
Table of Contents

Overview

Introduction	1.1
Rookie primer	1.2
Observable	1.3
Observer	1.4
Subscription	1.5
Subject	1.6
Operators	1.7
Scheduler	1.8

Installation

Installation	2.1
--------------	-----

Tutorial

Tutorial	3.1
----------	-----

static operators

what are operators?	4.1
bindCallback	4.2
bindNodeCallback	4.3
combineLatest	4.4
concat	4.5
create	4.6
deffer	4.7

empty	4.8
forkJoin	4.9
from	4.10
fromEvent	4.11
fromEventPattern	4.12
fromPromise	4.13
interval	4.14
merge	4.15
never	4.16
of	4.17
range	4.18
throw	4.19
timer	4.20
toAsync	4.21
using	4.22
when	4.23
while	4.24
wrap	4.25
webSocket	4.26
zip	4.27

instance operators

audit	5.1
auditTime	5.2
buffer	5.3
bufferCount	5.4
bufferTime	5.5
bufferToggle	5.6
bufferWhen	5.7
catch	5.8

combineAll	5.9
combineLatest	5.10

RxJS-Chinese

自己学习的时候发现国内资源甚少，就顺便翻译了下来，为大家提供个方便。由于本人能力有限，所以疏漏和错误之处不可避免。翻译的初衷也是为了自己能够加深理解能够方便大家。如大家发现疏漏错误的地方请及时告知，以免误导他人。最后，我强烈建议大家有能力的最好能阅读英文官方原文并以此为标准，而把译文当做参考。

学习该文档的基础知识链接：

阮老师的ES6入门教程[ES6](#)

TS中文文档[TypeScript](#)

本篇对应于官方的介绍篇，因英文介绍与`gitbook`文件名冲突，所以改了一下

RxJS是一个通过使用可观察序列来构建异步和基于事件的程序的库。它提供了一个核心类型:**Observable**、卫星类型(大概是这些类型均围绕于**Observable**，也就是**Observable**是根基，而这些是辅助类型):**Observer**、**Schedulers**、**Subjects**)和操作符-衍生自一些数组方法，使得我们可以把异步事件以集合的方式进行处理。

把RxJS当做一个针对事件的Lodash(一个JS库)。

ReactiveX将观察者模式与迭代器模式和使用集合的函数式编程组合在一起，来满足这种管理事件序列的理想方式

RxJS中解决异步事件管理的基本概念如下：

- **Observable**可观察对象：表示一个可调用的未来值或者事件的集合。
- **Observer**观察者：一个回调函数集合,它知道怎样去监听被**Observable**发送的值
- **Subscription**订阅：表示一个可观察对象的执行，主要用于取消执行。
- **Operators**操作符：纯粹的函数，使得以函数编程的方式处理集合比如:`map`,`filter`,`contact`,`flatMap`。
- **Subject**(主题)：等同于一个事件驱动器，是将一个值或者事件广播到多个观察者的唯一途径。
- **Schedulers**(调度者)：用来控制并发，当计算发生的时候允许我们协调，比如`setTimeout`,`requestAnimationFrame`。

第一个例子

通常你这样注册事件监听：

```
var button = document.querySelector('button');
button.addEventListener('click', () => console.log('Clicked!'));
```

使用RxJS创建一个可观察对象：

```
var button = document.querySelector('button');
Rx.Observable.fromEvent(button, 'click')
  .subscribe(() => console.log('Clicked!'));
```

Purity

RxJS能够使用纯函数的方式生产值的能力使得它强大无比。这意味着你的代码不再那么频繁的出现错误提示。

通常情况下你会创建一个非纯粹的函数，然后你的代码的其他部分可能搞乱你的程序状态。

```
var count = 0;
var button = document.querySelector('button');
button.addEventListener('click', () => console.log(`Clicked ${++count} times`));
```

使用RxJS来隔离你的状态

```
var button = document.querySelector('button');
Rx.Observable.fromEvent(button, 'click')
  .scan(count => count + 1, 0)
  .subscribe(count => console.log(`Clicked ${count} times`));
```

scan操作符和数组中reduce方法的类似，它需要一个传递给回调函数的参数值。回调函数的返回值将成为下一次回调函数运行时要传递的下一个参数值。

Flow 流

RxJS有着众多的操作符，可以帮助您控制事件如何流入可观察对象observables。

每秒最多只能点击一次的实现，使用纯JavaScript：

```
var count = 0;
var rate = 1000;
var lastClick = Date.now() - rate;
var button = document.querySelector('button');
button.addEventListener('click', () => {
  if (Date.now() - lastClick >= rate) {
    console.log(`Clicked ${++count} times`);
    lastClick = Date.now();
  }
});
```

使用RxJS

```
var button = document.querySelector('button');
Rx.Observable.fromEvent(button, 'click')
  .throttleTime(1000)
  .scan(count => count + 1, 0)
  .subscribe(count => console.log(`Clicked ${count} times`));
```

其他的流操作符是filter, delay, debounceTime, take, takeUntil, distinct, distinctUntilChanged 等等。

Values 值

你可以通过可观察对象来转化值

下面的程序可以在每次点击鼠标时获取X坐标位置

纯的JS实现

```
var count = 0;
var rate = 1000;
var lastClick = Date.now() - rate;
var button = document.querySelector('button');
button.addEventListener('click', (event) => {
  if (Date.now() - lastClick >= rate) {
    console.log(++count + event.clientX)
    lastClick = Date.now();
  }
});
```

RxJS实现

```
var button = document.querySelector('button');
Rx.Observable.fromEvent(button, 'click')
  .throttleTime(1000)
  .map(event => event.clientX)
  .scan((count, clientX) => count + clientX, 0)
  .subscribe(count => console.log(count));
```

其他的值生产者还有 `pluck`, `pairwise`, `sample` 等等.

observable可观察对象

可观察对象以惰性的方式推送多值的集合。

	Single单值	Multiple多值
pull拉	Function	Iterator
push推	Promise	Observable

下面的例子是一个推送1,2,3,4数值的可观察对象，一旦它被订阅1,2,3,就会被推送，4则会在订阅发生一秒之后被推送，紧接着完成推送。

```
var observable = Rx.Observable.create(function (observer) {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  setTimeout(() => {
    observer.next(4);
    observer.complete();
  }, 1000);
});
```

调用可观察对象然后得到它所推送的值，我们订阅它，如下

```
console.log('just before subscribe');
observable.subscribe({
  next: x => console.log('got value ' + x),
  error: err => console.error('something wrong occurred: ' + err),
  complete: () => console.log('done'),
});
console.log('just after subscribe');
```

结果如下

```
just before subscribe
got value 1
got value 2
got value 3
just after subscribe
got value 4
done
```

Pull拉取 VS Push推送

拉和推是数据生产者和数据的消费者两种不同的交流协议(方式)

什么是"**Pull**拉"?在"拉"体系中,数据的消费者决定何时从数据生产者那里获取数据,而生产者自身并不会意识到什么时候数据将会被发送给消费者。

每一个JS函数都是一个“拉”体系,函数是数据的生产者,调用函数的代码通过“拉出”一个单一的返回值来消费该数据(`return` 语句)。

ES6介绍了[iterator迭代器](#)和[Generator生成器](#)——另一中“拉”体系,调用 `iterator.next()` 的代码是消费者,可从中拉取多个值。

	producer	Consumer
pull 拉	Passive(被动的一方):被请求的时候产生数据	Active(起主导的一方):决定何时请求数据
push 推	Active:按自己的节奏生产数据	Passive:对接收的数据做出反应(处理接收到的数据)

什么是"**Push**推"?在推体系中,数据的生产者决定何时发送数据给消费者,消费者不会在接收数据之前意识到它将要接收这个数据。

[Promise\(承诺\)](#)是当今JS中最常见的Push推体系,一个Promise(数据的生产者)发送一个resolved value(成功状态的值)来注册一个回调(数据消费者),但是不同于函数的地方的是:Promise决定着何时数据才被推送至这个回调函数。

RxJS引入了Observables(可观察对象),一个全新的"推体系"。一个可观察对象是一个产生多值的生产者,并"推送给"Observer(观察者)。

- Function:只在调用时惰性的计算后同步地返回一个值
- Generator(生成器):惰性计算,在迭代时同步的返回零到无限个值(如果有可能)

的话)

- **Promise**是一个可能(也可能不)返回一个单值的计算。
- **Observable**是一个从它被调用开始，可异步或者同步的返回零到多个值的惰性执行运算。

可观察对象——作为更一般化的函数

与常见的主张相悖的是，可观察对象不像**EventEmitters**(事件驱动)，也不象**Promises**因为它可以返回多个值。可观察对象可能会在某些情况下有点像**EventEmitters**(事件驱动)，也即是当它们使用**Subjects**被多播时，但是大多数情况下，并不像**EventEmitters**。

可观察对象像一个零参的函数，但是允许返回多个值使得其更加的一般化。

思考下面的程序

```
function foo() {  
  console.log('Hello');  
  return 42;  
}  
  
var x = foo.call(); // same as foo()  
console.log(x);  
var y = foo.call(); // same as foo()  
console.log(y);
```

我们可以看到这样的输出。

```
"Hello"  
42  
"Hello"  
42
```

使用**Observables**得到同样的结果

```
var foo=Rx.Observable.create(function(observer){
  console.log('Hello');
  observer.next(42);
});

foo.subscribe(function(x){
  console.log(x);
});
foo.subscribe(function (y){
  console.log(y);
});
```

得到同样的输出

```
"Hello" 42 "Hello" 42
```

这种情况源自于函数和可观察对象均是惰性计算。如果你不'call(调用)'函数,console.log('Hello')将不会发生。可观察对象同样如此,如果你不"调用"(使用subscribe, 订阅), console.log('Hello')也将不会发生。此外,'calling'或者'subscribing'是一个独立的操作:两次函数调用触发两个独立副作用,两次订阅触发两个独立的副作用。相反的,EventEmitters:共享副作用并且不管订阅者的存在而去执行。

订阅一个可观察对象类似于调用一个函数。

一些人认为可观察对象是异步的。这并不确切,如果你用一些log语句包围在订阅程序的前后:

```
console.log('before');
console.log(foo.call());
console.log('after');
```

你可以得到如下结果

"

```
before"  
"Hello"  
42  
"after"
```

类似的，使用可观察对象：

```
console.log('before');  
foo.subscribe(function (x) {  
  console.log(x);  
});  
console.log('after');
```

输出如下：

```
"before"  
"Hello"  
42  
"after"
```

以上可以显示对foo的订阅是完全同步的，就像调用一个函数。

可观察对象可以以同步或者异步的方式发送多个值。

好吧，调转方向，说一说可观察对象和函数的不同之处。可观察对象可以随时"return"多个值。然而函数却做不到，你不能够使得如下的情况发生：

```
function foo() {  
  console.log('Hello');  
  return 42;  
  return 100; // dead code. will never happen  
}
```

函数仅仅可以返回一个值，然而，不要惊讶，可观察对象却可以做到这些。

```
var foo = Rx.Observable.create(function (observer) {  
  console.log('Hello');  
  observer.next(42);  
  observer.next(100); // "return" another value  
  observer.next(200); // "return" yet another  
});  
  
console.log('before');  
foo.subscribe(function (x) {  
  console.log(x);  
});  
console.log('after');
```

同步输出:

```
"before"  
"Hello"  
42  
100  
200  
"after"
```

当然，你也可以以异步的方式返回值。

```
var foo = Rx.Observable.create(function (observer) {
  console.log('Hello');
  observer.next(42);
  observer.next(100);
  observer.next(200);
  setTimeout(() => {
    observer.next(300); // happens asynchronously
  }, 1000);
});

console.log('before');
foo.subscribe(function (x) {
  console.log(x);
});
console.log('after');
```

输出

```
"before"
"Hello"
42
100
200
"after"
300
```

总结:

- `fun.call()`意味着同步的给我一个值
- `observable.subscribe()`意味着给我任意多个值，同步也好异步也罢。

剖析可观察对象

使用`Rx.Observable.create`或者一个能产生可观察对象的操作符来创建一个可观察对象，使用一个观察者订阅它，执行然后给观察者发送`next/error/complete`通知。他们的执行可能会被`disposed`(处理)。这四个方面均被编码进可观察对象的实例中。但是其中的某些方面和其他的类型有关，如`Observer`和`Subscription`

核心的可观察对象概念(个人觉得这一块还是不翻译的好，原汁原味)

- Creating Observables
- Subscribing to Observables
- Executing the Observable
- Disposing Observables

Creating Observables

`Rx.Observable.create` 是可观察对象构造函数的别名，它接受一个参数:the subscribe function。

下面的例子创造一每秒向观察者发射一个字符串"hi"的可观察对象。

```
var observable = Rx.Observable.create(function subscribe(observer) {
  var id = setInterval(() => {
    observer.next('hi')
  }, 1000);
});
```

可观察对象可以使用`create`创建，但是通常我们使用被称为creation operators, 像`of`,`from`,`interval`等。

在上面的例子中，`subscribe`(订阅)函数是订阅可观察对象最重要的部分。接下来让我们看下订阅的含义是什么。

订阅可观察对象

观察对象可以像下面的例子那样被订阅:

```
observable.subscribe(x => console.log(x));
```

`observable.subscribe`和`Observable.create(function subscribe(observer){})`的`subscribe`回调函数有着同样的名字并不是因缘巧合。在RxJS中，他们是不同的，但是为了更使用的目的，你可以认为他们在概念上是等价的。

这显示出对于同一个可观察对象进行订阅的多个观察者之间的回调函数是不共享信息的。当使用`observer`调用`observable.subscribe`时，`Observable.create(function subscribe(observer){})`中的`subscribe`函数为既定的`observer`运行。每次调用`observable.subscribe`为给定的观察者触发它自身独立的设定程序。

订阅一个可观察对象就像调用一个函数，在数据将被发送的地方提供回调。

完全不同于诸如`addEventListener/removeEventListener`事件句柄API.使用`observable.subscribe`,给定的观察者并没有作为一个监听者被注册。可观察对象甚至也不保存有哪些观察者。

订阅是启动可观察对象执行和发送值或者事件给观察者的简单方式。

执行可观察对象

在`Observable.create(function(observer){...})`中的代码，表示了一个可观察对象的执行，一个仅在观察者订阅的时候发生的惰性计算。执行随着时间产生多个值，以同步或者异步的方式。

下面是可观察对象执行可以发送的三种类型的值

- **"Next"**: 发送一个数字/字符串/对象等值。
- **"Error"**: 发送一个JS错误或者异常。
- **"Complete"** 不发送值。

Next通知是最重要且最常见的类型:它们代表发送给观察者的确切数据，**Error**和**Complete**通知可能仅在可观察对象执行期间仅发生一次，但仅会执行二者之中的一个。

这些约束条件能够在可观察对象语法以类似于正则表达式的方式表达的更清晰:

```
next*(error|complete)?
```

一个可观察对象的执行期间，零个到无穷多个`next`通知被发送。如果**Error**或者**Complete**通知一旦被发送，此后将不再发送任何值。

下面这个例子，可观察对象执行然后发送三个`next`通知，然后**completes**:

```
var observable = Rx.Observable.create(function subscribe(observer) {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();
});
```

可观察对象严格的坚守这个契约，所以，下面的代码将不会发送包含数值4的next通知

```
var observable = Rx.Observable.create(function subscribe(observer) {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();
  observer.next(4); // Is not delivered because it would violate the contract
});
```

不失为一个好方式的是，使用try/catch语句包裹通知语句，如果捕获了异常将会发送一个错误通知。

```
var observable = Rx.Observable.create(function subscribe(observer) {
  try {
    observer.next(1);
    observer.next(2);
    observer.next(3);
    observer.complete();
  } catch (err) {
    observer.error(err); // delivers an error if it caught one
  }
});
```

处理可观察对象的执行

由于可观察对象的执行可能是无限的(不停地next)，而对于观察者来说却往往希望在有限的时间内终止执行，因此我们需要一个API来取消执行。因为每一次的执行仅仅服务于一个观察者，一旦观察者听得接收数据，它就不得不通过一个方式去终止执行，从而避免浪费大量的计算性能和内存资源。

当`observable.subscribe`被调用，观察者将专注于最新被创建的可观察对象的执行，并且这个调用返回一个对象:the `Subscription`

```
var subscription = observable.subscribe(x => console.log(x));
```

the `Subscription`(订阅)表示正在进行的执行，这里有一个用于终止执行的小型API。阅读更多关于`Subscription`的信息。使用`subscription.unsubscribe()`你可以取消正在进行的执行:

```
var observable = Rx.Observable.from([10, 20, 30]);
var subscription = observable.subscribe(x => console.log(x));
// Later:
subscription.unsubscribe();
```

在你订阅了之后，你将会得到一个`Subscription`对象，它表示正在进行的执行。大胆的去使用`unsubscribe()`去终止执行吧。

当我们使用`create()`创建可观察对象，每一个可观察对象必须确定怎样去处理该执行的资源。你可以通过在`subscribe`函数内返回的`subscription`调用`unsubscribe`函数做到这一点。

作为示例，下面是怎样去清除一个`setInterval`间隔执行

```
var observable = Rx.Observable.create(function subscribe(observer) {
  // Keep track of the interval resource
  var intervalID = setInterval(() => {
    observer.next('hi');
  }, 1000);

  // Provide a way of canceling and disposing the interval resource

  return function unsubscribe() {
    clearInterval(intervalID);
  };
});
```

就像`observable.subscribe`类似于`Observable.create(function subscribe(){...})`,我们从`subscribe`函数中返回的`unsubscribe`函数在概念上等价于`subscription.unsubscribe`。事实上,如果我们移除环绕于这些概念之外的`ReactiveX`类型,也就只剩下更加直观的`JavaScript`。

```
function subscribe(observer) {
  var intervalID = setInterval(() => {
    observer.next('hi');
  }, 1000);

  return function unsubscribe() {
    clearInterval(intervalID);
  };
}

var unsubscribe = subscribe({next: (x) => console.log(x)});

// Later:
unsubscribe(); // dispose the resources
```

我们使用诸如可观察对象,观察者,和订阅等`Rx`类型的原因是能够兼顾安全性(例如`Observable,Observer,Subscription`)和操作符的可组合性。

observer观察者

什么是观察者？观察者是可观察对象所发送数据的消费者，观察者简单而言是一组回调函数，分别对应一种被可观察对象发送的通知的类型:next, error和complete。下面是一个典型的观察者对象的例子:

```
var observer={
  next:x=>console.log('Observer got a next value: ' + x),
  error: err => console.error('Observer got an error: ' + err),
  complete: () => console.log('Observer got a complete notification')
}
```

去使用观察者，需要订阅可观察对象:

```
observable.subscribe(observer)
```

观察者不过是三个回调函数组成的对象，每个回调函数分别对应可观察对象的通知类型。

RxJS中的观察者是可选的，如果你不提供某个回调函数，可观察对象的执行仍然会照常发生，当然某个类型的通知将不会发生，因为在观察者对象中没有对应于他们的回调函数。

下面的例子中是一个没有complete回调的观察者对象:

```
var observer={
  next:x=>console.log('Observer got a next value: ' + x),
  error: err => console.error('Observer got an error: ' + err),
}
```

当订阅一个可观察对象，你可能仅仅提供回调来作为参数就够了，并不需要完整的观察者对象，作为示例:

```
observable.subscribe(x => console.log('Observer got a next value  
: ' + x));
```

在`observable.subscribe`内部，它将使用第一个回调参数作为`next`的处理句柄创建一个观察者对象。也可以通过将三个函数作为参数提供三种回调:

```
observable.subscribe(  
x => console.log('Observer got a next value: ' + x),  
err => console.error('Observer got an error: ' + err),  
() => console.log('Observer got a complete notification')  
);
```

Subscription 订阅

什么是订阅？订阅是一个表示一次性资源的对象，通常是一个可观察对象的执行。订阅对象有一个重要的方法:unsubscribe，该方法不需要参数，仅仅去废弃掉可观察对象所持有的资源。在以往的RxJS的版本中，"Subscription订阅"被称为"Disposable"。

```
var observable = Rx.Observable.interval(1000);
var subscription = observable.subscribe(x => console.log(x));
// Later:
// This cancels the ongoing Observable execution which
// was started by calling subscribe with an Observer.
subscription.unsubscribe();
```

订阅对象有一个unsubscribe()方法用来释放资源或者取消可观察对象的执行。

订阅对象也可以被放置在一起，因此对一个订阅对象的unsubscribe()进行调用，可以对多个订阅进行取消。做法是:把一个订阅"加"进另一个订阅。

```
var observable1 = Rx.Observable.interval(400);
var observable2 = Rx.Observable.interval(300);

var subscription = observable1.subscribe(x => console.log('first
: ' + x));
var childSubscription = observable2.subscribe(x => console.log('
second: ' + x));

subscription.add(childSubscription);

setTimeout(() => {
// Unsubscribes BOTH subscription and childSubscription
subscription.unsubscribe();
}, 1000);
```

执行之后，我们可以在控制台得到:


```
second: 0  
first: 0  
second: 1  
first: 1  
second: 2
```

订阅也有一个`remove(otherSubscription)`方法,用于解除被`add`添加的子订阅。

Subject主题

什么是Subject？Subject是允许值被多播到多个观察者的一种特殊的Observable。然而纯粹的可观察对象是单播的(每一个订阅的观察者拥有单独的可观察对象的执行)。

Subject就是一个可观察对象，只不过可以被多播至多个观察者。同时Subject也类似于EventEmitter:维护着众多事件监听器的注册表。

每一个**Subject**都是一个**observable**可观察对象，给定一个Subject后，你可以订阅它，提供的观察者将会正常的开始接收值。从观察者的角度来看，它不能判断一个可观察对象的执行时来自于单播的Observable还是来自于一个Subject.

在Subject的内部，**subscribe**并不调用一个新的发送值得执行。它仅仅在观察者注册表中注册给定的观察者，类似于其他库或者语言中的**addListener**的工作方式。

每一个**Subject**都是一个**Observer**观察者对象。它是一个拥有**next()/error()/complete()**方法的对象。要想Subject提供一个新的值，只需调用**next()**，它将会被多播至用来监听Subject的观察者。

下面的例子，Subject有两个观察者，

```
var subject = new Rx.Subject();

subject.subscribe({
  next: (v) => console.log('observerA: ' + v)
});
subject.subscribe({
  next: (v) => console.log('observerB: ' + v)
});

subject.next(1);
subject.next(2);
```

输出如下:

```
observerA: 1
observerB: 1
observerA: 2
observerB: 2
```

由于Subject也是一个观察者，这就意味着你可以提供一个Subject当做observable.subscribe()的参数，如下：

```
var subject = new Rx.Subject();

subject.subscribe({
  next: (v) => console.log('observerA: ' + v)
});
subject.subscribe({
  next: (v) => console.log('observerB: ' + v)
});

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

observable.subscribe(subject); // You can subscribe providing a
Subject
```

输出如下：

```
observerA: 1
observerB: 1
observerA: 2
observerB: 2
observerA: 3
observerB: 3
```

用上面的方式，我们本质上是通过将一个单播的可观察对象转化为多播。这个演示了Subjects是如何将可观察对象执行分享给多个观察者的。(译者注:注意观察上面的两个subject.subscribe()中传入的两个观察者对象)

多播的可观察对象

一个"多播的可观察对象"通过具有多个订阅者的Subject对象传递通知。然而一个单纯的"单播可观察对象"仅仅给一个单一的观察者发送通知。

一个多播的可观察对象使用一个Subject，使得多个观察者可以看到同一个可观察对象的执行。

```
var source=Rx.Observable.from([1,2,3]);
var subject=new Rx.Subject();
var multicasted=source.multicast(subject);
multicasted.subscribe({
  next:(v)=>console.log('observerA:' +v);
});
multicasted.subscribe({
  next: (v) => console.log('observerB: ' + v)
});
multicasted.connect();
```

multicast方法返回一个看起来很像普通的可观察对象的可观察对象，但是在订阅时却有着和Subject一样的行为，multicast返回一个**ConnectableObservable**，它只是一个具有connect（）方法的Observable。

connect()方法对于在决定何时开始分享可观察对象的执行是非常重要的。因为connect（）在source下面有source.subscribe（subject），connect（）返回一个Subscription，你可以取消订阅，以取消共享的Observable执行。

reference counting 引用计数

调用connect()手动的处理Subscription是很麻烦的，我们想要在第一个观察者到达时自动的连接，并且在最后一个观察者取消订阅后自动的取消可观察对象的执行。考虑下面的例子，其中按照此列表概述的方式进行订阅。

1. 第一个观察者订阅多播可观察对象
2. 多播可观察对象被连接
3. next value 0 被发送给第一个观察者
4. 第二个观察者订阅多播可观察对象
5. next value 1 被发送给第一个观察者

6. next value 1 被发送给第二个观察者
7. 第一个观察者取消订阅多播可观察对象
8. next value 1 被发送给第二个观察者
9. 第二个观察者取消订阅多播可观察对象
10. 到多播可观察对象的连接被取消

我们通过显示的调用`connect()`来实现，如下：

```
var source = Rx.Observable.interval(500);
var subject = new Rx.Subject();
var multicasted = source.multicast(subject);
var subscription1, subscription2, subscriptionConnect;
subscription1 = multicasted.subscribe({
  next: (v) => console.log('observerA: ' + v)
});
// We should call `connect()` here, because the first
// subscriber to `multicasted` is interested in consuming values
subscriptionConnect = multicasted.connect();
setTimeout(() => {
  subscription2 = multicasted.subscribe({
    next: (v) => console.log('observerB: ' + v)
  });
}, 600);
setTimeout(() => {
  subscription1.unsubscribe();
}, 1200);
// We should unsubscribe the shared Observable execution here,
// because `multicasted` would have no more subscribers after this
setTimeout(() => {
  subscription2.unsubscribe();
  subscriptionConnect.unsubscribe(); // for the shared Observable execution
}, 2000);
```

如果我们希望避免显式的调用`connect()`，我们可以使用`ConnectableObservable`对象的`refCount()`方法(引用计数)，它返回一个追踪它自身有多少个订阅者的可观察对象。当订阅者的数量从0增加到1，它将会为我们调用`connect()`，这将开始分享可观察对象的执行。在当且仅在订阅者的数量降低到0的时候它将会完全取消订阅，停掉更进一步的执行。

`refCount`使得多播可观察对象在其第一个观察者开始订阅时自动的开始执行，在其最后一个订阅者取消的时候终止执行

下面的例子

```
var source = Rx.Observable.interval(500);
var subject = new Rx.Subject();
var refCounted = source.multicast(subject).refCount();
var subscription1, subscription2, subscriptionConnect;
// This calls `connect()`, because
// it is the first subscriber to `refCounted`
console.log('observerA subscribed');
subscription1 = refCounted.subscribe({
  next: (v) => console.log('observerA: ' + v)
});

setTimeout(() => {
  console.log('observerB subscribed');
  subscription2 = refCounted.subscribe({
    next: (v) => console.log('observerB: ' + v)
  });
}, 600);

setTimeout(() => {
  console.log('observerA unsubscribed');
  subscription1.unsubscribe();
}, 1200);

// This is when the shared Observable execution will stop, because
// `refCounted` would have no more subscribers after this
setTimeout(() => {
  console.log('observerB unsubscribed');
  subscription2.unsubscribe();
}, 2000);
```

上面的例子得到如下的输出:

```
observerA subscribed
observerA: 0
observerB subscribed
observerA: 1
observerB: 1
observerA unsubscribed
observerB: 2
observerB unsubscribed
```

引用计数方法`refCount()`仅存在于`ConnectableObservable`，并且它返回一个`Observable`，而不是另一个`ConnectableObservable`。

BehaviorSubject

`Subjects`的一个变体是`BehaviorSubject`，其有"当前值"的概念。它储存着要发射给消费者的最新的值。无论何时一个新的观察者订阅它，都会立即接受到这个来自`BehaviorSubject`的"当前值"。

`BehaviorSubject`对于表示"随时间的值"是很有用的。举个例子，人的生日的事件流是一个`Subject`，然而人的年龄的流是一个`BehaviorSubject`。

在下面的例子中，`BehaviorSubject`被数值0初始化，第一个观察者将会在订阅的时候收到这个值。第二个观察者接收数值2，即使它是在数值2被发送之后订阅的。


```
var subject = new Rx.BehaviorSubject(0); // 0 is the initial value

subject.subscribe({
  next: (v) => console.log('observerA: ' + v)
});

subject.next(1);
subject.next(2);

subject.subscribe({
  next: (v) => console.log('observerB: ' + v)
});

subject.next(3);
```

输出如下:

```
observerA: 0
observerA: 1
observerA: 2
observerB: 2
observerA: 3
observerB: 3
```

ReplaySubject

一个ReplaySubject类似于一个BehaviorSubject，因为它可以发送一个过去的值(old values)给一个新的订阅者，但是它也可以记录可观察对象的一部分执行。

一个ReplaySubject 从一个可观察对象的执行中记录多个值，并且可以重新发送给新的订阅者。

当创建一个ReplaySubject，你可指定有多少值需要重发。

```
var subject = new Rx.ReplaySubject(3); // buffer 3 values for new subscribers ，注:缓存了三个值。

subject.subscribe({
  next: (v) => console.log('observerA: ' + v)
});

subject.next(1);
subject.next(2);
subject.next(3);
subject.next(4);

subject.subscribe({
  next: (v) => console.log('observerB: ' + v)
});

subject.next(5);
```

输出如下

```
observerA: 1
observerA: 2
observerA: 3
observerA: 4
observerB: 2
observerB: 3
observerB: 4
observerA: 5
observerB: 5
```

除了缓存值得个数之外，你也可以指定一个以毫秒为单位的时间，来决定过去多久出现的值可以被重发。在下面的例子中指定一百个缓存值，但是时间参数仅为500ms。

```
var subject = new Rx.ReplaySubject(100, 500 /* windowTime */);

subject.subscribe({
  next: (v) => console.log('observerA: ' + v)
});

var i = 1;
setInterval(() => subject.next(i++), 200);

setTimeout(() => {
  subject.subscribe({
    next: (v) => console.log('observerB: ' + v)
  });
}, 1000);
```

输出如下:

```
observerA: 1
observerA: 2
observerA: 3
observerA: 4
observerA: 5
observerB: 3
observerB: 4
observerB: 5
observerA: 6
observerB: 6
...
```

AsyncSubject

`AsyncSubject`是另一个变体，它只发送给观察者可观察对象执行的最新值，并且仅在执行结束时。

```
var subject = new Rx.AsyncSubject();

subject.subscribe({
  next: (v) => console.log('observerA: ' + v)
});

subject.next(1);
subject.next(2);
subject.next(3);
subject.next(4);

subject.subscribe({
  next: (v) => console.log('observerB: ' + v)
});

subject.next(5);
subject.complete();
```

输出:

```
observerA: 5
observerB: 5
```

AsyncSubject类似于last()操作符,因为它为了发送单一值而等待complete通知。

由于操作符数量过多，加上用法介绍，会导致译文过于冗杂，而且官方文档的点击需要的类型层层的选择也很麻烦。为了查阅方便，将会在一个单独的**RxJS**类模块进行整理介绍，类上定义的每一个操作符都会单独介绍并附上源码，知其然更要知其所以然。如需查阅，请直接前往类模块。此文仅涵盖操作符的基本介绍，以及一个操作符是怎样产生的。

Operators操作符

RxJS如此强大的原因正是源自于它的操作符，即便可观察对象扮演着根基的角色。操作符使得复杂的异步代码轻松的以声明的方式容易地组成。

什么是操作符？

操作符是可观察对象上定义的方法，例如`.map(...)`、`.filter(...)`、`.merge(...)`，等等。当他们被调用，并不会去改变当前存在的可观察对象实例。相反，他们返回一个新的可观察对象，而且新返回的`subscription`订阅对象逻辑上基于调用他们的我观察对象。

每一个操作符都是基于当前可观察对象创建一个新的可观察对象的函数。这是一个单纯无害的操作:之前的可观察对象仍然保持不变。

一个操作符本质上是一个将某个可观察对象作为输入然后输出另一个可观察对象的纯函数。对输出的新的可观察对象进行订阅的同时也会订阅作为输入的那个可观察对象。下面的例子，我们创建一个自定义的运算符函数，将从作为输入的可观察对象的每个值乘以10.

```
function multiplyByTen(input) {  
  var output = Rx.Observable.create(function(observer){  
    input.subscribe({  
      next: (v) => observer.next(10*v),  
      error: (err) => observer.error(err),  
      complete: () => observer.complete()  
    });  
  });  
  return output;  
}  
var input = Rx.Observable.from([1,2,3,4]);  
var output = multiplyByTen(input);  
output.subscribe(x=>console.log(x));
```

输出如下:

```
10  
20  
30  
40
```

注意一下我们对output的订阅导致了作为输入的可观察对象也被订阅了。我们称这种现象为"操作符的订阅链"。

实例操作符(instance operator)VS静态操作符(static operator)

什么是实例操作符？通常，当引用运算符，我们假设是来自可观察对象实例的运算符。作为示例，如果multiplyByTen一个官方库中的操作符，那它大概是下面的样子:

```
Rx.Observable.prototype.multiplyByTen = function(){  
  var input=this;  
  return Rx.Observable.create(function subscribe(observer){  
    input.subscribe({  
      next: (v) => observer.next(10*v),  
      error: (err) => observer.error(err),  
      complete: () => observer.complete()  
    });  
  });  
}
```

译者注:考虑到可能会有些刚接触的人不太理解。上面的代码是这样的,操作符定义在可观察对象的原型对象上,如果某个可观察对象调用这个方法, `input` 首先取得作为输入的该可观察对象的引用(后面会在闭包里用到),然后该方法返回一个由 `create` 方法创建的可观察对象,也就是作为输出的可观察对象。而 `input.subscribe({...})` 是对 `input` 进行订阅。

实例操作符是使用 `this` 关键词来推演出"输入可观察对象"是神马东东

注意 `input` 并不是作为函数的参数,而是作为 `this` 所指代的那个对象。下面是我们如何使用这个操作符:

```
var observable = Rx.Observable.from([1,2,3,4]).multiplyByTen();  
observable.subscribe(x => console.log(x));
```

什么是静态操作符?不同于实例操作符,静态操作符是直接定义在类上的。一个静态操作符并不在其内部使用 `this`,而是完全依赖于它的参数。

静态操作符是定义在类上的函数,通常被用于从头重新创建一个可观察对象。

最常规的静态操作符是被称作构造操作符(Creation Operators,原谅我如此翻译,总是隐隐觉得很合适~)。不同于将一个输入的可观察对象转换成一个输出的可观察对象,它们仅接收一个非可观察对象作为参数,比如一个数字,然后构造出一个可观察对象。

一个静态操作符的典型例子是 `interval` 函数。它接收一个数字(而不是一个可观察对象)作为输入的参数,然后到一个可观察对象作为输出。

```
var observable = Rx.Observable.interval(1000)//1000毫秒
```

另一个构造操作符是`create`,我们在之前的例子中曾大量使用,一些合并操作符可能是静态的,例如:`merge`,`combineLatest`,`concat`,等等。将这些操作符声明为静态的是有意义的,因为他们接收多个可观察对象作为参数,而不是一个,示例如下:

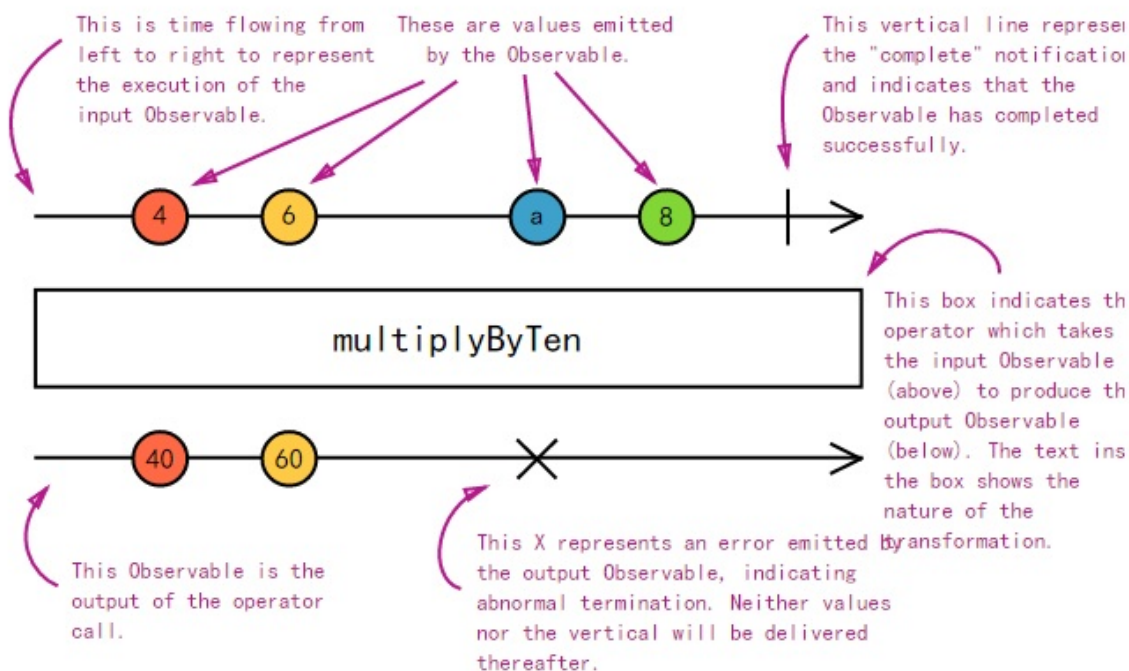
```
var observable1 = Rx.Observable.interval(1000);
var observable2 = Rx.Observable.interval(400);
var merged = Rx.Observable.merge(observable1, observable2);
```

Marble diagrams 弹珠图标

使用文字来形容操作符到底是怎样得工作模式终究还是有那么点力不从心。许多的操作符是和时间线密不可分的,在实际中他们可能需要对值进行延迟、采样、节流、或者抖动的发射。使用图表通常能够对他们的过程进行更好的表述。弹珠图标能够囊括作为输入的可观察对象、操作符、及其参数、作为输出的可观察对象来对操作符的工作方式进行生动的表示。

在一个弹珠图标中,时间流向右侧,操作符描述了值(弹珠)在可观察对象执行时被怎样的发射。

下图你可以看到对整个弹珠图表流程的分析。



贯穿此后的文档，我们将会广泛的使用大理石图标来解释操作的工作过程。他们在其它的内容的表述上也是如此的有用,例如在一个白板或者我们的单元测试中。

Scheduler调度者

什么是调度者？调度者控制着何时启动一个订阅和何时通知被发送。它有三个组件构成：

- 一个调度者是一个数据结构。它知道如何根据优先级或其他标准存储和排列任务。
- 一个调度者是一个执行上下文。它表示何处何时任务被执行(例如: immediately(立即), or in another callback mechanism(回调机制) such as setTimeout or process.nextTick, or the animation frame)
- 一个调度者具有虚拟的时钟。它通过调度器上的getter方法now()提供了“时间”的概念。在特定调度程序上调度的任务将仅仅遵守由该时钟表示的时间。

调度者使得你可以确定可观察对象在什么执行上下文中给观察者发送通知

下面的例子，我们使用常见的的可观察对象，它同步的发送三个数值1/2/3。使用observeOn操作符指定用于传递这些值的异步调度程序。

```
var observable = Rx.Observable.create(function (observer) {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();
})
.observeOn(Rx.Scheduler.async);

console.log('just before subscribe');
observable.subscribe({
  next: x => console.log('got value ' + x),
  error: err => console.error('something wrong occurred: ' + err),
  complete: () => console.log('done'),
});
console.log('just after subscribe');
```

输出

```
just before subscribe
just after subscribe
got value 1
got value 2
got value 3
done
```

注意如何获取值...被发送在"just after subscribe"之后，这是不同于我们目前为止所看到的默认行为。这是因为`observeOn(Rx.Scheduler.async)`在`Observable.create`和最终的`Observer`之间引入了一个代理`Observer`。让我们重命名一些标识符，使这个重要的区别在下面代码中显而易见：

```
var observable = Rx.Observable.create(function (proxyObserver) {
  proxyObserver.next(1);
  proxyObserver.next(2);
  proxyObserver.next(3);
  proxyObserver.complete();
})
.observeOn(Rx.Scheduler.async);

var finalObserver = {
  next: x => console.log('got value ' + x),
  error: err => console.error('something wrong occurred: ' + err),
  complete: () => console.log('done'),
};

console.log('just before subscribe');
observable.subscribe(finalObserver);
console.log('just after subscribe');
```

`proxyObserver`在 `observeOn(Rx.Scheduler.async)`被创建，它的`next`通知函数大约如下：

```
var proxyObserver = {
  next: (val) => {
    Rx.Scheduler.async.schedule(
      (x) => finalObserver.next(x),
      0 /* delay */,
      val /* will be the x for the function above */
    );
  },

  // ...
}
```

scheduler Types

Scheduler	Purpose
null	通过不传递任何调度程序，通知被同步和递归地传递。用于恒定时操作或尾递归操作。

ES6 via npm

```
$ npm install rxjs-es
```

导入核心功能块

```
import Rx from 'rxjs/Rx';  
  
Rx.Observable.of(1,2,3)
```

仅作为补充部分导入你所需要的(通常这种做法有利于减小文件大小)

```
import {Observable} from 'rxjs/Observable';  
import 'rxjs/add/operator/map';  
  
Observable.of(1,2,3).map(x => x + '!!!'); // etc
```

要导入您需要的并且与提出的绑定操作符一起使用它

```
import { Observable } from 'rxjs/Observable';  
import { of } from 'rxjs/observable/of';  
import { map } from 'rxjs/operator/map';  
  
Observable::of(1,2,3)::map(x => x + '!!!'); // etc
```

The basics 基础

Converting to observables 转换为可观察对象

```
//From one or multiple values 一个或多个值->可观察对象
Rx.observable.of('foo', 'bar');
```

```
//From array of values 数组->可观察对象
Rx.Observable.from([1, 2, 3]);
```

```
//From an event 事件->可观察对象
Rx.Observable.fromEvent(document.querySelector('button'), 'click')
);
注:arg1为DOM对象, arg2为事件类型
```

```
//From a promise promise->可观察对象
Rx.Observable.fromPromise(fetch('/users'));
```

```
//From a callback(last argument is a callback) 回调函数->可观察对象

//fs.exists = (path,cb(exists))
var exists = Rx.Observable.bindCallback(fs.exists);
exists('file.txt').subscribe(exists => console.log('Does file exist?', exists));
```

```
// From a callback (last argument is a callback)
// fs.rename = (pathA, pathB, cb(err, result))
var rename = Rx.Observable.bindNodeCallback(fs.rename);
rename('file.txt', 'else.txt').subscribe(() => console.log('Renamed!'));
```

Creating observables 创建可观察对象

外部产生新事件

```
var myObservable = new Rx.Subject();
myObservable.subscribe(value => console.log(value));
myObservable.next('foo');
```

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));
```

你选择哪一个取决于场景.通常当你想封装随时间产生值的功能时, **Observable** 是很好的选择。另一个websocket连接例子, 使用**Subject**可以从任何地方触发事件, 并且你可以连接现存的**observable**到**Subject**。

Controlling the flow 控制流

```
// typing "hello world"
var input = Rx.Observable.fromEvent(document.querySelector('input'), 'keypress');

// Filter out target values less than 3 characters long
input.filter(event => event.target.value.length > 2)
  .subscribe(value => console.log(value)); // "hel"

// Delay the events
input.delay(200)
  .subscribe(value => console.log(value)); // "h" -200ms-> "e" -
200ms-> "l" ...

// Only let through an event every 200 ms
input.throttleTime(200)
  .subscribe(value => console.log(value)); // "h" -200ms-> "w"

// Let through latest event after 200 ms
input.debounceTime(200)
  .subscribe(value => console.log(value)); // "o" -200ms-> "d"

// Stop the stream of events after 3 events
input.take(3)
  .subscribe(value => console.log(value)); // "hel"

// Passes through events until other observable triggers an event
var stopStream = Rx.Observable.fromEvent(document.querySelector(
  'button'), 'click');
input.takeUntil(stopStream)
  .subscribe(value => console.log(value)); // "hello" (click)
```

producing values 生产值


```
// typing "hello world"
var input = Rx.Observable.fromEvent(document.querySelector('input'), 'keypress');

// Pass on a new value
input.map(event => event.target.value)
    .subscribe(value => console.log(value)); // "h"

// Pass on a new value by plucking it
input.pluck('target', 'value')
    .subscribe(value => console.log(value)); // "h"

// Pass the two previous values
input.pluck('target', 'value').pairwise()
    .subscribe(value => console.log(value)); // ["h", "e"]

// Only pass unique values through
input.pluck('target', 'value').distinct()
    .subscribe(value => console.log(value)); // "heho wrd"

// Do not pass repeating values through
input.pluck('target', 'value').distinctUntilChanged()
    .subscribe(value => console.log(value)); // "heho world"
```

Creating applications 创建应用程序

RxJS是一个能够使得你的代码保持更小的错误倾向的强大的工具。这源自于它使用无状态的纯函数。但是应用程序是有状态的，那么我们是如何将RxJS的无状态世界里与应用程序的有状态联系起来的呢？让我们看一个简单的例子:对数值0的状态存储。在每一次点击我们想增加在状态存储中的计数。

```
var button = document.querySelector('button');
Rx.Observable.fromEvent(button, 'click')
  // scan (reduce) to a stream of counts
  .scan(count => count + 1, 0)
  // Set the count on an element each time it changes
  .subscribe(count => document.querySelector('#count').innerHTML
    = count);
```

可以看出，状态是在RxJS中产生的，但是最后一行中改变DOM是一个副作用。

静态操作符和实例操作符

加这一篇主要是为了只接触过JS(es5)的同学，如果有接触过基于类的面向对象语言的，跳过。**static**操作符,静态的，只能通过类本身调用。**instance**实例操作符，每一个实例化的类，既每一个对象都可以继承下来并调用。如果还想了解更多，可以去了解一门基于类的面向对象的语言。此外，每个操作符都有基本语法，如果看不明白，自行查阅TS教程里的类型注解。文档首页有TS中文网链接。

由于官方文档对于操作符的例子过少，我从其他资料上获取的范例会以 F-eg 示出，取自文档中的例子以 eg 示出；

bindCallback()

- 语法:

```
public static bindCallback(func: function, selector: function, scheduler: Scheduler): function(...params: *): Observable
```

- 功能: 将一个回调函数API转化为一个能返回一个Observable的函数;

实际上这不是一个严格意义上的操作符，因为无论它的输入还是输出都不是Observable。在输入函数的参数中，最后一个参数为一个回调函数，bindCallback输出的函数接收和inputfn一样的参数(除去最后的callback)，当输出函数被调用时，它将返回一个可观察对象。

```
// 例子
// Convert jQuery's getJSON to an Observable API
// Suppose we have jQuery.getJSON('/my/url', callback)
var getJSONAsObservable = Rx.Observable.bindCallback(jQuery.getJSON);
var result = getJSONAsObservable('/my/url');
result.subscribe(x => console.log(x), e => console.error(e));
```

bindNodeCallback

- 语法:

```
public static bindNodeCallback(func: function, selector: function  
, scheduler: Scheduler): function(...params: *): Observable
```

- 功能: 将一个NodeJS风格的回调函数API转化为一个能返回可观察对象的函数;

基本等同于bindCallback,不同的是作为输入函数的参数的回调函数要有error参数: callback(err,result).

bindNodeCallback也不是一个真正意义上的操作符,因为它的输入和输出不是Observable。输入是一个带有一些参数的函数func,最后一个参数是func调用完成后的回调函数。回调函数应该遵循Node.js约定,其中回调的第一个参数是错误,而剩余的参数是回调结果。bindNodeCallback的输出是一个函数,它使用与func相同的参数,但是除去最后一个(回调)。当使用参数调用输出函数时,它将返回一个Observable,其中结果将被传递。

eg :

```
import * as fs from 'fs';  
var readFileAsObservable = Rx.Observable.bindNodeCallback(fs.readFile);  
var result = readFileAsObservable('./roadNames.txt', 'utf8');  
result.subscribe(x => console.log(x), e => console.error(e));
```

combineLatest

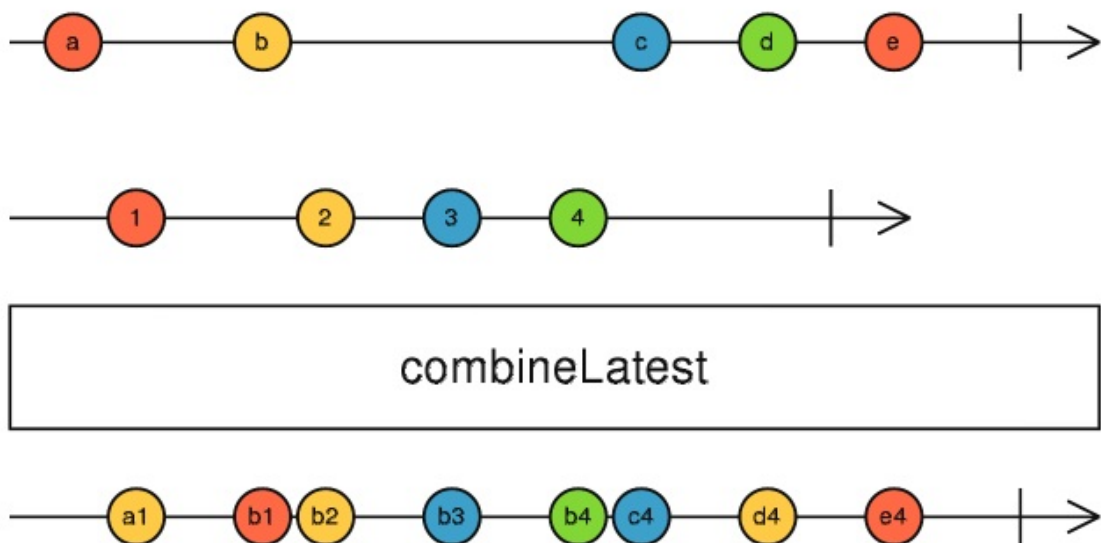
- 语法：

```
public static combineLatest(observable1: Observable, observable2
: Observable, project: function, scheduler: Scheduler): Observab
le
```

- 功能:

组合多个Observable产生一个新的Observable，其发射的值根据其每个输入Observable的最新值计算。

无论何时作为输入的Observable发出一个值，它取所有输入的最新值作为它的发射值。



eg:

```
// Dynamically calculate the Body-Mass Index from an Observable
of weight and one for height
var weight = Rx.Observable.of(70, 72, 76, 79, 75);
var height = Rx.Observable.of(1.76, 1.77, 1.78);
var bmi = Rx.Observable.combineLatest(weight, height, (w, h) =>
w / (h * h));
bmi.subscribe(x => console.log('BMI is ' + x));
```

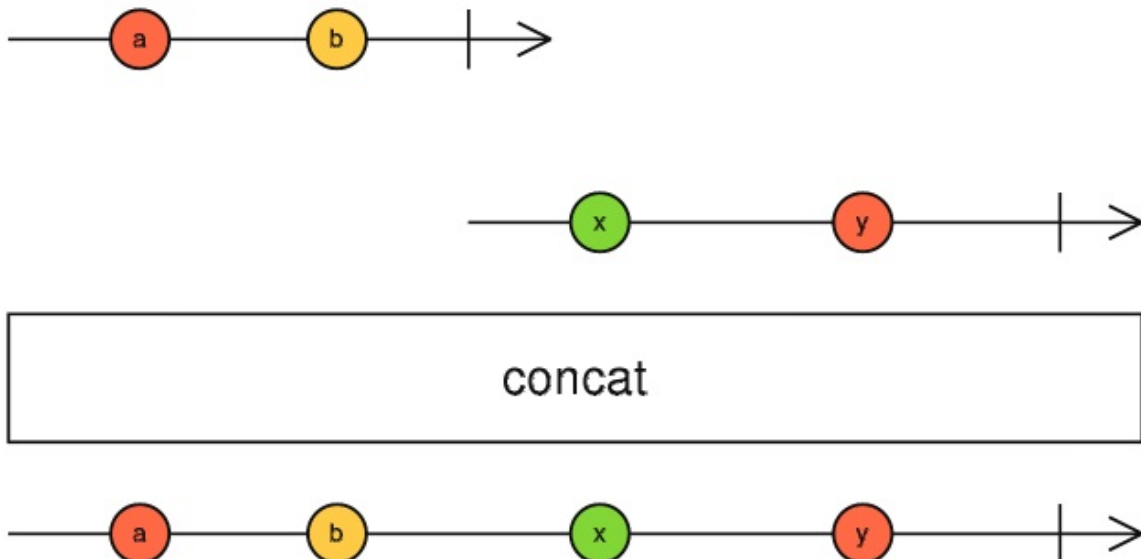

concat

- 语法:

```
public static concat(input1: Observable, input2: Observable, scheduler: Scheduler): Observable
```

- 功能: 产生一个Observable，它取作为参数的Observable发射的每个值，并顺序(基于作为输入的可观察对象的顺序)发出所有值。

连接多个可观察对象，并顺序发出他们的值，一个可观察对象跟在另一个的后面



eg1 :

```
// Concatenate a timer counting from 0 to 3 with a synchronous sequence from 1 to 10
var timer = Rx.Observable.interval(1000).take(4);
var sequence = Rx.Observable.range(1, 10);
var result = Rx.Observable.concat(timer, sequence);
result.subscribe(x => console.log(x));
```

eg2 :


```
var timer1 = Rx.Observable.interval(1000).take(10);  
var timer2 = Rx.Observable.interval(2000).take(6);  
var timer3 = Rx.Observable.interval(500).take(10);  
var result = Rx.Observable.concat(timer1, timer2, timer3);  
result.subscribe(x => console.log(x));
```

create

- 语法:

```
public static create(subscribe: function(subscriber: Subscriber)
: TeardownLogic): Observable
```

- 功能: 创建一个新的Observable，当被订阅时，它将执行指定的函数。

创建一个拥有在订阅函数中给定逻辑的可观察对象

```
create(obs => { obs.next(1); })
```



`create`将`subscribe`函数转换为实际的Observable。这相当于调用Observable构造函数。编写`subscribe`函数，使其作为一个Observable：它应该调用订阅者的`next`，`error`和`complete`方法,遵循Observable约束；良好的Observable必须调用Subscriber的`complete`方法一次或其`error`方法一次，然后再不会调用之后的`next`。

大多数时候，您不需要使用`create`，因为现有的创建操作符（以及实例组合运算符）允许您为大多数用例创建一个Observable。但是，`create`是低级的，并且能够创建任何Observable。

eg:

```
var result = Rx.Observable.create(function (subscriber) {
  subscriber.next(Math.random());
  subscriber.next(Math.random());
  subscriber.next(Math.random());
  subscriber.complete();
});
result.subscribe(x => console.log(x));
```

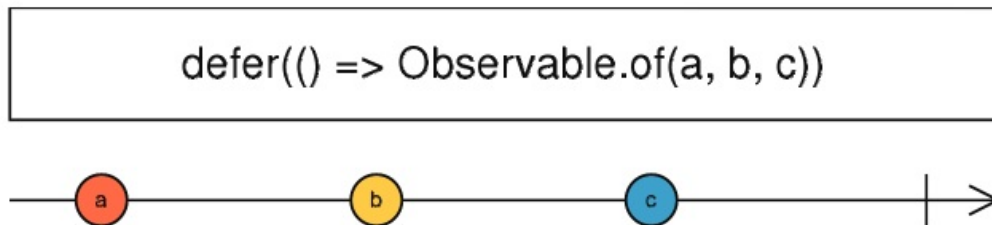

defer

- 语法:

```
public static defer(observableFactory: function(): Observable | Promise): Observable
```

以惰性的方式产生一个Observable,也就是说,当被订阅的时候才会产生。

- 功能: 参数为一个Observable工厂函数,当被订阅时工厂函数被调用产生一个可观察对象。



defer允许您仅在Observer订阅时创建Observable,并为每个Observer创建一个新的Observable。它等待一个Observer订阅它,然后它生成一个Observable,通常有一个Observable工厂函数。它为每个用户分别产生一个Observable,所以虽然每个用户可能认为它们是订阅的同一个Observable,事实上每个订阅者都有自己的单独的Observable。

eg:

```
var clicksOrInterval = Rx.Observable.defer(function () {  
  if (Math.random() > 0.5) {  
    return Rx.Observable.fromEvent(document, 'click');  
  } else {  
    return Rx.Observable.interval(1000);  
  }  
});  
clicksOrInterval.subscribe(x => console.log(x));
```

f-eg :

```
/* Using an observable sequence */
var source = Rx.Observable.defer(() => Rx.Observable.return(42))
;

var subscription = source.subscribe(
  x => console.log(`onNext: ${x}`),
  e => console.log(`onError: ${e}`),
  () => console.log('onCompleted'));

// => onNext: 42
// => onCompleted
```

```
/* Using a promise */
var source = Rx.Observable.defer(() => RSVP.Promise.resolve(42))
;

var subscription = source.subscribe(
  x => console.log(`onNext: ${x}`),
  e => console.log(`onError: ${e}`),
  () => console.log('onCompleted'));

// => onNext: 42
// => onCompleted
```

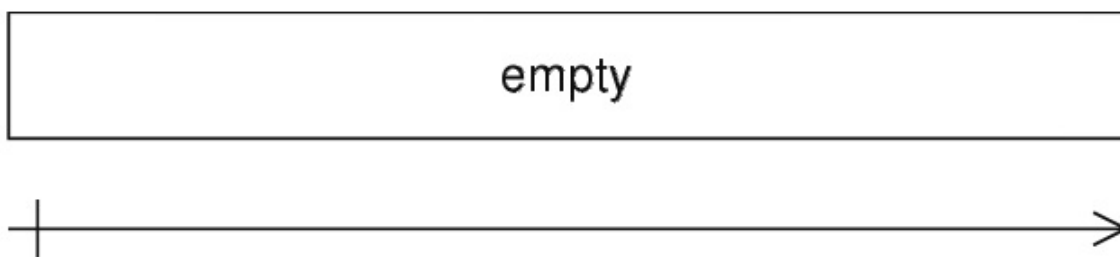
empty

- 语法:

```
public static empty(scheduler: Scheduler): Observable
```

- 功能: 创建一个不发射任何值的Observable,它只会发射一个complete通知。

仅仅发射一个‘complete’,除此之外,不会发射其他值。



该操作符创建一个仅发射‘complete’的通知。通常用于和其他操作符一起组合使用。

eg:

```
//Emit the number 7, then complete.  
var result = Rx.Observable.empty().startWith(7);  
result.subscribe(x => console.log(x));
```

```
//Map and flatten only odd numbers to the sequence 'a', 'b', 'c'  
var interval = Rx.Observable.interval(1000);  
var result = interval.mergeMap(x =>  
    x % 2 === 1 ? Rx.Observable.of('a', 'b', 'c') : Rx.Observable.  
    empty()  
);  
result.subscribe(x => console.log(x));
```

f-eg:

```
var source = Rx.Observable.empty();

var subscription = source.subscribe(
  x => console.log(`onNext: ${x}`),
  e => console.log(`onError: ${e}`),
  () => console.log('onCompleted'));

// => onCompleted
```

forkJoin

- 语法:

```
Rx.Observable.forkJoin(...args [resultSelector])
```

- 功能：并行运行所有可观察序列并收集其最后的元素。

f-eg:

```
/* Using observables and Promises */
var source = Rx.Observable.forkJoin(
  Rx.Observable.return(42),
  Rx.Observable.range(0, 10),
  Rx.Observable.fromArray([1, 2, 3]),
  RSVP.Promise.resolve(56)
);

var subscription = source.subscribe(
  x => console.log(`onNext: ${x}`),
  e => console.log(`onError: ${e}`),
  () => console.log('onCompleted'));

// => Next: [42, 9, 3, 56]
// => Completed
```

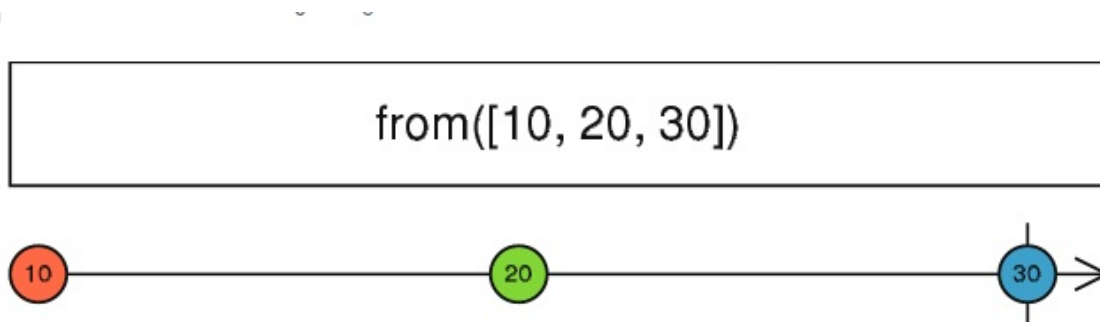

from

- 语法:

```
public static from(ish: ObservableInput<T>, scheduler: Scheduler): Observable<T>
```

- 功能：将一个数组、类数组(字符串也可以)，Promise、可迭代对象，类可观察对象、转化为一个Observable

可将几乎所有的东西转化一个可观察对象



eg:

```
// 数组=>Observable
var array = [10, 20, 30];
var result = Rx.Observable.from(array);
result.subscribe(x => console.log(x));
```

```
// 无限迭代对象=>Observable
function* generateDoubles(seed) {
  var i = seed;
  while (true) {
    yield i;
    i = 2 * i; // double it
  }
}

var iterator = generateDoubles(3);
var result = Rx.Observable.from(iterator).take(10);
result.subscribe(x => console.log(x));
```

f-eg

```
function* generateDoubles(seed) {
  var i = seed;
  while (true) {
    yield i;
    i = 2 * i; // double it
  }
}

var iterator = generateDoubles(3);
var result = Rx.Observable.from(iterator).take(10);
result.subscribe(x => console.log(x));
```

```
var s = new Set(["foo", window]);

Rx.Observable.from(s).subscribe(
  x => console.log(`onNext: ${x}`),
  e => console.log(`onError: ${e}`),
  () => console.log('onCompleted'));

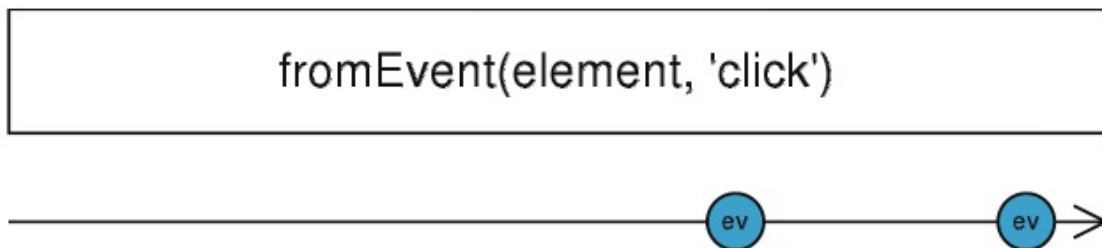
// => onNext: foo
// => onNext: window
// => onCompleted
```


fromEvent

- 语法:

```
Rx.Observable.fromEvent(element, eventName, [selector])
```

- 功能: 将一个元素上的事件转化为一个Observable
- 注意: 使用jQuery, Zepto, Backbone.Marionette, AngularJS和Ember.js的库方法, 并且如果不存在, 则回退到本地绑定。如果您使用AMD, 您可能需要将这些库作为RxJ的依赖关系包括在requirejs配置文件中。当决定使用哪个库时, RxJ将尝试检测它们的存在。



eg :

```
var clicks = Rx.Observable.fromEvent(document, 'click');  
clicks.subscribe(x => console.log(x));
```

f-eg:

```
var input = $('#input');

var source = Rx.Observable.fromEvent(input, 'click');

var subscription = source.subscribe(
  function (x) {
    console.log('Next: Clicked!');
  },
  function (err) {
    console.log('Error: ' + err);
  },
  function () {
    console.log('Completed');
  });

input.trigger('click');

// => Next: Clicked!
```

```
// nodejs

var EventEmitter = require('events').EventEmitter,
    Rx = require('rx');

var eventEmitter = new EventEmitter();

var source = Rx.Observable.fromEvent(
    eventEmitter,
    'data',
    function (args) {
        return { foo: args[0], bar: args[1] };
    });

var subscription = source.subscribe(
    function (x) {
        console.log('Next: foo -' x.foo + ', bar -' + x.bar);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

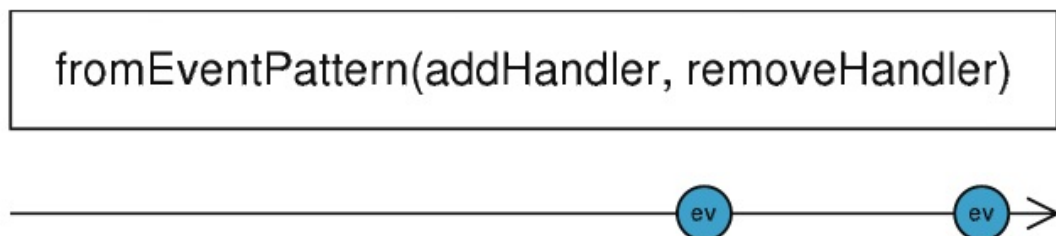
eventEmitter.emit('data', 'baz', 'quux');
// => Next: foo - baz, bar - quux
```

fromEventPattern

- 语法：

```
Rx.Observable.fromEventPattern(addHandler, [removeHandler], [selector])
```

- 功能: 通过使用addHandler和removeHandler函数添加和删除处理程序。当输出Observable被订阅时，addHandler被调用，并且当订阅被取消订阅时调用removeHandler。



eg:

```
function addClickHandler(handler) {
    document.addEventListener('click', handler);
}

function removeClickHandler(handler) {
    document.removeEventListener('click', handler);
}

var clicks = Rx.Observable.fromEventPattern(
    addClickHandler,
    removeClickHandler
);
clicks.subscribe(x => console.log(x));
```

f-eg

```
//jQuery
var input = $('#input');

var source = Rx.Observable.fromEventPattern(
    function add (h) {
        input.bind('click', h);
    },
    function remove (h) {
        input.unbind('click', h);
    }
);

var subscription = source.subscribe(
    function (x) {
        console.log('Next: Clicked!');
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    }
));

input.trigger('click');

// => Next: Clicked!`
```


fromPromise

- 语法:

```
public static fromPromise(promise: Promise<T>, scheduler: Scheduler): Observable<T>
```

- 功能: 转化一个Promise为一个Observable

将ES2015 Promise转换为Observable。如果Promise为成功状态，则Observable会将成功的值作为next发出，然后complete。如果Promise被失败，则输出Observable发出相应的错误。

eg:

```
//convert the Promise returned by Fetch to an Observable  
  
var result = Rx.Observable.fromPromise(fetch('http://myserver.com/'));  
result.subscribe(x => console.log(x), e => console.error(e));
```

f-eg:

```
// Create a promise which resolves 42
var promise = new RSVP.Promise(function (resolve, reject) {
    resolve(42);
});

var source1 = Rx.Observable.fromPromise(promise);

var subscription1 = source1.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: 42
// => Completed
```

```
// Create a promise which rejects with an error
var promise = new RSVP.Promise(function (resolve, reject) {
    reject(new Error('reason'));
});

var source1 = Rx.Observable.fromPromise(promise);

var subscription1 = source1.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

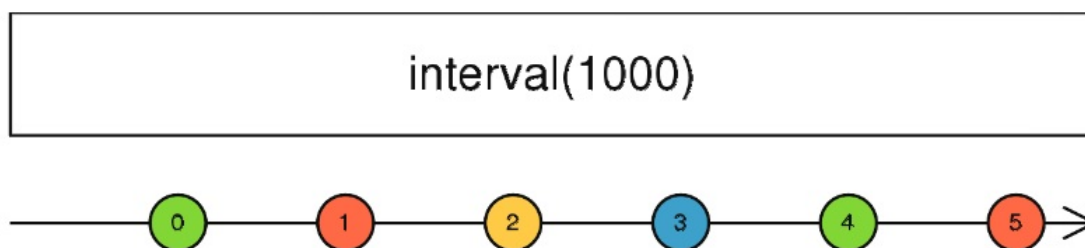
// => Error: Error: reason
```

interval

- 语法:

```
public static interval(period: number, scheduler: Scheduler): Observable
```

- 功能: 返回一个以周期性的、递增的方式发射值的Observable



`interval`返回一个`Observable`，它发出一个递增的无限整数序列。第一个参数为时间间隔。需要注意的是，第一发射不立即发送，而是在第一个周期过去之后发送。第二个参数，默认情况下，`interval`使用异步调度程序提供时间概念，但可以将任何调度程序传递给它。

eg:

```
var numbers = Rx.Observable.interval(1000);  
numbers.subscribe(x => console.log(x));
```

f-eg:

```
var source = Rx.Observable
    .interval(500 /* ms */)
    .timeInterval()
    .take(3);

var subscription = source.subscribe(
    function (x) {
        console.log('Next:', x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: {value: 0, interval: 500}
// => Next: {value: 1, interval: 500}
// => Next: {value: 2, interval: 500}
// => Completed
```

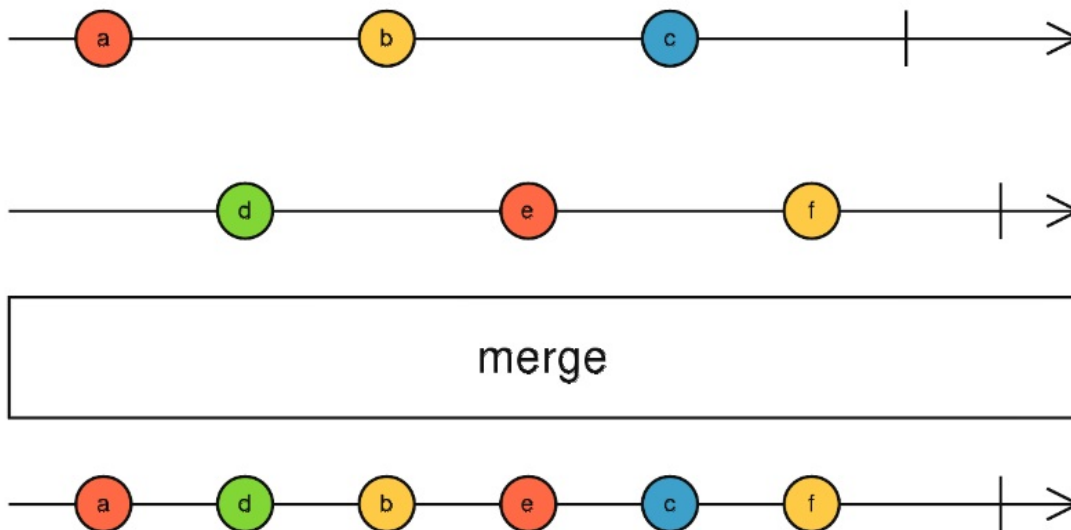
merge

- 语法：

```
public static merge(observable:...Observable, concurrent:number, scheduler:Schedule):Observable
```

- 功能：创建一个发射所有被合并的observable所发射的值。

通过将多个观察者的值合并到一个观察者中进行发射。



输出Observable只有在所有输入Observable完成后才会完成。由输入Observable传递的任何错误将立即在输出Observable上发出。

eg:

```
var clicks = Rx.Observable.fromEvent(document, 'click');  
var timer = Rx.Observable.interval(1000);  
var clicksOrTimer = Rx.Observable.merge(clicks, timer);  
clicksOrTimer.subscribe(x => console.log(x));
```

```
//Merge together 3 Observables, but only 2 run concurrently

var timer1 = Rx.Observable.interval(1000).take(10);
var timer2 = Rx.Observable.interval(2000).take(6);
var timer3 = Rx.Observable.interval(500).take(10);
var concurrent = 2; // the argument
var merged = Rx.Observable.merge(timer1, timer2, timer3, concurrent);
merged.subscribe(x => console.log(x));
```

f-eg:

```
var source1 = Rx.Observable.interval(100)
    .timeInterval()
    .pluck('interval');
var source2 = Rx.Observable.interval(150)
    .timeInterval()
    .pluck('interval');

var source = Rx.Observable.merge(
    source1,
    source2).take(10);

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: 100
// => Next: 150
// => Next: 100
// => Next: 150
// => Next: 100
// => Completed
```


never

- 语法：

```
public static never():Observable
```

- 功能：创建一个不发射任何值的Observable



这个静态操作符对需要创建一个不发射next值、error错误、也不发射complete的简单Observable很有用。它可以用于测试或与其他Observable组合。请不要说，从不发出一个完整的通知，这个Observable保持订阅不被自动处置。订阅需要手动处理。

eg:

```
function info() {  
    console.log('Will not be called');  
}  
var result = Rx.Observable.never().startWith(7);  
result.subscribe(x => console.log(x), info, info);
```

f-eg:

```
// This will never produce a value, hence never calling any of the callbacks
var source = Rx.Observable.never();

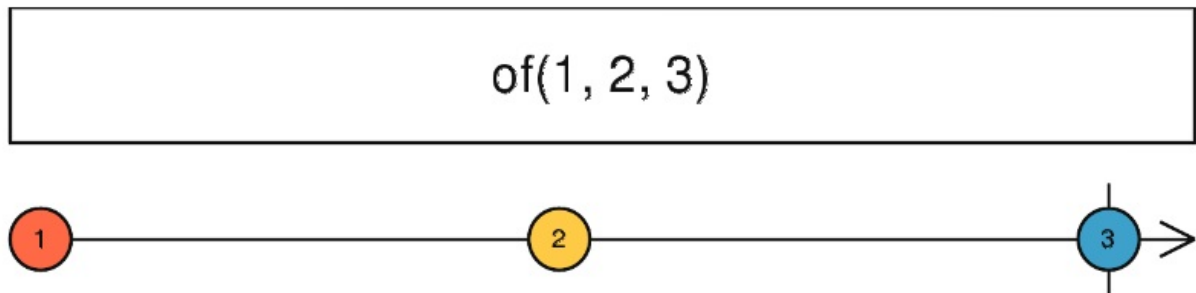
var subscription = source.subscribe(
  function (x) {
    console.log('Next: ' + x);
  },
  function (err) {
    console.log('Error: ' + err);
  },
  function () {
    console.log('Completed');
  });
```

of

语法：

```
public static of (values:...T,scheduler:Schedulr):Observable
```

- 功能：创建一个Observable，发射指定参数的值，一个接一个，最后发出complete。



`of`用于创建一个简单的Observable，只发出给定的参数，然后发出完整的通知。它可以用于与其他Observable组合，如`concat`。默认情况下，它使用一个空的调度程序，这意味着`next`通知是同步发送(看下面的例子)，虽然使用不同的调度程序，可以确定这些通知何时将被交付。

eg:

```
// Emit 1, 2, 3, then 'a', 'b', 'c', then start ticking every se  
cond.  
var numbers = Rx.Observable.of(1, 2, 3);  
var letters = Rx.Observable.of('a', 'b', 'c');  
var interval = Rx.Observable.interval(1000);  
var result = numbers.concat(letters).concat(interval);  
result.subscribe(x => console.log(x));
```

of

1	<u>script.js:35</u>
2	<u>script.js:35</u>
3	<u>script.js:35</u>
A	<u>script.js:35</u>
B	<u>script.js:35</u>
C	<u>script.js:35</u>
0	<u>script.js:35</u>
1	<u>script.js:35</u>
2	<u>script.js:35</u>
3	<u>script.js:35</u>

f-eg:

```
var source = Rx.Observable.of(1,2,3);

var subscription = source.subscribe(
  function (x) {
    console.log('Next: ' + x);
  },
  function (err) {
    console.log('Error: ' + err);
  },
  function () {
    console.log('Completed');
  });

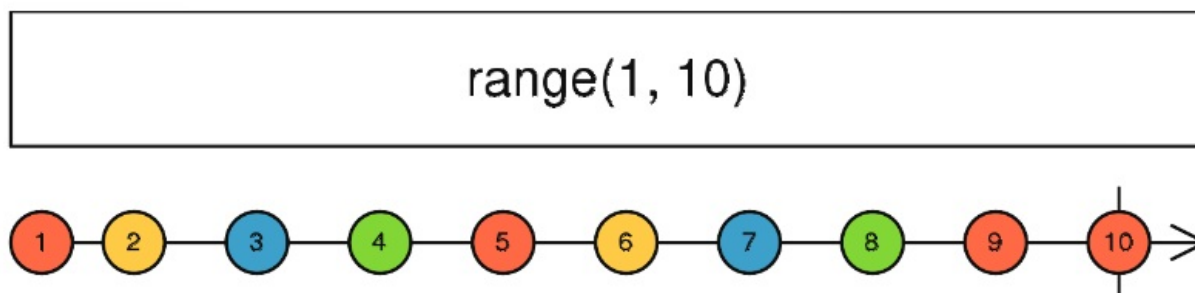
// => Next: 1
// => Next: 2
// => Next: 3
// => Completed
```

range

- 语法：

```
public static range(start:number, count:number, scheduler:Scheduler):Observable
```

- 功能：创建发射一个数字序列的observable



`range`按顺序发出一系列连续整数，参数分别为起点和长度(注意不是终点)。默认情况下，不使用调度程序，只是同步传递通知，但可以使用可选的调度程序来调节这些传递。

eg:

```
var numbers = Rx.Observable.range(1, 10);  
numbers.subscribe(x => console.log(x));
```

range

result:

1	<u>script.js:39</u>
2	<u>script.js:39</u>
3	<u>script.js:39</u>
4	<u>script.js:39</u>
5	<u>script.js:39</u>
6	<u>script.js:39</u>
7	<u>script.js:39</u>
8	<u>script.js:39</u>
9	<u>script.js:39</u>
10	<u>script.js:39</u>

>

f-eg:

```
var source = Rx.Observable.range(0, 3);

var subscription = source.subscribe(
  function (x) {
    console.log('Next: ' + x);
  },
  function (err) {
    console.log('Error: ' + err);
  },
  function () {
    console.log('Completed');
  });

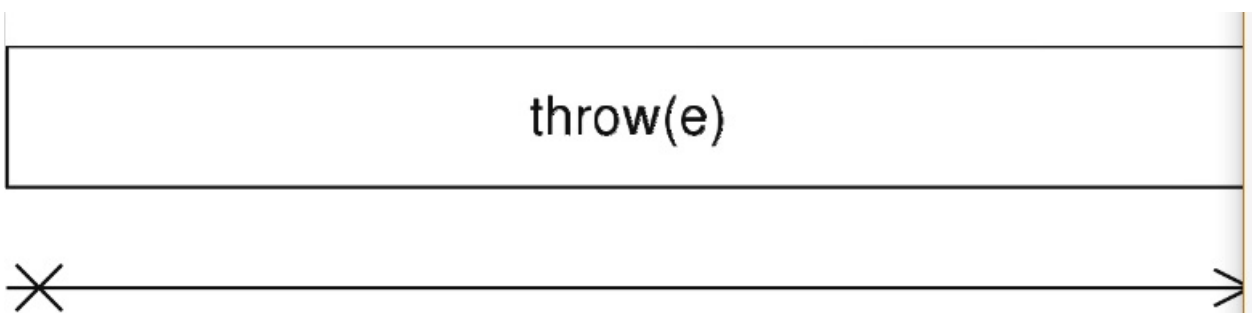
// => Next: 0
// => Next: 1
// => Next: 2
// => Completed
```

throw

- 语法：

```
public static throw(error:any, scheduler:Scheduler):Observable
```

- 功能：创建一个只发出error通知的Observable。

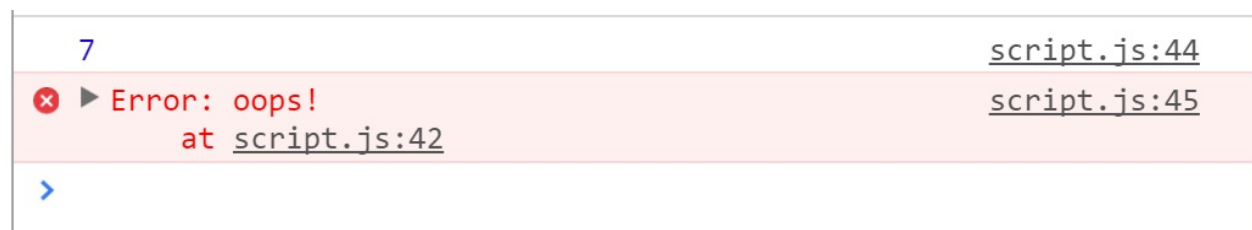


`throw`操作符对于创建一个只发出error通知的Observable非常有用。它可以用于与其他Observable合并，如在`mergeMap`中。

eg1:

```
var result = Rx.Observable.throw(new Error('oops!')).startWith(7);
result.subscribe(x => console.log(x), e => console.error(e));
```

result:



eg2:

```
var interval = Rx.Observable.interval(1000);
var result = interval.mergeMap(
  function(x) {
    return x === 13 ? Rx.Observable.throw('Thirteens are bad') :
    Rx.Observable.of('a', 'b', 'c');
  });
result.subscribe(
  function(x){ console.log(x)},
  function(e){console.error(e)}
);
```

eg2-result:

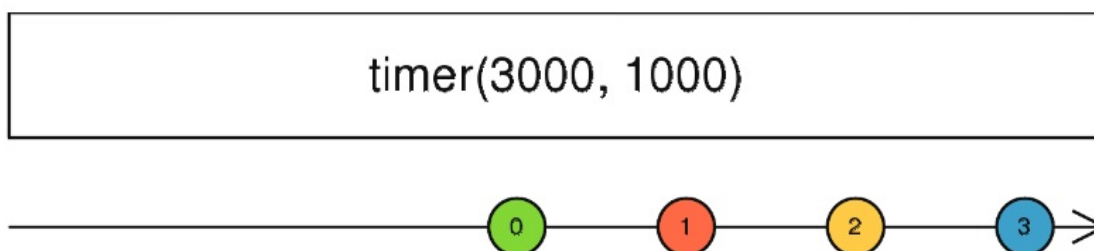
[illegible]

timer

- 语法：

```
public static timer(initialDelay:number|Date,period:number,scheduler):Observable
```

类似于interval,但是第一个参数用来设置发射第一个值得延迟时间



`timer`返回一个Observable，发出一个无限的上升整数序列，第二个参数为时间的间隔。第一次发射发生在指定的延迟之后。初始延迟可以是日期。默认情况下，此运算符使用异步Scheduler提供时间概念，但可以将任何scheduler传递给它。如果未指定period，则输出Observable仅发出一个值0，否则将发出无限序列。

eg:

```
var numbers = Rx.Observable.timer(3000, 1000);  
numbers.subscribe(x => console.log(x));
```

eg-result:

0	script.js:69
1	script.js:69
2	script.js:69
3	script.js:69
4	script.js:69
5	script.js:69
6	script.js:69
7	script.js:69
8	script.js:69
9	script.js:69
10	script.js:69
11	script.js:69
12	script.js:69
13	script.js:69
14	script.js:69
15	script.js:69
16	script.js:69
17	script.js:69
18	script.js:69
19	script.js:69
20	script.js:69
21	script.js:69
22	script.js:69
23	script.js:69
24	script.js:69
25	script.js:69
26	script.js:69
27	script.js:69
28	script.js:69
29	script.js:69
30	script.js:69
31	script.js:69

toAsync

- 语法：

```
Rx.Observable.toAsync(function:Function,[scheduler],[context]):Function
```

- 功能：将函数转换为异步函数。生成的异步函数的每次调用都会导致调用指定调度程序上的原始同步函数。

eg:

```
var func = Rx.Observable.toAsync(function (x, y) {
    return x + y;
});

// Execute function with 3 and 4
var source = func(3, 4);

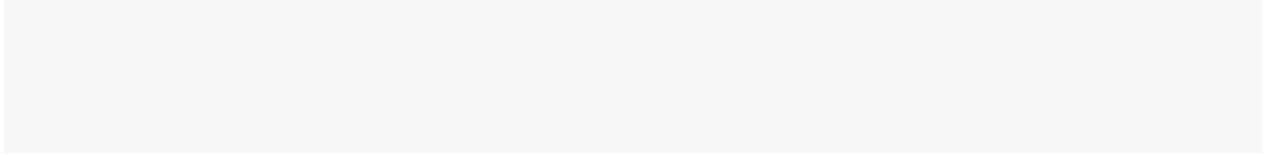
var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: 7
// => Completed
```

注意：该操作符返回的不是**observable**，而是一个异步函数，当异步函数被调用后（注意，调用并未立即执行），返回一个**observable**，该**observable**被订阅时，原函数才会被执行，并返回值。

using

- 语法：



webSocket

- 语法：

```
public static websocket(urlConfigOrSource: *): WebSocketSubject
```

官方无案例，等待后期更新

zip

- 语法：

```
public static zip(observables: *): Observable<R>
```

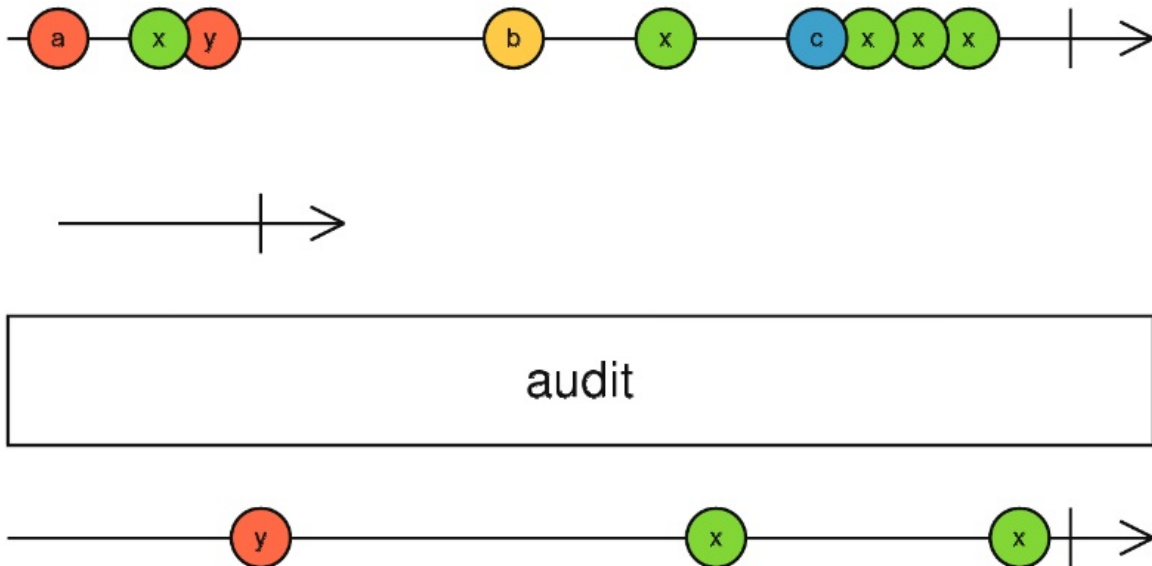

audit

- 语法：

```
public audit(durationSelector: function(value:T):Observable):Observable<T>
```

- 功能：在某个持续时间段内忽略原始observable发射的值，该方法的参数为一个函数，该函数需返回一个决定持续时长的observable或者promise。之后从原始observable发射最近的值，不断重复这个过程。

audit很像auditTime，但是其持续时长是由第二个observable所决定。



eg:

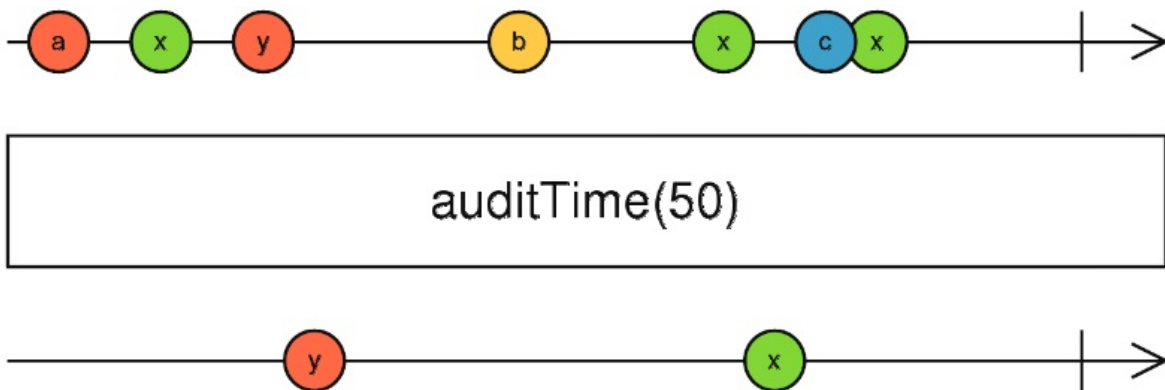
```
var clicks = Rx.Observable.fromEvent(document, 'click');  
var result = clicks.audit(ev => Rx.Observable.interval(1000));  
result.subscribe(x => console.log(x));  
// 每秒只会有一次单击会被发射，发射的时间点为每隔1秒
```

auditTime

- 语法:

```
public auditTime(durationTime: number, [scheduler: Scheduler]): Observable
```

- 功能：在某个时间段内，忽略原始observable发射的值，该时间段由设定的duration的值(单位为ms)来决定，每隔一个设定的时间段，将从原始的observable发射最近的值。不断重复这个过程。



eg:

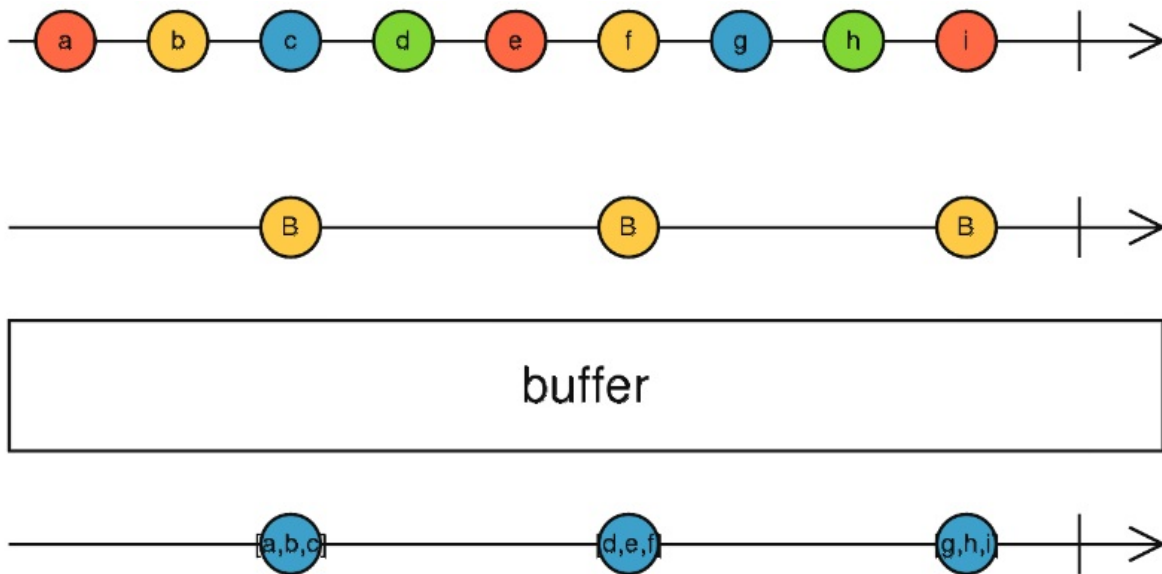
```
var clicks = Rx.Observable.fromEvent(document, 'click');  
var result = clicks.auditTime(1000);  
result.subscribe(x => console.log(x));  
// 每秒最多出现一次有效点击
```

buffer

- 语法：

```
public buffer(closingNotifier: Observable):Observable<T[]>
```

- 功能：缓存原始observable发射的值，直到作为参数的另一个observable发射了值。之后返回一个由这些缓存值组成的数组。



eg:

```
var clicks = Rx.Observable.fromEvent(document, 'click');  
var interval = Rx.Observable.interval(1000);  
var buffered = interval.buffer(clicks);  
buffered.subscribe(x => console.log(x));
```

eg-result:

► [0, 1, 2, 3, 4, 5] script.js:153

► [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17] script.js:153

script.js:153

[18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
► 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73,
74, 75, 76, 77, 78, 79, 80, 81, 82]

script.js:153

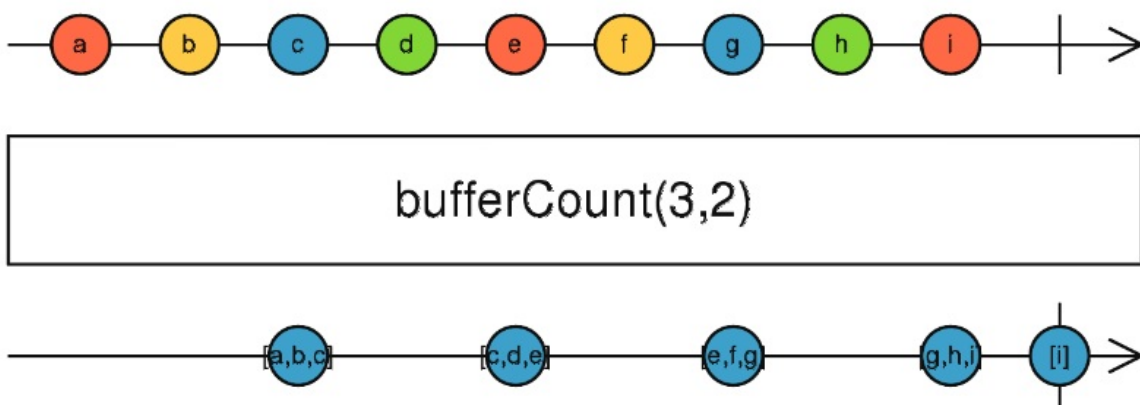
[83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96,
97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108,
109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119,
120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130,
► 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141,
142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152,
153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163,
164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174,
175, 176, 177, 178, 179, 180, 181, 182...]

bufferCount

- 语法：

```
public bufferCount(bufferSize: Number, startBufferEvery): Observable<T[]>
```

- 功能：缓存原始observable发射的值，直到达到bufferSize给定的上限



eg:

```
// 每点击2次，发射一次由之前点击事件组成的数组  
var clicks = Rx.Observable.fromEvent(document, 'click');  
var buffered = clicks.bufferCount(2);  
buffered.subscribe(x => console.log(x));
```

eg-result:

▶ [MouseEvent, MouseEvent]	<u>script.js:157</u>
▶ [MouseEvent, MouseEvent]	<u>script.js:157</u>
▶ [MouseEvent, MouseEvent]	<u>script.js:157</u>
▶ [MouseEvent, MouseEvent]	<u>script.js:157</u>
▶ [MouseEvent, MouseEvent]	<u>script.js:157</u>
▶	

eg-2(第二个参数的作用):

// 首次点击3次，发射一次由之前三次点击事件组成的数组。之后，每点击两次发射一次重置产生新数组，新数组的长度为3，第一个元素为上次发射数组的最后一个元素，后两个元素为本次2次点击事件。具体看结果截图，注意观察点击事件发生的**位置**。

```
var clicks = Rx.Observable.fromEvent(document, 'click');
var buffered = clicks.bufferCount(2, 1);
buffered.subscribe(x => console.log(x));
```

eg-2-result: 发射的为三个事件元素组成的数组(即第一个参数设定的值)

▶ [MouseEvent, MouseEvent, MouseEvent]	<u>script.js:157</u>
▶ [MouseEvent, MouseEvent, MouseEvent]	<u>script.js:157</u>
▶ [MouseEvent, MouseEvent, MouseEvent]	<u>script.js:157</u>
▶ [MouseEvent, MouseEvent, MouseEvent]	<u>script.js:157</u>
▶ [MouseEvent, MouseEvent, MouseEvent]	<u>script.js:157</u>
▶	

注意第二个数组第三次点击事件发生的位置

```
▼ Array[3] ⓘ  
  ► 0: MouseEvent  
  ► 1: MouseEvent  
  ▼ 2: MouseEvent  
    altKey: false  
    bubbles: true  
    button: 0  
    buttons: 0  
    cancelBubble: false  
    cancelable: true  
    clientX: 314  
    clientY: 354  
    composed: true
```

继续观察第二个数组，注意第一个事件元素发生的位置

```
► [MouseEvent, MouseEvent, MouseEvent]
```

```
▼ Array[3] ⓘ  
  ▼ 0: MouseEvent  
    altKey: false  
    bubbles: true  
    button: 0  
    buttons: 0  
    cancelBubble: false  
    cancelable: true  
    clientX: 314  
    clientY: 354  
    composed: true  
    ctrlKey: false  
    currentTarget: null  
    defaultPrevented: false  
    detail: 1  
    eventPhase: 0  
    fromElement: null  
    isTrusted: true  
    layerX: 314  
    layerY: 354  
    metaKey: false
```

总结一个：第一个参数为返回数组的长度，第二个为再发射几个值重置并产生新的数组。

此外，第二个参数的值可以大于第一个参数，比如，`arg1=3`，`arg2=4`。以上例为基础，这种情况会发生如下的事情：1、初次点击三次，发射一个数组

2、再点击四次，发射第二个数组

3、...重复步骤2...

这种情况的出现，有一件事就非常值得关注，在步骤2之后每一次发射的数组，我们都点击了四次，但是数组只会包含三个点击事件，是哪三个呢？答案是四次点击的后三个，也就是四次中的第一次会被忽略。

下面就用事实来验证在：首先我会在中间位置点击三次，产生一个数组，然后点击四次产生第二个数组、这四次的点击位置一次为:右上->左上->左下->右下

然后我们可以查看所包含的三次点击事件所发生的的位置

▼ 0: MouseEvent

```
altKey: false
bubbles: true
button: 0
buttons: 0
cancelBubble: false
cancelable: true
clientX: 3
clientY: 3
composed: true
```

第一个很明显是左上

▼ 1: MouseEvent

```
altKey: false
bubbles: true
button: 0
buttons: 0
cancelBubble: false
cancelable: true
clientX: 4
clientY: 612
composed: true
```


第二个很明显是左下

▼ 2: MouseEvent

```
altKey: false
bubbles: true
button: 0
buttons: 0
cancelBubble: false
cancelable: true
clientX: 1162
clientY: 608
```

第三个很明显是右下

所以我们失去了四次点击的第一次点击。

补充一个明了的例子。

```
var interval = Rx.Observable.interval(1000).bufferCount(3,4).subscribe(
  function(arr){
    console.log(arr);
  }
)
```



第二个参数直白点来说就是下一个缓存区的起点相对于上个缓存区起点的偏移量，取值范围是大于0，当第一个参数大于第二个时，则会忽略中间得发射值。比如这个例子，具体看截图。

bufferTime

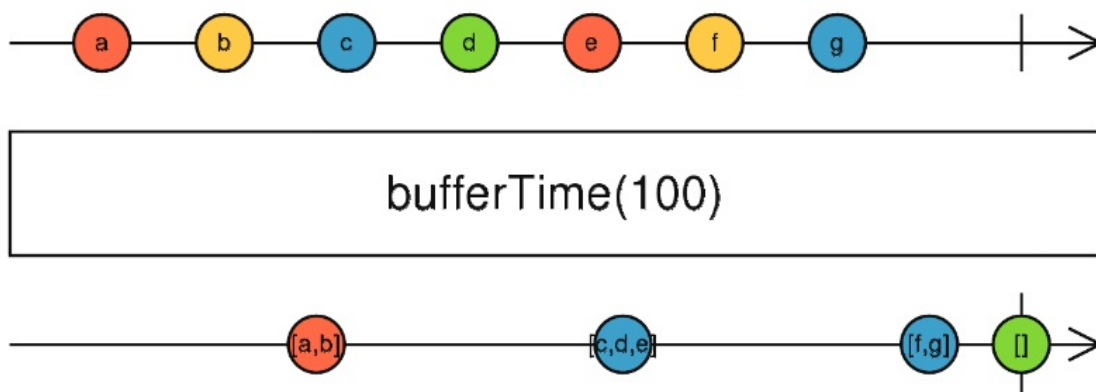
- 语法:

```
public bufferTime(bufferTimeSpan:number,[bufferCreationInterval:
numbewr],[maxBufferSize:number],[scheduler:Scheduler]):Observabl
e<T[]>
```

params:

- arg1: ****必须****、bufferTimeSpan设置发射值的时间间隔
- arg2: 可选、设置打开缓存区和发射值的时间间隔
- arg2: 可选、设置缓存区长度
- scheduler: 可选

- 功能: 把过去的值放入一个缓存区，并且按时间定期发射这些数组



eg-1 :

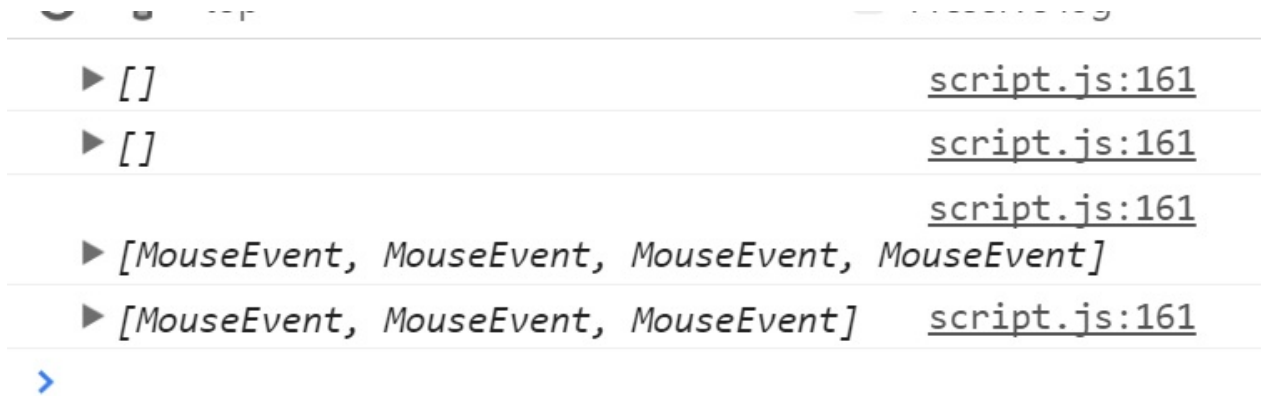
```
//只有第一个参数时
//每间隔一秒发射一次，数据包含在该时间段的值
var clicks = Rx.Observable.fromEvent(document, 'click');
var buffered = clicks.bufferTime(1000);
buffered.subscribe(x => console.log(x));
```

eg-1-result:

每隔bufferTimeSpan毫秒发送和重置缓冲区。

在第二次结束到第三次发射之间我点击了四次

第三次到第四次之间点击了三次



▶ []	<u>script.js:161</u>
▶ []	<u>script.js:161</u>
	<u>script.js:161</u>
▶ [MouseEvent, MouseEvent, MouseEvent, MouseEvent]	
▶ [MouseEvent, MouseEvent, MouseEvent]	<u>script.js:161</u>

>

eg-2

```
// 有前两个参数时
var clicks = Rx.Observable.fromEvent(document, 'click');
var buffered = clicks.bufferTime(2000, 5000);
buffered.subscribe(x => console.log(x));
```

eg-2

```
var clicks = Rx.Observable.fromEvent(document, 'click');
var buffered = clicks.bufferTime(2000, 5000);
buffered.subscribe(x => console.log(x));
```

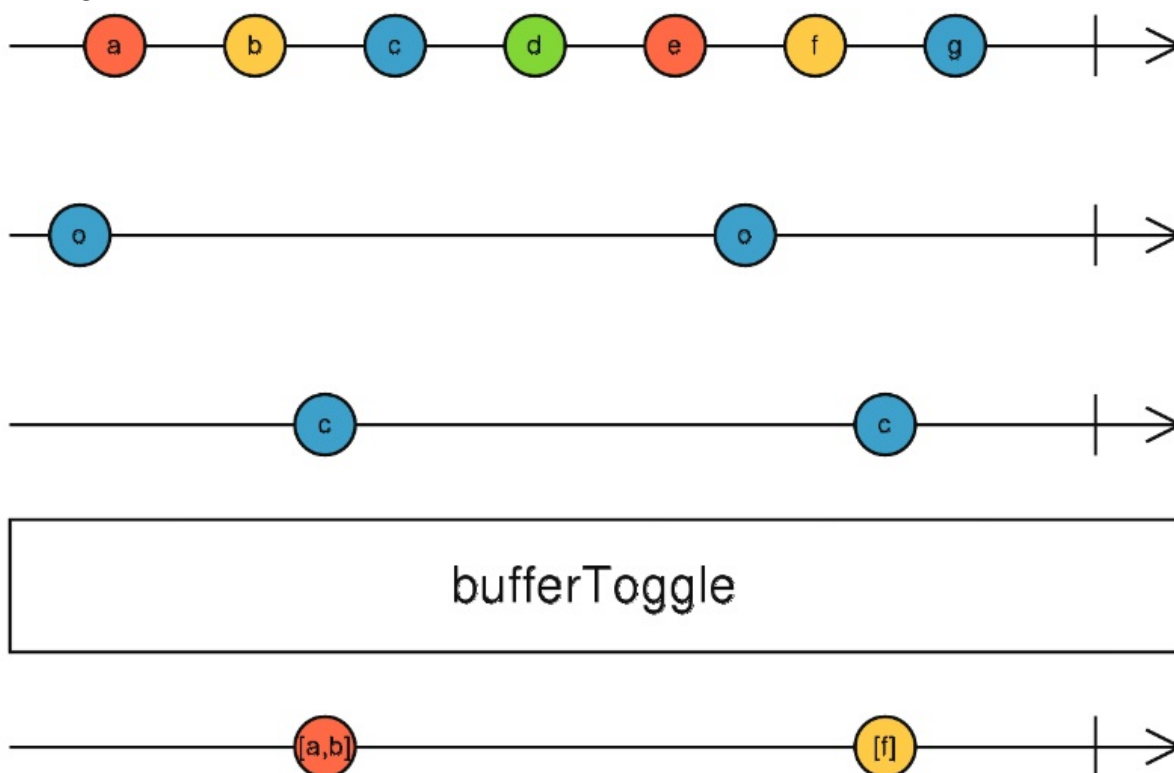
如果给定了bufferCreationInterval，则此操作符每隔bufferCreationInterval毫秒打开缓冲区，并且每隔bufferTimeSpan毫秒关闭（发射和重置）缓冲区。当指定可选参数maxBufferSize时，缓冲区将在bufferTimeSpan毫秒后或包含maxBufferSize元素时关闭。

bufferToggle

语法:

```
public bufferToggle(openings: SubscribableOrPromise<O>, closingSelector: function(value: O): SubscribableOrPromise): Observable<T[]>
```

功能: 以数组形式收集过去的值。在opening发射值时开始收集，并调用closingSelector函数获取一个Observable，以告知何时关闭缓存区。

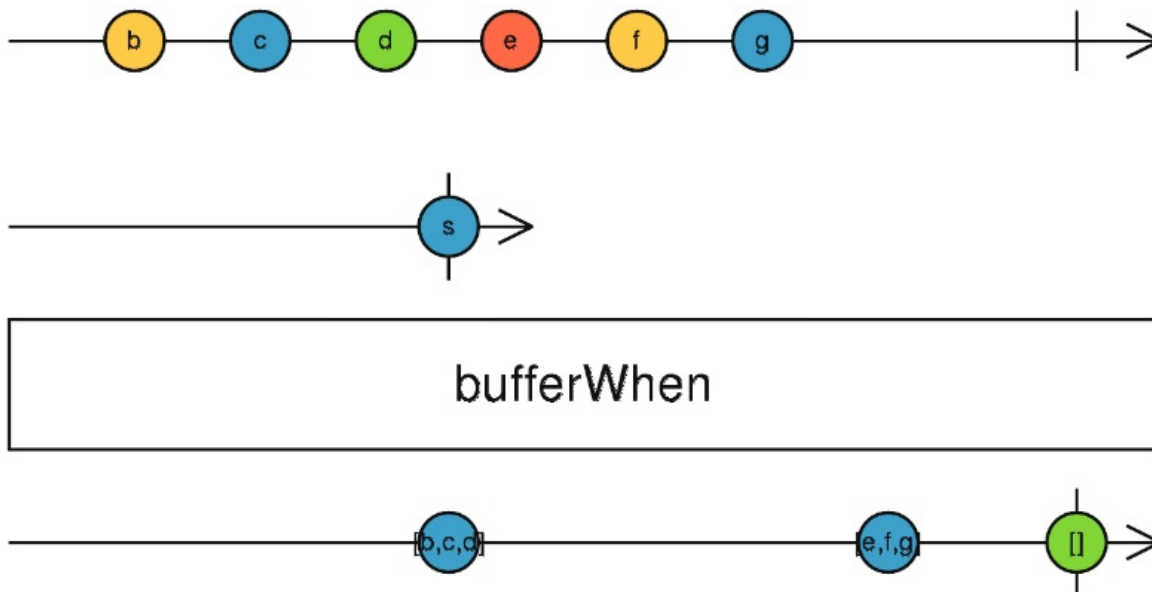


```
var clicks = Rx.Observable.fromEvent(document, 'click');
var openings = Rx.Observable.interval(1000);
var buffered = clicks.bufferToggle(openings, i =>
  i % 2 ? Rx.Observable.interval(500) : Rx.Observable.empty()
);
buffered.subscribe(x => console.log(x));
```


bufferWhen

-语法：

```
public bufferWhen(closingSelector: function(): Observable): Observable<T[]>
```



eg:

```
// 每隔1~5秒发射一次最新的click事件数组
var clicks = Rx.Observable.fromEvent(document, 'click');
var buffered = clicks.bufferWhen(() =>
  Rx.Observable.interval(1000 + Math.random() * 4000)
);
buffered.subscribe(x => console.log(x));
```

catch

语法:

```
public catch(selector: function): Observable
```

参数为一个函数: 它接受作为参数`err`，这是错误，并捕获。需要注意的是，如果使用了该操作符，`observer`的`error`方法将不会被执行，因为错误已经被`catch`操作符捕获。此外，不可以使用`try...catch`，因为此代码块是同步执行的。

combineAll

语法:

```
public combineAll(selector: function): Observable
```