图，它可以发出3种不同的事件(上一句已经把它们叫做事件)：一个某种类型的值事件，一个错误事件和一个完成事件。当一个完成事件发生时，在某些情况下，我们可能会做这样的操作：关闭包含那个按钮的窗口或者视图组件。

我们只能异步捕捉被发出的事件，使得我们可以在发出一个值事件时执行一个函数，发出错误事件时执行一个函数，发出完成事件时执行另一个函数。有时候你可以忽略后两个事件，只需聚焦于如何定义和设计在发出值事件时要执行的函数，监听这个事件流的过程叫做订阅，我们定义的函数叫做观察者，而事件流就可以叫做被观察的主题(或者叫被观察者)。你应该察觉到了，对的，它就是观察者模式。

上面的示意图我们也可以用ASCII码的形式重新画一遍，请注意，下面的部分教程中我们会继续使用这幅图：

```
--a---b-c---d---X---|->

a, b, c, d 是值事件
X 是错误事件
| 是完成事件
---> 是时间线(轴)
```

现在你对响应式编程事件流应该非常熟悉了，为了不让你感到无聊，让我们来做一些新的尝试吧：我们将创建一个由原始点击事件流演变而来的一种新的点击事件流。

首先，让我们来创建一个记录按钮点击次数的事件流。在常用的响应式库中，每个事件流都会附有一些函数，例如map,filter,scan等，当你调用这其中的一个方法时，比如clickStream.map(f)，它会返回基于点击事件流的一个新事件流。它不会对原来的点击事件流做任何的修改。这种特性叫做不可变性(immutability)，而且它可以和响应式事件流搭配在一起使用，就像豆浆和油条一样完美的搭配。这样我们可以用链式函数的方式来调用，例如：clickStream.map(f).scan(g):

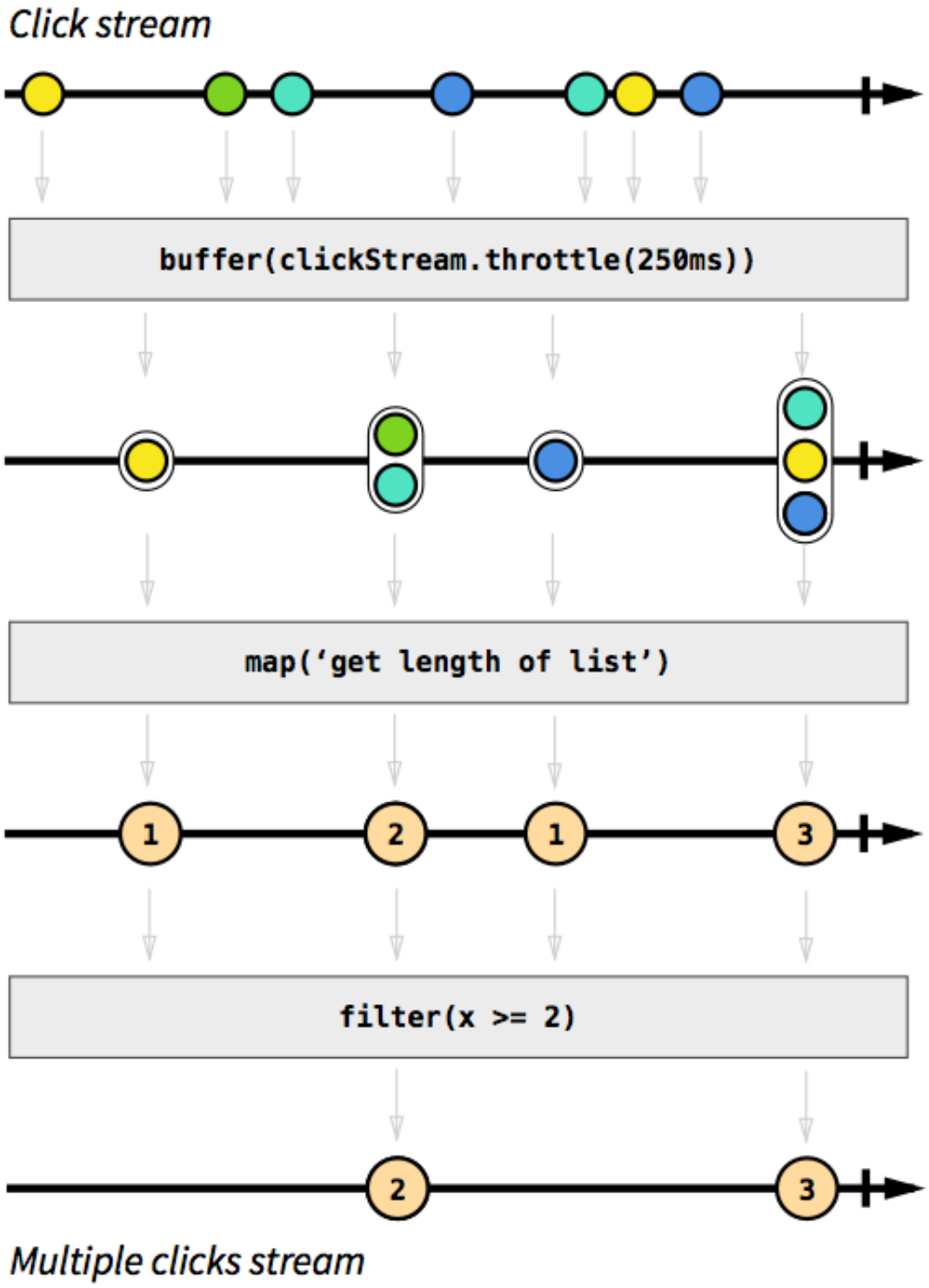
```
clickStream: ---c----c--c----c------c-->
               vvvvv map(c becomes 1) vvvv
               ---1----1--1----1------1-->
               vvvvvvvvv scan(+) vvvvvvvvv
counterStream: ---1----2--3----4------5-->
```

map(f)函数会根据你提供的f函数把原事件流中每一个返回值分别映射到新的事件流中。在上图的例子中，我们把每一次点击事件都映射成数字1，scan(g)函数则把之前映射的值聚集起来，然后根据x = g(accumulated, current)算法来作相应的处理，而本例的g函数其实就是简

单的加法函数。然后，当一个点击事件发生时，counterStream函数则上报当前点击事件总数。

为了展示响应式编程真正的魅力，我们假设你有一个"双击"事件流，为了让它更有趣，我们假设这个事件流同时处理"三次点击"或者"多次点击"事件，然后深吸一口气想想如何用传统的命令式和状态式的方式来处理，我敢打赌，这么做会相当的讨厌，其中还要涉及到一些变量来保存状态，并且还得做一些时间间隔的调整。

而用响应式编程的方式处理会非常的简洁，实际上，逻辑处理部分只需要四行代码。但是，当前阶段让我们现忽略代码的部分，无论你是新手还是专家，看着图表思考来理解和建立事件流将是一个非常棒的方法。



图中，灰色盒子表示将上面的事件流转换下面的事件流的函数过程，首先根据250毫秒的间隔

时间(event silence, 译者注：无事件发生的时间段，上一个事件发生到下一个事件发生的间隔时间)把点击事件流一段一隔开，再将每一段的一个或多个点击事件添加到列表中(这就是这个函数：buffer(stream.throttle(250ms))所做的事情，当前我们先不要急着去理解细节，我们只需专注响应式的部分先)。现在我们得到的是多个含有事件流的列表，然后我们使用了map()中的函数来算出每一个列表长度的整数数值映射到下一个事件流当中。最后我们使用了过滤filter(x >= 2)函数忽略掉了小于1的整数。就这样，我们用了3步操作生成了我们想要的事件流，接下来，我们就可以订阅("监听")这个事件并作出我们想要的操作了。

我希望你能感受到这个示例的优雅之处。当然了，这个示例也只是响应式编程魔力的冰山一角而已，你同样可以将这3步操作应用到不同种类的事件流中去，例如，一串API响应的事件流。另一方面，你还有非常多的函数可以使用。

"我为什么要采用响应式编程？"

响应式编程可以加深你代码抽象的程度，让你可以更专注于定义与事件相互依赖的业务逻辑，而不是把大量精力放在实现细节上，同时，使用响应式编程还能让你的代码变得更加简洁。

特别对于现在流行的webapps和mobile apps，它们的 UI 事件与数据频繁地产生交互，在开发这些应用时使用响应式编程的优点将更加明显。十年前，web页面的交互是通过提交一个很长的表单数据到后端，然后再做一些简单的前端渲染操作。而现在的Apps则演变的更具有实时性：仅仅修改一个单独的表单域就能自动的触发保存到后端的代码，就像某个用户对一些内容点了赞，就能够实时反映到其他已连接的用户一样，等等。

当今的Apps都含有丰富的实时事件来保证一个高效的用户体验，我们就需要采用一个合适的工具来处理，那么响应式编程就正好是我们想要的答案。

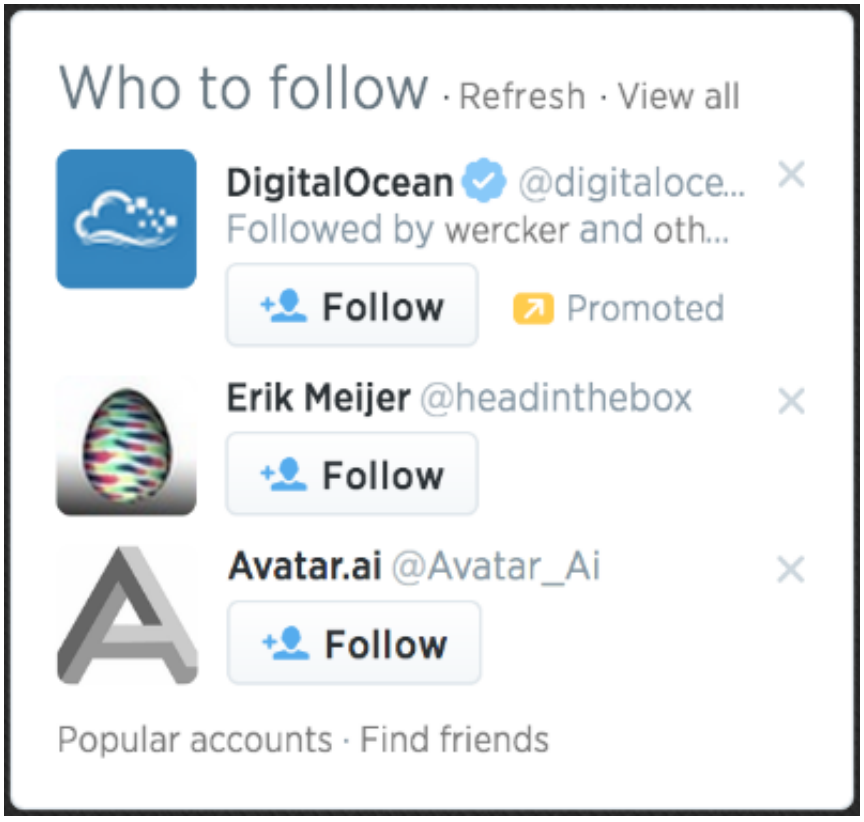
以响应式编程方式思考的例子

让我们深入到一些真实的例子，一个能够一步一步教你如何以响应式编程的方式思考的例子，没有虚构的示例，没有一知半解的概念。在这个教程的末尾我们将产生一些真实的函数代码，并能够知晓每一步为什么那样做的原因(知其然，知其所以然)。

我选了JavaScript和RxJS来作为本教程的编程语言，原因是：JavaScript是目前最多人熟悉的语言，而Rx系列的库对于很多语言和平台的运用是非常广泛的，例如(.NET, Java, Scala, Clojure, JavaScript, Ruby, Python, C++, Objective-C/Cocoa, Groovy等等。所以，无论你用的是什么语言、库、工具，你都能从下面这个教程中学到东西(从中受益)。

实现一个推荐关注(Who to follow)的功能

在Twitter里有一个UI元素向你推荐你可以关注的用户，如下图：



我们将聚焦于模仿它的主要功能，它们是：

- 开始阶段，从API加载推荐关注的用户账户数据，然后显示三个推荐用户
- 点击刷新，加载另外三个推荐用户到当前的三行中显示
- 点击每一行的推荐用户上的'x'按钮，清楚当前被点击的用户，并显示新的一个用户到当前行
- 每一行显示一个用户的头像并且在点击之后可以链接到他们的主页。

我们可以先不管其他的功能和按钮，因为它们是次要的。因为Twitter最近关闭了未经授权的公共API调用，我们将用Github获取用户的API代替，并且以此来构建我们的UI。

如果你想先看一下最终效果，这里有完成后的代码。

Request和Response

在Rx中是怎么处理这个问题呢？，在开始之前，我们要明白，(几乎)一切都可以成为一个事件流，这就是Rx的准则(mantra)。让我们从最简单的功能开始："开始阶段，从API加载推荐关注的用户账户数据，然后显示三个推荐用户"。其实这个功能没什么特殊的，简单的步骤分为：(1)发出一个请求，(2)获取响应数据，(3)渲染响应数据。ok，让我们把请求作为一个事件流，一开始你可能会觉得这样做有些夸张，但别急，我们也得从最基本的开始，不是吗？

开始时我们只需做一次请求，如果我们把它作为一个数据流的话，它只能成为一个仅仅返回一个值的事件流而已。一会儿我们还会有很多请求要做，但当前，只有一个。

```
1 --a------|->
2
3 a就是字符串：'https://api.github.com/users'
```

这是一个我们要请求的URL事件流。每当发生一个请求时，它将告诉我们两件事：什么时候和做了什么事(when and what)。什么时候请求被执行，什么时候事件就被发出。而做了什么就是请求了什么，也就是请求的URL字符串。

在Rx中，创建返回一个值的事件流是非常简单的。其实事件流在Rx里的术语是叫"被观察者"，也就是说它是可以被观察的，但是我发现这名字比较傻，所以我更喜欢把它叫做事件流。

```
1 var requestStream = Rx.Observable.just('https://api.github.com/use
```

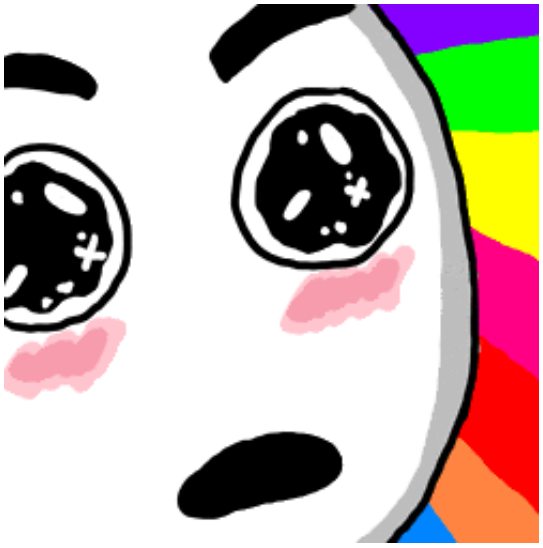
但现在，这只是一个字符串的事件流而已，并没有做其他操作，所以我们需要在发出这个值的时候做一些我们要做的操作，可以通过订阅(subscribing)这个事件来实现。

```
1 requestStream.subscribe(function(requestUrl) { // execute the requ
2 }
```

注意到我们这里使用的是JQuery的AJAX回调方法(我们假设你已经很了解JQuery和AJAX了)来的处理这个异步的请求操作。但是，请稍等一下，Rx就是用来处理异步数据流的，难道它就不能处理来自请求(request)在未来某个时间响应(response)的数据流吗？好吧，理论上是可以的，让我们尝试一下。

```
1 requestStream.subscribe(function(requestUrl) { // execute the requ
2     jQuery.getJSON(requestUrl)
3     .done(function(response) { observer.onNext(response); })
4     .fail(function(jqXHR, status, error) { observer.onError(error);
5     .always(function() { observer.onCompleted(); });
6   });
7
8   responseStream.subscribe(function(response) { // do something wit
9 }
```

Rx.Observable.create()操作就是在创建自己定制的事件流，且对于数据事件(onNext())和错误事件(onError())都会显示的通知该事件每一个观察者(或订阅者)。我们做的只是小小的封装一下jQuery Ajax Promise而已。等等，这是否意味者jQuery Ajax Promise本质上就是一个被观察者呢(Observable)？



是的。

Promise++就是被观察者(Observable)，在Rx里你可以使用这样的操作：var stream = Rx.Observable.fromPromise(promise)，就可以很轻松的将Promise转换成一个被观察者(Observable)，非常简单的操作就能让我们现在就开始使用它。不同的是，这些被观察者都不能兼容Promises/A+，但理论上并不冲突。一个Promise就是一个只有一个返回值的简单的被观察者，而Rx就远超于Promise，它允许多个值返回。

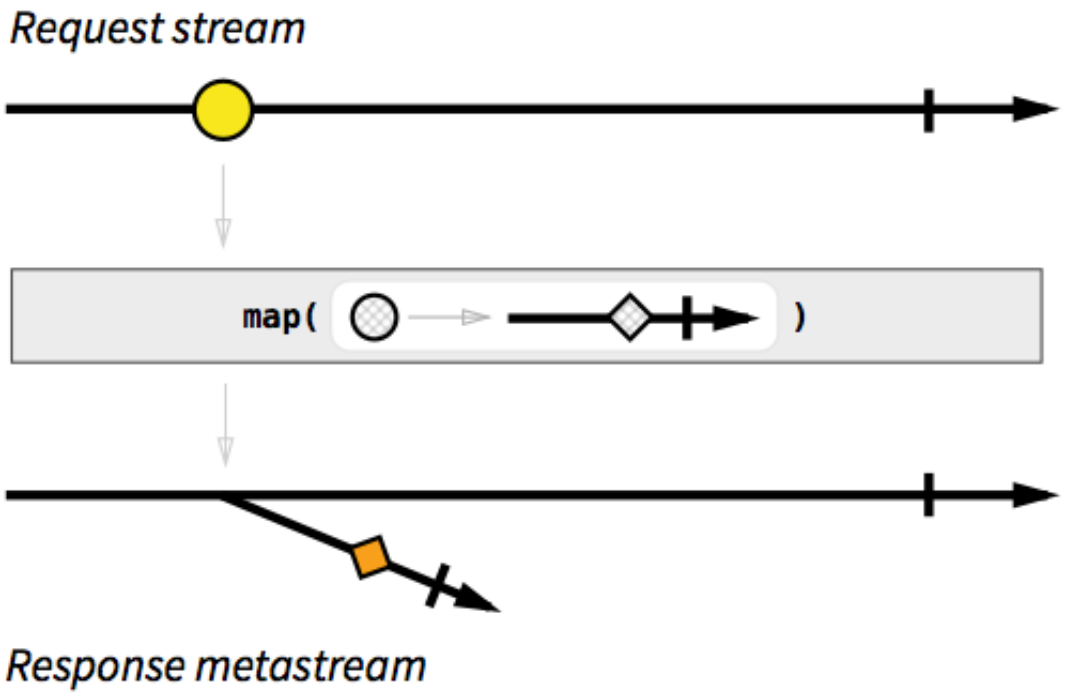
这样更好，这样更突出被观察者至少比Promise强大，所以如果你相信Promise宣传的东西，那么也请留意一下响应式编程能胜任些什么。

现在回到示例当中，你应该能快速发现，我们在subscribe()方法的内部再次调用了subscribe()方法，这有点类似于回调地狱(callback hell)，而且responseStream的创建也是依赖于requestStream的。在之前我们说过，在Rx里，有很多很简单的机制来从其他事件流的转化并创建出一些新的事件流，那么，我们也应该这样做试试。

现在你需要了解的一个最基本的函数是map(f)，它可以从事件流A中取出每一个值，并对每一个值执行f()函数，然后将产生的新值填充到事件流B。如果将它应用到我们的请求和响应事件流当中，那我们就可以将请求的URL映射到一个响应Promises上了(伪装成数据流)。

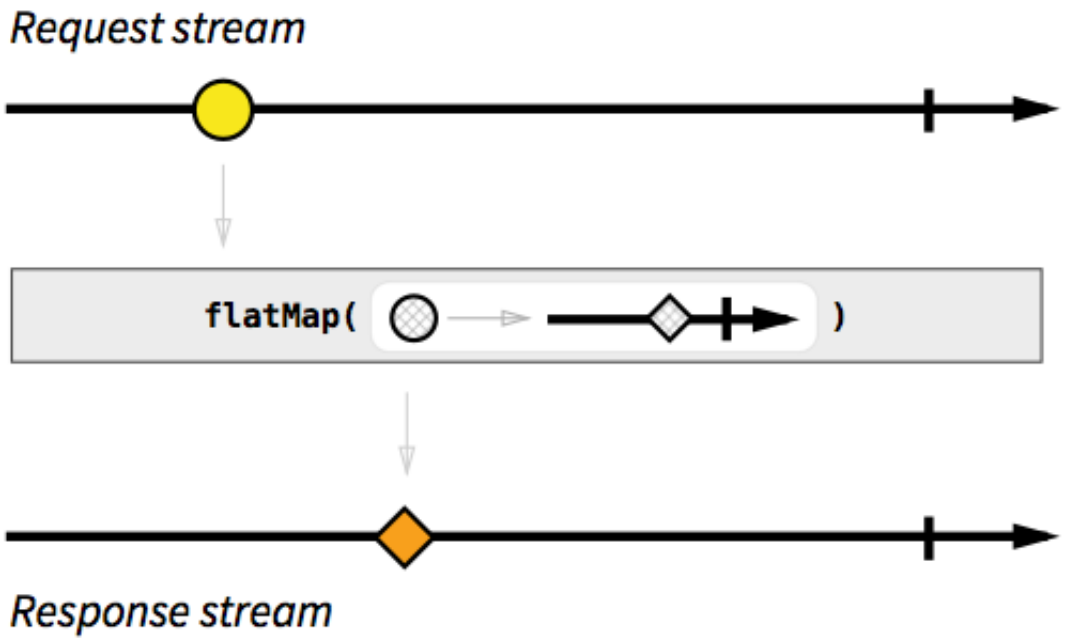
```
1  var responseMetastream = requestStream
2    .map(function(requestUrl) { return Rx.Observable.fromPromise(jQue
3    });
```

然后，我们创造了一个叫做"metastream"的怪兽：一个装载了事件流的事件流。先别惊慌，metastream就是每一个发出的值都是另一个事件流的事件流，你看把它想象成一个[指针(pointers)]((https://en.wikipedia.org /wiki/Pointer (computer programming))数组：每一个单独发出的值就是一个 指针 ，它指向另一个事件流。在我们的示例里，每一个请求URL都映射到一个指向包含响应数据的promise数据流。



一个响应的metastream，看起来确实让人容易困惑，看样子对我们一点帮助也没有。我们只想要一个简单的响应数据流，每一个发出的值是一个简单的JSON对象就行，而不是一个'Promise' 的JSON对象。ok，让我们来见识一下另一个函数：Flatmap，它是map()函数的另一个版本，它比metastream更扁平。一切在"主躯干"事件流发出的事件都将在"分支"事件流中发出。Flatmap并不是metastreams的修复版，metastreams也不是一个bug。它俩在Rx中都是处理异步响应事件的好工具、好帮手。

```
1  var responseStream = requestStream
2    .flatMap(function(requestUrl) { return Rx.Observable.fromPromise(
3    });
```



很赞，因为我们的响应事件流是根据请求事件流定义的，如果我们以后有更多事件发生在请求事件流的话，我们也将会在相应的响应事件流收到响应事件，就如所期待的那样：

```
requestStream:  --a-----b--c------------|->
responseStream: -----A--------B-----C---|->

(小写的是请求事件流，大写的是响应事件流)
```

现在，我们终于有响应的事件流了，并且可以用我们收到的数据来渲染了：

```
responseStream.subscribe(function(response) { // render `response`
```

让我们把所有代码合起来，看一下：

```
var requestStream = Rx.Observable.just('https://api.github.com/use
  .flatMap(function(requestUrl) { return Rx.Observable.fromPromise(
  });

responseStream.subscribe(function(response) { // render `response`
```

刷新按钮

我还没提到本次响应的JSON数据是含有100个用户数据的list，这个API只允许指定页面偏移量(page offset)，而不能指定每页大小(page size)，我们只用到了3个用户数据而浪费了其他97个，现在可以先忽略这个问题，稍后我们将学习如何缓存响应的数据。

每当刷新按钮被点击，请求事件流就会发出一个新的URL值，这样我们就可以获取新的响应数据。这里我们需要两个东西：点击刷新按钮的事件流(准则：一切都能作为事件流)，我们需要将点击刷新按钮的事件流作为请求事件流的依赖(即点击刷新事件流会引起请求事件流)。幸运的是，RxJS已经有了可以从事件监听者转换成被观察者的方法了。

```
var refreshButton = document.querySelector('.refresh'); var refres
```

因为刷新按钮点击事件不会携带将要请求的API的URL，我们需要将每次的点击映射到一个实际的URL上，现在我们将请求事件流转换成了一个点击事件流，并将每次的点击映射成一个随机的页面偏移量(offset)参数来组成API的URL。

```
var requestStream = refreshClickStream
  .map(function() { var randomOffset = Math.floor(Math.random()*500
  });
```

因为我比较笨而且也没有使用自动化测试，所以我刚把之前做好的一个功能搞烂了。这样，请求在一开始的时候就不会执行，而只有在点击事件发生时才会执行。我们需要的是两种情况都要执行：刚开始打开网页和点击刷新按钮都会执行的请求。

我们知道如何为每一种情况做一个单独的事件流：

```
var requestOnRefreshStream = refreshClickStream
  .map(function() { var randomOffset = Math.floor(Math.random()*500
  }); var startupRequestStream = Rx.Observable.just('https://api.gi
```

但是我们是否可以将这两个合并成一个呢？没错，是可以的，我们可以使用merge()方法来实现。下图可以解释merge()函数的用处：

```
stream A: ---a--------e-----o----->
stream B: -----B---C-----D-------->
          vvvvvvvvv merge vvvvvvvvv
          ---a-B---C--e--D--o----->
```

现在做起来应该很简单：

```
var requestOnRefreshStream = refreshClickStream
  .map(function() { var randomOffset = Math.floor(Math.random()*500
  }); var startupRequestStream = Rx.Observable.just('https://api.gi
  requestOnRefreshStream, startupRequestStream
);
```

还有一个更干净的写法，省去了中间事件流变量：

```
var requestStream = refreshClickStream
  .map(function() { var randomOffset = Math.floor(Math.random()*500
  })
  .merge(Rx.Observable.just('https://api.github.com/users'));
```

甚至可以更简短，更具有可读性：

```
var requestStream = refreshClickStream
  .map(function() { var randomOffset = Math.floor(Math.random()*500
  })
  .startWith('https://api.github.com/users');
```

startWith()函数做的事和你预期的完全一样。无论你的输入事件流是怎样的，使用startWith(x)函数处理过后输出的事件流一定是一个x开头的结果。但是我没有总是重复代码(DRY)，我只是在重复API的URL字符串，改进的方法是将startWith()函数挪到refreshClickStream那里，这样就可以在启动时，模拟一个刷新按钮的点击事件了。

```
var requestStream = refreshClickStream.startWith('startup click')
  .map(function() { var randomOffset = Math.floor(Math.random()*500
  });
```

不错，如果你倒回到"搞烂了的自动测试"的地方，然后再对比这两个地方，你会发现我仅仅是加了一个startWith()函数而已。

用事件流将3个推荐的用户数据模型化

直到现在，在响应事件流(responseStream)的订阅(subscribe())函数发生的渲染步骤里，我们只是稍微提及了一下推荐关注的UI。现在有了刷新按钮，我们就会出现一个问题：当你点击了刷新按钮，当前的三个推荐关注用户没有被清楚，而只要响应的数据达到后我们就拿到了新的推荐关注的用户数据，为了让UI看起来更漂亮，我们需要在点击刷新按钮的事件发生的时候清楚当前的三个推荐关注的用户。

```
refreshClickStream.subscribe(function() { // clear the 3 suggestio
```

不，老兄，还没那么快。我们又出现了新的问题，因为我们现在有两个订阅者在影响着推荐关注的UI DOM元素(另一个是responseStream.subscribe())，这看起来并不符合关注分离(Separation of concerns)原则，还记得响应式编程的原则么？

现在，让我们把推荐关注的用户数据模型化成事件流形式，每个被发出的值是一个包含了推荐关注用户数据的JSON对象。我们将把这三个用户数据分开处理，下面是推荐关注的1号用户数据的事件流：

```
var suggestion1Stream = responseStream
  .map(function(listUsers) { // get one random user from the list r
  });
```

其他的，如推荐关注的2号用户数据的事件流suggestion2Stream和推荐关注的3号用户数据的事件流suggestion3Stream都可以方便的从suggestion1Stream复制粘贴就好。这里并不是重复代码，只是为让我们的示例更加简单，而且我认为这是一个思考如何避免重复代码的好案例。

Instead of having the rendering happen in responseStream's subscribe(), we do that here:

```
suggestion1Stream.subscribe(function(suggestion) { // render the 1
```

我们不在responseStream的subscribe()中处理渲染了，我们这样处理：

```
suggestion1Stream.subscribe(function(suggestion) { // render the 1
```

回到"当刷新时，清楚掉当前的推荐关注的用户"，我们可以很简单的把刷新点击映射为没有推荐数据(nullsuggestion data)，并且在suggestion1Stream中包含进来，如下：

```
var suggestion1Stream = responseStream
  .map(function(listUsers) { // get one random user from the list r
  })
  .merge(
```

```
    refreshClickStream.map(function(){ return null; })
  );
```

当渲染时，我们将null解释为"没有数据"，然后把UI元素隐藏起来。

```
suggestion1Stream.subscribe(function(suggestion) {
  if (suggestion === null) {
    // hide the first suggestion DOM element
  }
  else {
    // show the first suggestion DOM element
    // and render the data
  }
});
```

现在我们大概的示意图如下：

```
refreshClickStream: ----------o--------o---->
     requestStream: -r--------r--------r---->
    responseStream: ----R---------R------R-->
 suggestion1Stream: ----s-----N---s----N-s-->
 suggestion2Stream: ----q-----N---q----N-q-->
 suggestion3Stream: ----t-----N---t----N-t-->
```

N代表null

作为一种补充，我们可以在一开始的时候就渲染空的推荐内容。这通过把startWith(null)添加到推荐关注的事件流就可以了：

```
var suggestion1Stream = responseStream
  .map(function(listUsers) { // get one random user from the list r
  })
  .merge(
    refreshClickStream.map(function(){ return null; })
  )
  .startWith(null);
```

结果是这样的：

```
refreshClickStream: ----------o---------o---->
     requestStream: -r--------r---------r---->
    responseStream: ----R----------R------R-->
 suggestion1Stream: -N--s-----N----s----N-s-->
 suggestion2Stream: -N--q-----N----q----N-q-->
 suggestion3Stream: -N--t-----N----t----N-t-->
```

推荐关注的关闭和使用已缓存的响应数据(responses)

只剩这一个功能没有实现了，每个推荐关注的用户UI会有一个'x'按钮来关闭自己，然后在当前的用户数据UI中加载另一个推荐关注的用户。最初的想法是：点击任何关闭按钮时都需要发起一个新的请求：

```
var close1Button = document.querySelector('.close1');
var close1ClickStream = Rx.Observable.fromEvent(close1Button, 'cli
// and the same for close2Button and close3Button

var requestStream = refreshClickStream.startWith('startup click')
  .merge(close1ClickStream) // we added this
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });
```

这样没什么效果，这样会关闭和重新加载全部的推荐关注用户，而不仅仅是处理我们点击的那一个。这里有几种方式来解决这个问题，并且让它变得有趣，我们将重用之前的请求数据来解决这个问题。这个API响应的每页数据大小是100个用户数据，而我们只使用了其中三个，所以还有一大堆未使用的数据可以拿来用，不用去请求更多数据了。

ok，再来，我们继续用事件流的方式来思考。当'close1'点击事件发生时，我们想要使用最近发出的响应数据，并执行responseStream函数来从响应列表里随机的抽出一个用户数据来，就像下面这样：

```
requestStream: --r--------------->
   responseStream: ------R----------->
close1ClickStream: ------------c----->
suggestion1Stream: ------s-----s----->
```

在Rx中一个组合函数叫做[combineLatest](), 应该是我们需要的。这个函数会把数据流A和数据流B作为输入，并且无论哪一个数据流发出一个值了，combineLatest函数就会将从两个数据流最近发出的值a和b作为f函数的输入，计算后返回一个输出值(c = f(x,y))，下面的图表会让这个函数的过程看起来会更加清晰：

```
stream A: --a----------e--------i-------->
stream B: -----b----c--------d-------q---->
          vvvvvvvv combineLatest(f) vvvvvvv
          ----AB---AC--EC---ED--ID--IQ---->

f是转换成大写的函数
```

这样，我们就可以把combineLatest()函数用在close1ClickStream和responseStream上了，只要关闭按钮被点击，我们就可以获得最近的响应数据，并在suggestion1Stream上产生出一个新值。另一方面，combineLatest()函数也是相对的：每当在responseStream上发出一个新的响应，它将会结合一次新的点击关闭按钮事件来产生一个新的推荐关注的用户数据，这非常有趣，因为它可以给我们的suggestion1Stream简化代码：

```
var suggestion1Stream = close1ClickStream
  .combineLatest(responseStream, function(click, listUsers) { retur
    }
  )
  .merge(
    refreshClickStream.map(function(){ return null; })
  )
  .startWith(null);
```

现在，我们的拼图还缺一小块地方。combineLatest()函数使用了最近的两个数据源，但是如

果某一个数据源还没有发出任何东西，combineLatest()函数就不能在输出流上产生一个数据事件。如果你看了上面的ASCII图表(文章中第一个图表)，你会明白当第一个数据流发出一个值a时并没有任何的输出，只有当第二个数据流发出一个值b的时候才会产生一个输出值。

这里有很多种方法来解决这个问题，我们使用最简单的一种，也就是在启动的时候模拟'close 1'的点击事件：

```
var suggestion1Stream = close1ClickStream.startWith('startup clic
  .combineLatest(responseStream,
    function(click, listUsers) {l
      return listUsers[Math.floor(Math.random()*listUsers.length)]
    }
  )
  .merge(
    refreshClickStream.map(function(){ return null; })
  )
  .startWith(null);
```

封装起来

我们完成了，下面是封装好的完整示例代码：

```
var refreshButton = document.querySelector('.refresh');
var refreshClickStream = Rx.Observable.fromEvent(refreshButton, 'c

var closeButton1 = document.querySelector('.close1');
var close1ClickStream = Rx.Observable.fromEvent(closeButton1, 'cli
// and the same logic for close2 and close3

var requestStream = refreshClickStream.startWith('startup click')
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });

var responseStream = requestStream
  .flatMap(function (requestUrl) {
    return Rx.Observable.fromPromise($.ajax({url: requestUrl}));
  });

var suggestion1Stream = close1ClickStream.startWith('startup click
  .combineLatest(responseStream,
    function(click, listUsers) {
      return listUsers[Math.floor(Math.random()*listUsers.length)]
    }
  )
  .merge(
    refreshClickStream.map(function(){ return null; })
  )
  .startWith(null);
// and the same logic for suggestion2Stream and suggestion3Stream

suggestion1Stream.subscribe(function(suggestion) {
```

```
32    if (suggestion === null) {
33      // hide the first suggestion DOM element
34    }
35    else {
36      // show the first suggestion DOM element
37      // and render the data
38    }
39  });
```

你可以在这里看到可演示的示例工程

以上的代码片段虽小但做到很多事：它适当的使用关注分离(separation of concerns)原则的实现了对多个事件流的管理，甚至做到了响应数据的缓存。这种函数式的风格使得代码看起来更像是声明式编程而非命令式编程：我们并不是在给一组指令去执行，只是定义了事件流之间关系来告诉它这是什么。例如，我们用Rx来告诉计算机suggestion1Stream是'close 1'事件结合从最新的响应数据中拿到的一个用户数据的数据流，除此之外，当刷新事件发生时和程序启动时，它就是null。

留意一下代码中并未出现例如if,for,while等流程控制语句，或者像JavaScript那样典型的基于回调(callback-based)的流程控制。如果可以的话(稍候会给你留一些实现细节来作为练习)，你甚至可以在subscribe()上使用filter()函数来摆脱if和else。在Rx里，我们有例如：map,filter,scan,merge,combineLatest,startWith等数据流的函数，还有很多函数可以用来控制事件驱动编程(event-driven program)的流程。这些函数的集合可以让你使用更少的代码实现更强大的功能。

接下来

如果你认为Rx将会成为你首选的响应式编程库，接下来就需要花一些时间来熟悉一大批的函数用来变形、联合和创建被观察者。如果你想在事件流的图表当中熟悉这些函数，那就来看一下这个：RxJava's very useful documentation with marble diagrams。请记住，无论何时你遇到问题，可以画一下这些图，思考一下，看一看这一大串函数，然后继续思考。以我个人经验，这样效果很有效。

一旦你开始使用了Rx编程，请记住，理解Cold vs Hot Observables的概念是非常必要的，如果你忽视了这一点，它就会反弹回来并残忍的反咬你一口。我这里已经警告你了，学习函数式编程可以提高你的技能，熟悉一些常见问题，例如Rx会带来的副作用

但是响应式编程库并不仅仅是Rx，还有相对容易理解的，没有Rx那些怪癖的Bacon.js。Elm Language则以它自己的方式支持响应式编程：它是一门会编译成Javascript + HTML + CSS的响应式编程语言，并有一个time travelling debugger功能，很棒吧。

而Rx对于像前端和App这样需要处理大量的编程效果是非常棒的。但是它不只是可以用在客户端，还可以用在后端或者接近数据库的地方。事实上，RxJava就是Netflix服务端API用来处理并行的组件。Rx并不是局限于某种应用程序或者编程语言的框架，它真的是你编写任何事件驱动程序，可以遵循的一个非常棒的编程范式。

如果这篇教程对你有帮助，那么就请来转发一下吧(tweet it forward).

相关资讯 — 更多

- 我感觉到的前端变化

相关文档 — 更多

- iOS平台响应式编程Functional

- 15个国外响应式网页设计经典教程推荐
- FEX 技术周刊-2015/07/20
- 响应式编程的基本概念
- 2015年16个最佳的免费响应式HTML5框架
- 设计师不应该错过的响应式设计框架
- 响应式设计的现状与趋势
- 2015年15+最佳的响应式HTML5网站模板
- 响应式设计不值得搞的5个原因
- 网页响应式设计的现状与趋势
- Web开发人员不容错过的十款最佳HTML5响应式框架
- 你的网站需要针对移动端优化的10个理由
- 细数2014年5个最流行的前端框架
- 2014年10个最好的免费响应式HTML5/CSS3框架
- .Net响应式编程框架Rx的Java实现，RxJava 1.0.16 发布
- 《通过Actor模型实现响应式消息处理模式》书评及与Vaughn Vernon的问答
- .Net响应式编程框架Rx的Java版本开源实现，RxJava 1.0.15 发布
- 2014 年 20 个最好的响应式 HTML5 框架
- 2015年 16 个最好的免费响应式HTML 5 框架
- 2015 年网站设计我们将看到这十大趋势
- Reactive Programming on iOS.pdf
- 利用Bootstrap 3设计移动优先的响应式网站.pdf
- The Responsive Web 响应式设计.pdf
- Ctrip 首页响应式设计.pptx
- 构建响应式Web页面布局.pdf
- 通过实例介绍响应式Web设计.pdf
- Bootstrap用户手册：设计响应式网站.pdf
- 响应式 web.pdf
- 基于scala/akka构建响应式流计算.pptx
- 响应式Web设计：HTML5和CSS3实战.doc
- 响应式web设计：HTML5和CSS3实战(简单版).pdf
- 响应式Web设计：HTML5 和CSS3实战.pdf
- 响应式web设计实践试读.pdf
- 响应式Web设计：HTML5和CSS3实战.pdf
- 响应式web设计：html5和css3实战.pdf
- 响应式web设计：HTML5和CSS3实战.pdf
- 响应式Web设计：HTML5和CSS3实战.pdf
- 响应式Web设计：HTML5和CSS3实战.pdf
- 响应式Web设计：HTML5和CSS3实战.pdf
- Bootstrap 用户手册：响应式设计.pdf

内容信息

0.0

(已有0个人评价)

0%

0%

0%

0%

0%

收藏：1人 发布时间1：2015-04-17 21:31:14

关注网站微信公众号

经验标签
响应式
同类热门经验

- 前端开发资源大全
  37071次浏览
- Twitter Bootstrap 框架介绍
  52572次浏览
- Google Web应用开发指南第二章：交互设计
  15835次浏览
- 基于HTML5的前端UI框架 - Kitjs
  48957次浏览
- 使用 Metro Studio 创建 Metro 风格的图标
  20649次浏览
- CSS基础代码库 Nice UE
  20516次浏览

相关经验

- RxJava资源集合
  0人评
- Dapper，大规模分布式系统的跟踪系统
  0人评
- 【译】Android应用架构
  0人评
- 细谈Android应用架构
  0人评
- 码农周刊分类整理
  1人评
- 【译】Android应用架构
  0人评
- AngularJS - 下一个大框架
  1人评
- 深入浅出-iOS函数式编程 && 响应式编程概念
  0人评
- iOS响应式编程：ReactiveCocoa vs RxSwift 选谁好
  0人评
- RxSwift 函数响应式编程
  0人评
- 响应式设计介绍

0人评

- 响应式的 Web 设计库 Gridpak
  0人评
- 响应式的表格插件：Tablesaw
  0人评
- 响应式的 CSS 框架 MQFramework
  0人评

相关讨论 - 更多

- 一些好的规则
- 编程趋势
- Web编程是函数式编程
- Web开发者必备的15个非常有用的HTML5工具和资源
- 少即是极多
- 什么是Node.js?
- 那些年，追过的开源软件和技术