**转换成 observables**

// 来自一个或多个值

Rx.Observable.of('foo', 'bar');

// 来自数组

Rx.Observable.from([1,2,3]);

// 来自事件

Rx.Observable.fromEvent(document.querySelector('button'), 'click');

// 来自 Promise

Rx.Observable.fromPromise(fetch('/users'));

// 来自回调函数(最后一个参数得是回调函数，比如下面的 cb)

// fs.exists = (path, cb(exists))

var exists = Rx.Observable.bindCallback(fs.exists);

exists('file.txt').subscribe(exists => console.log('Does file exist?', exists));

// 来自回调函数(最后一个参数得是回调函数，比如下面的 cb)

// fs.rename = (pathA, pathB, cb(err, result))

var rename = Rx.Observable.bindNodeCallback(fs.rename);

rename('file.txt', 'else.txt').subscribe(() => console.log('Renamed!'));

**创建 observables**

在外部产生新事件。

var myObservable = new Rx.Subject();

myObservable.subscribe(value => console.log(value));

myObservable.next('foo');

在内部产生新事件。

var myObservable = Rx.Observable.create(observer => {

observer.next('foo');

setTimeout(() => observer.next('bar'), 1000);

});

myObservable.subscribe(value => console.log(value));

选择哪种方式需要根据场景。当你想要包装随时间推移产生值的功能时，普通的 **Observable** 就已经很好了。使用 **Subject**，你可以从任何地方触发新事件，并且将已存在的 observables 和它进行连接。

**控制流动**

// 输入 "hello world"

var input = Rx.Observable.fromEvent(document.querySelector('input'), 'input');

// 过滤掉小于3个字符长度的目标值

input.filter(event => event.target.value.length > 2)

.map(event => event.target.value)

.subscribe(value => console.log(value)); // "hel"

// 延迟事件

input.delay(200)

.map(event => event.target.value)

.subscribe(value => console.log(value)); // "h" -200ms-> "e" -200ms-> "l" ...

// 每200ms只能通过一个事件

input.throttleTime(200)

.map(event => event.target.value)

.subscribe(value => console.log(value)); // "h" -200ms-> "w"

// 停止输入后200ms方能通过最新的那个事件

input.debounceTime(200)

.map(event => event.target.value)

.subscribe(value => console.log(value)); // "o" -200ms-> "d"

// 在3次事件后停止事件流

input.take(3)

.map(event => event.target.value)

.subscribe(value => console.log(value)); // "hel"

// 直到其他 observable 触发事件才停止事件流

var stopStream = Rx.Observable.fromEvent(document.querySelector('button'), 'click');

input.takeUntil(stopStream)

.map(event => event.target.value)

.subscribe(value => console.log(value)); // "hello" (点击才能看到)

**产生值**

// 输入 "hello world"

var input = Rx.Observable.fromEvent(document.querySelector('input'), 'input');

// 传递一个新的值

input.map(event => event.target.value)

.subscribe(value => console.log(value)); // "h"

// 通过提取属性传递一个新的值

input.pluck('target', 'value')

.subscribe(value => console.log(value)); // "h"

// 传递之前的两个值

input.pluck('target', 'value').pairwise()

.subscribe(value => console.log(value)); // ["h", "he"]

// 只会通过唯一的值

input.pluck('data').distinct()

.subscribe(value => console.log(value)); // "helo wrd"

// 不会传递重复的值

input.pluck('data').distinctUntilChanged()

.subscribe(value => console.log(value)); // "helo world"

**创建应用**

RxJS 是个很好的工具，可以让你的代码更少出错。它是通过使用无状态的纯函数来做到这点的。但是应用是有状态的，那么我们如何将 RxJS 的无状态世界与我们应用的有状态世界连接起来呢？

我们来创建一个只存储值为0的简单状态。每次点击我们想要增加存储在状态中的 count 。

var button = document.querySelector('button');

Rx.Observable.fromEvent(button, 'click')

// 对流进行 scan (reduce) 操作，以获取 count 的值

.scan(count => count + 1, 0)

// 每次改变时都在元素上设置 count

.subscribe(count => document.querySelector('#count').innerHTML = count);

所以产生状态是在 RxJS 的世界中完成的，但最后一行代码中改变 DOM 却是一种副作用。

**状态和存储 (State Store)**

应用使用状态和存储来保持状态。状态存储在不同的框架中有着不同的名称，像 store、reducer 和 model ，但重点是它们都只是普通的对象。我们还需要处理的是多个 observables 可以更新同一个状态存储。

var increaseButton = document.querySelector('#increase');

var increase = Rx.Observable.fromEvent(increaseButton, 'click')

// 我们映射到一个函数，它会改变状态

.map(() => state => Object.assign({}, state, {count: state.count + 1}));

我们在这所做的是将点击事件映射成改变状态的函数。所以我们映射到一个函数，而不是映射到一个值。函数会改变状态存储中的状态。那么现在我们来看下如何实际地做出改变。

var increaseButton = document.querySelector('#increase');

var increase = Rx.Observable.fromEvent(increaseButton, 'click')

.map(() => state => Object.assign({}, state, {count: state.count + 1}));

// 我们使用初始状态创建了一个对象。每当状态发生变化时，我们会接收到改变状态的函数，

// 并把状态传递给它。然后返回新的状态并准备在下次点击后再次更改状态。

var state = increase.scan((state, changeFn) => changeFn(state), {count: 0});

现在我们还可以再添加几个 observables ，它们同样也可以更改同一个状态存储。

var increaseButton = document.querySelector('#increase');

var increase = Rx.Observable.fromEvent(increaseButton, 'click')

// 我们再一次映射到一个函数，它会增加 count

.map(() => state => Object.assign({}, state, {count: state.count + 1}));

var decreaseButton = document.querySelector('#decrease');

var decrease = Rx.Observable.fromEvent(decreaseButton, 'click')

// 我们还是映射到一个函数，它会减少 count

.map(() => state => Object.assign({}, state, {count: state.count - 1}));

var inputElement = document.querySelector('#input');

var input = Rx.Observable.fromEvent(inputElement, 'keypress')

// 我们还将按键事件映射成一个函数，它会产生一个叫做 inputValue 状态

.map(event => state => Object.assign({}, state, {inputValue: event.target.value}));

// 我们将这三个改变状态的 observables 进行合并

var state = Rx.Observable.merge(

increase,

decrease,

input

).scan((state, changeFn) => changeFn(state), {

count: 0,

inputValue: ''

});

// 我们订阅状态的变化并更新 DOM

state.subscribe((state) => {

document.querySelector('#count').innerHTML = state.count;

document.querySelector('#hello').innerHTML = 'Hello ' + state.inputValue;

});

// 为了优化渲染，我们可以检查什么状态是实际上已经发生变化了的

var prevState = {};

state.subscribe((state) => {

if (state.count !== prevState.count) {

document.querySelector('#count').innerHTML = state.count;

}

if (state.inputValue !== prevState.inputValue) {

document.querySelector('#hello').innerHTML = 'Hello ' + state.inputValue;

}

prevState = state;

});

我们可以采用状态存储的方式，并且结合一些不同的框架和库来进行使用。

**Immutable JS**

你还可以使用 [Immutable JS](https://facebook.github.io/immutable-js/) 来为你的应用创建一个全局的状态存储。Immutable JS 是创建不变的状态存储的好方法，它允许你通过对更改的值进行浅检查来优化渲染。

import Immutable from 'immutable';

import someObservable from './someObservable';

import someOtherObservable from './someOtherObservable';

var initialState = {

foo: 'bar'

};

var state = Observable.merge(

someObservable,

someOtherObservable

).scan((state, changeFn) => changeFn(state), Immutable.fromJS(initialState));

export default state;

现在，你可以在使用的任何 UI 层中导入状态。

import state from './state';

state.subscribe(state => {

document.querySelector('#text').innerHTML = state.get('foo');

});

**React**

我们来看一个示例，当组件进入 componentDidMount 生命周期事件时订阅 observable，而当进入 componentWillUnmount 生命周期事件时取消订阅。

import messages from './someObservable';

class MyComponent extends ObservableComponent {

constructor(props) {

super(props);

this.state = {messages: []};

}

componentDidMount() {

this.messages = messages

// 在数组中累积我们的消息

.scan((messages, message) => [message].concat(messages), [])

// 当得到一条新消息时进行渲染

.subscribe(messages => this.setState({messages: messages}));

}

componentWillUnmount() {

this.messages.unsubscribe();

}

render() {

return (

<div>

<ul>

{this.state.messages.map(message => <li>{message.text}</li>)}

</ul>

</div>

);

}

}

export default MyComponent;

还有许多其他的方式可以使用 React 和 observables。看看这些：

* [rxjs-react-component](https://www.npmjs.com/package/rxjs-react-component)。它允许你暴露映射到状态更改的 observable 。还可以使用 observable 版的生命周期钩子。

**外部参考资料**

还没有学到足够的 RxJS 知识？查看其他这些超级棒的资源吧！

译者注：下面不少资源还是基于 RxJS 4，但思想是相同的 (=.=)

**教程**

* [RxJS 课程 - Egghead.io](https://egghead.io/technologies/rx)
* [不容错过的响应式编程介绍](https://gist.github.com/staltz/868e7e9bc2a7b8c1f754)
* [2分钟告诉你什么是 Rx](https://medium.com/@andrestaltz/2-minute-introduction-to-rx-24c8ca793877)
* [学习 RxJS - @jhusain](https://github.com/jhusain/learnrx)
* [Rx 工场](http://rxworkshop.codeplex.com/)
* [响应式编程和 MVC](http://aaronstacy.com/writings/reactive-programming-and-mvc/)
* [RxJS 培训 - @andrestaltz](https://github.com/staltz/rxjs-training)

**书籍**

* [RxJS 实战](https://www.manning.com/books/rxjs-in-action)
* [Rx Book](http://xgrommx.github.io/rx-book/)
* [Rx 介绍](http://www.amazon.com/Introduction-to-Rx-ebook/dp/B008GM3YPM/)
* [响应式编程扩展 LINQ](http://www.amazon.com/Programming-Reactive-Extensions-Jesse-Liberty/dp/1430237473/)
* [使用 RxJS 进行响应式编程](https://pragprog.com/book/smreactjs/reactive-programming-with-rxjs)

**视频**

* [实用的 Rx - Channel 9](http://channel9.msdn.com/posts/Bart-De-Smet-Jafar-Hussain-Matthew-Podwysocki-Pragmatic-Rx)
* [Netflix 和 RxJS - Channel 9](http://channel9.msdn.com/posts/Rx-and-Netflix)
* [你好 RxJS - Channel 9](http://channel9.msdn.com/Blogs/Charles/Introducing-RxJS-Reactive-Extensions-for-JavaScript)
* [RxJS - MIX 2011 - Channel 9](http://channel9.msdn.com/events/MIX/MIX11/HTM07)
* [RxJS 的今天和明天 - Channel 9](http://channel9.msdn.com/Blogs/Charles/Matthew-Podwysocki-and-Bart-J-F-De-Smet-RxJS-Today-and-Tomorrow)
* [响应式编程扩展视频 - Channel 9](http://channel9.msdn.com/Tags/reactive+extensions)
* [Netflix 的异步 JavaScript - Netflix JavaScript 会话 - Jafar Husain](https://www.youtube.com/watch?v=XRYN2xt11Ek)
* [Netflix 的异步 JavaScript - MountainWest JavaScript 2014 - Jafar Husain](https://www.youtube.com/watch?v=XE692Clb5LU)
* [Netflix 的异步 JavaScript - HTML5DevConf - Jafar Husain](https://www.youtube.com/watch?v=5uxSu-F5Kj0)
* [使用 RxJS 来为函数式编程添加更多乐趣 - Ryan Anklam](https://www.youtube.com/watch?v=8EExNfm0gt4)
* [响应式的 Angular - Devoxx France 2014 - Martin Gontovnikas](http://parleys.com/play/53677646e4b0593229b85841/chapter0/about)
* [响应式游戏编程的时尚潮流 - JSConf 2014 - Bodil Stokke](https://www.youtube.com/watch?v=x8mmAu7ZR9Y)