Warm-up##

Trước khi tiếp xúc với những phần core, mình sẽ làm quên một chút với thế giới của RX !!!

**Thao tác các Event như 1 Array. **###

Đầu tiên, hãy xem đoạn code dưới đây.

```
[1,2,3,4,5,6,7,8].filter(function(num){
    return num%2;
});

// => [1, 3, 5, 7]
```

Đây là đoạn code trả về những số lẽ nằm trong các số tự nhiên từ 1 đến 8. Quá đơn giản phải ko ?

Tiếp theo mình sẽ thử viết 1 đoạn code cũng gần tương tự bằng RxJS nhưng khác biệt ở đây là thay đoạn xử lý Array bằng 1 Event.

Trước tiên hãy thử chạy Demo dưới đây. Click vào Click bao nhiêu lần cũng ko có xảy ra nhưng nếu ấn vào nút Alt và Click thì ở khung Console sẽ có xuất hiện dòng thông báo.

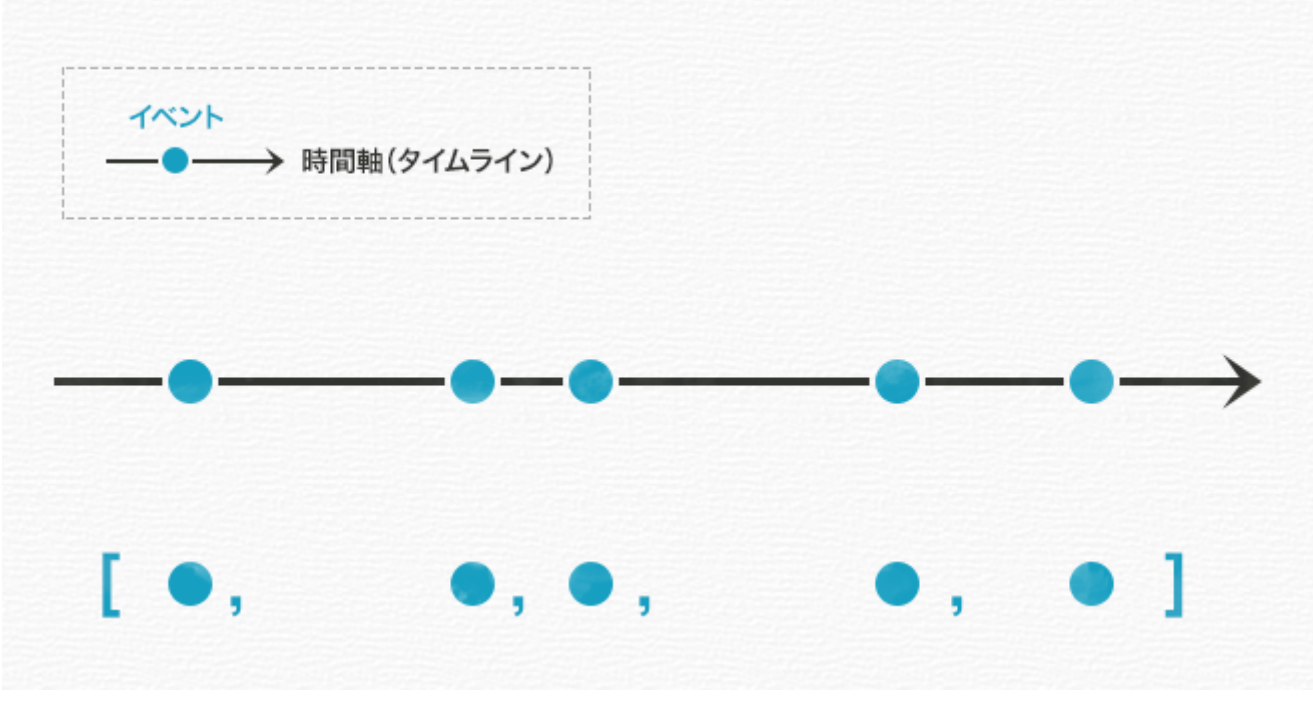
http://jsbin.com/bogoxetavu/1/edit?html,css,js,output

Đoạn code này sẽ như sau :

```
var btnClicks = Rx.Observable.fromEvent($('#btn'), "click");

btnClicks
    .filter(function (value) {
        return value.altKey;
    })
    .subscribe(function () {
        console.log('Holding Alt while Clicking! ');
    });
```

Ở đây, bạn có chút ý niệm gì về việc Event thì cũng khá giống Array ko ? Tất nhiên là nếu chỉ dừng lại ở 2 ví dụ trên thì ngoài việc tên method `filter()` mình đang đặt là giống nhau thì không có cái gì giống rồi.

Rõ ràng là khái niệm này hoàn toàn khác với Array, nhưng nếu nhìn từ 1 khía cạnh khác thì Event thực tế cũng là "1 điêm" phân tán nằm ở trục thời gian. Đến đây hẳn là các bạn sẽ có chút liên kết giữa 2 khái niệm này rồi nhỉ. Rõ ràng tập hợp các Event riêng biệt ở trên trục thời gian có thể được xem như là 1 Array.



Ok, đến đây chúng ta sẽ làm rõ việc liên tưởng này giải quyết được việc gì? Trong Array thì có rất nhiều các xử lý như là `filter()`、`map()`、`forEach()`、`reduce()`、`find()` . Nếu có thể áp dụng được những xử lý này một cách tương tự cho Event thì quá hữu ích.

Trở lại với ví dụ lúc nãy về việc viết 1 method `filter()` cho Array Click Event. Ở đây thì kết quả sẽ là 1 Array chứa những Event mà mình có ấn vào nút Alt. Cuối cùng thì để output được ra những kết quả mong muốn từ Array thư được, mình dùng phương thức `subscribe` .

Nếu bạn đọc nào cần một giải thích trực quan hơn thì có thể tham khảo chương được viết ở đây.

http://jsbin.com/tojapaxope/1/edit?html,css,js,output

Flow chính của quá trình xử lý này sẽ là như sau.

1 Event được sinh ra => cho qua `filter()` =>thỏa mãn các điều kiện cần tìm thì cho event đấy vào Array => thông báo đến `subscriber`

STREAM##

Cho đến bây giờ thì mình dùng khái niệm Mảng các Event cho dễ hiểu nhưng thực ra thì vẫn còn 1 cái tên khá hay hơn cho nó là Stream.

Vâng, đây chính là Stream !!!

Trong Rx thì khái niệm này thường được gọi là `Observable` / `Observable-Sequence` . Theo mình rõ ràng là cách gọi Stream dễ hiêu hơn nhiều. Trong bài viết này mình sẽ sử dụng cách gọi là Stream. Các bạn chú ý nhé.

Trong Rx thì có rất nhiều method phục vụ cho việc create các Stream. Đây là các danh sách những phương thức đó : `Create` , `Defer` , `Empty` / `Never` / `Throw` , `From` , `Interval` , `Just` , `Range` , `Start` , `Timer` .

Các Operator để thao tác với Stream##

Ngoài `filter` ra thì có rất nhiều cá method có sẵn trong RxJS hõ trợ việc thao tác với Stream như `map` , `reduce` , `merge` , `concat` , `zip` . Các method như thế này được gọi chung là Operator.

Cho đến bây giờ thì flow xử lý sẽ là:

1. Tạo mới 1 Stream (Observable)
2. Truyền thêm các Operator
3. Subcribe

Như ở ví dụ lúc nãy, giá trị trả về từ `Rx.Observable.fromEvent()` có thể xem như là Observable. Đối với `Observable` này mình có 1 Operator là （ `filter` ）, cuối cùng thì các giá trị két quả sẽ nhận được ở `subscribe` .

Các bạn có thể tham khảo link dưới đây để có thêm nhữn thông tin chi tiết hơn.

Operators by Categories：

http://xgrommx.github.io/rx-book/content/getting_started_with_rxjs/creating_and_querying_observable_sequences/operators_by_category.html

Operation Chain##

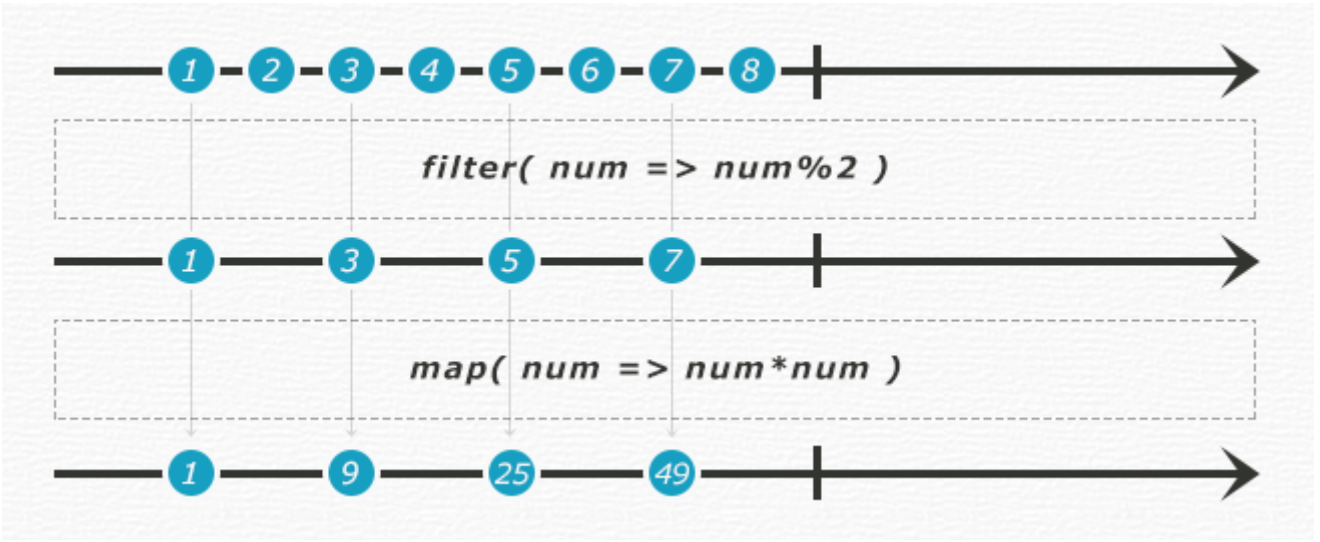
Đói với Array thì Method chain sẽ như dưới đây.

```
[1,2,3,4,5,6,7,8]
.filter(function(num){
    return num%2;
})
.map(function(num){
    return num*num;
});

// => [1, 9, 25, 49]
```

Tất nhiên RxJS cũng có khả năng làm như vậy.

```
Rx.Observable.from([1, 2, 3, 4, 5, 6, 7, 8]) // Thay Array bằng Stream (Observable)
    .filter(function (num) { //Giá trị trả về : Observable
        return num % 2;
    }).map(function (num) { // Giá trị trả về: Observable
        return num * num;
    }).forEach(function (num) {  // `forEach` là alias của `subscribe`. Giá trị trả về: Diposable
        return console.log(num);
    });

// => 1
// => 9
// => 25
// => 49
```

Dưới đây là hình ảnh mình họa giải thích cho ví dụ trên.



Kí tự 「|」 biểu hiện kết thúc Stream. Graph kiểu như thế này được gọi là Marble-Diagrams. Loại Graph này rất có ích cho việc giải thích hoạt động của Operator. Tiếp tục, chúng ta hãy thử dùng Operator `delayWithSelector`. output giá trị mỗi 50 mili giây.

```
// khởi tao observer để truyền vào `subscribe()`
var observer = Rx.Observer.create(function (num) {
    // Khi có 1 giá trị mới được push vào
    return console.log("onNext: " + num);
}, function (error) {
    // Khi có lỗi phát sinh
    return console.log("onError: " + error);
}, function () {
    // Stream sẽ kết thúc khi tất cả các gá trị được push vào
    return console.log('onCompleted');
});

Rx.Observable.from([1, 2, 3, 4, 5, 6, 7, 8])
    // return giá trị mỗi 500 mili giây
    .delayWithSelector(function (num) {
        return Rx.Observable.timer(num * 500);
    }).filter(function (num) {
        return num % 2;
    }).map(function (num) {
        return num * num;
    }).subscribe(observer);

// => onNext: 1
// => onNext: 9
// => onNext: 25
// => onNext: 49
// => onCompleted
```

http://jsbin.com/bedoca/1/edit

Cho đến bây giờ thì mình chưa thuyết mình 1 điểm, đó là `subscribe()` có 2 kiểu truyền tham số vào.

Kiểu thứ nhất sẽ là Object.

```
var onNext = function(){}; // callback khi Push
var onError = function(){};  // callback khi lỗi
var onCompleted = function(){}; // callback khi kết thúc

o.subscribe( onNext, onError, onCompleted );
```

Kiểu thứ 2 là truyền vào `observer Object`

```
o.subscribe(observer);
```

Observer và Observable###

Ở mục này chúng ta sẽ tìm hiểu mối quan hệ giữa Observer và Observable.

Bạn hãy tham khảo đoạn code sample ở đưới đây. Đoạn code này sinh tạo ra 1 Object Observable thông quá hàm `Rx.Observable.create()` , lưu giá trị ấy vào biến `source` . Ở đây thì có sử dụng o `bserver.onNext()` để sinh ra giá trị cho `observe` .Ở trong subscription thì có 1 hàm `dispose()` để thực hiện cho mục đích nếu giá trị trả về không thỏa mãn thì mình sẽ thực hiện loại trừ cái Observable đó.

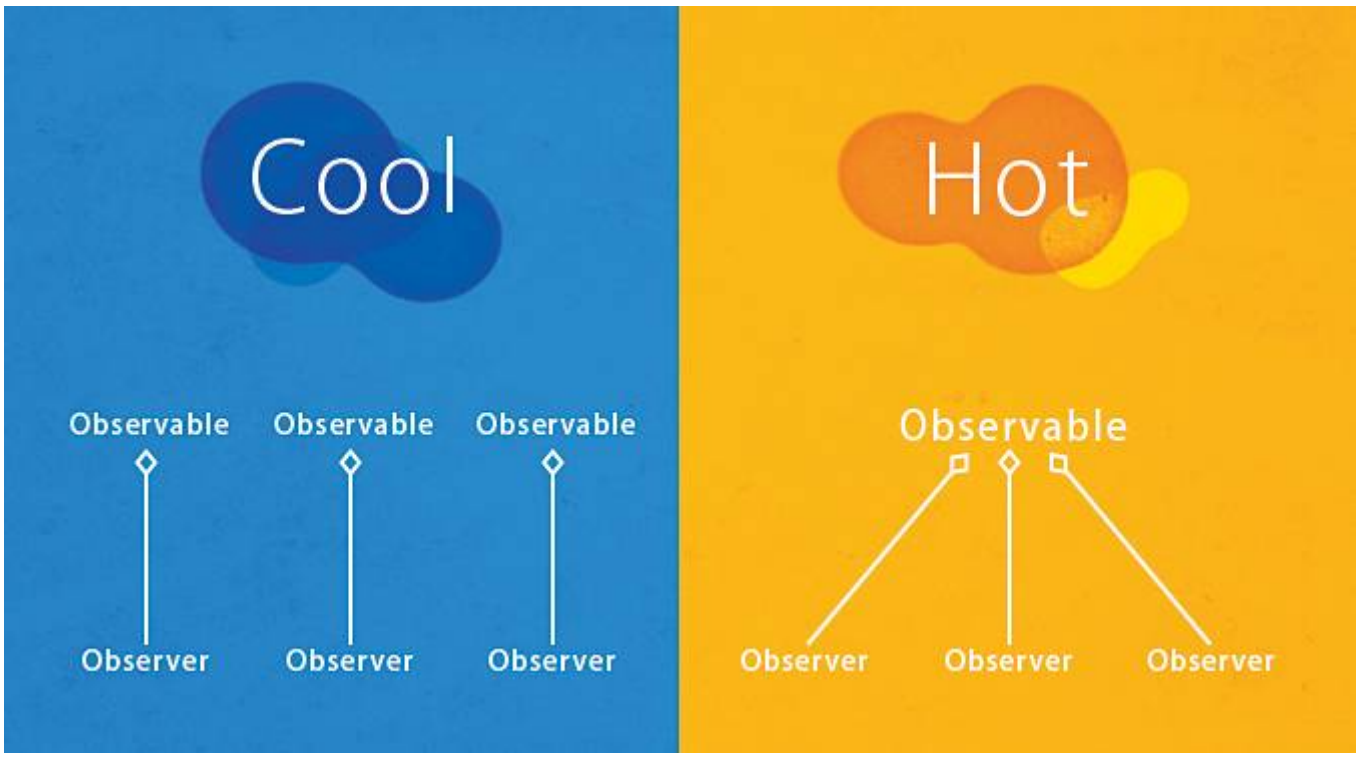
```
var source = Rx.Observable.create(function (observer) {
    // sử dụng `onNext` push `num` vào observer lần lượt 500 mili giây
    var num = 0;
    var id = setInterval(function () {
        observer.onNext(num++);
    }, 500);

    setTimeout(function () {
        observer.onCompleted();
    }, 10000);

    return function () {
        console.log('disposed');
        clearInterval(id);
    };
});

var subscription = source.subscribe(
    function (x) {
        console.log('onNext: ' + x);
    },
    function (e) {
        console.log('onError: ' + e.message);
    },
    function () {
        console.log('onCompleted');
    });

setTimeout(function () {
    subscription.dispose();
}, 5000);

// => onNext: 0
// => onNext: 1
// => onNext: 2
// => onNext: 3
// => onNext: 4
// => onNext: 5
// => onNext: 6
// => onNext: 7
// => onNext: 8
// => disposed
```

http://jsbin.com/luvazo/1/edit

Ở ví dụ trên thì mình mới chỉ sử dụng `Next` . Ngoài method này ra thì việc thông báo cho subscriber mình còn có thể sử dụng những method như là `Next` / `Error` / `Completed` .

Cold Observable và Hot Observable###

Observable có 2 trạng thái là 「`Cold`」và「`Hot`」.



Cold Observable##

Ví dụ ở dưới thì mình dùng 2 lần `subscribe()` cho cùng một Observable(`source`). Nếu bạn quan sát log, bạn sẽ nhận thấy các Observer lấy ra giá trị từ các Sequence mới.

Mội Observer sẽ lấy giá trị ra từ chính Observable của chính nó.

```
var source = Rx.Observable.interval(1000),
    subscription1 = source.subscribe(
        function (x) {
            console.log('Observer 1: onNext: ' + x);
        }
    ),
    subscription2;

setTimeout(function () {
    subscription2 = source.subscribe(
        function (x) {
            console.log('Observer 2: onNext: ' + x);
        }
    );
}, 2000);

setTimeout(function () {
    subscription1.dispose();
    subscription2.dispose();
}, 5000);

// => Observer 1: onNext: 0
// => Observer 1: onNext: 1
// => Observer 1: onNext: 2
// => Observer 2: onNext: 0
// => Observer 1: onNext: 3
// => Observer 2: onNext: 1
```

Hot Observable##

Ví dụ dưới đây thì sử dụng hàm `publish()`, chuyển từ Cold Observable（ `source` ）thành Hot Observable（ `hot` ）. Từ log, bạn có thể thấy điểm khác biệt với Cold Observable là mỗi Observer sẽ lấy giá trị mới nhất từ Hot Observable（ `hot` ）.

「Hot Observable」thì sẽ sinh ra 1 giá trị giống nhau ở cùng 1 timing cho tất cả các Observer.

```
// Tạo mới Observable
var source = Rx.Observable.interval(1000);

// Chuyển thành Hot Observable
var hot = source.publish();

// Tại thời điểm này thì giá trị chưa được push vào
var subscription1 = hot.subscribe(
    function (x) {
        console.log('Observer 1: onNext: %s', x);
    }
);

console.log('Current Time after 1st subscription: ' + Date.now());

// Sau đấy 3 giây ……
setTimeout(function () {
    // sử dụng hàm `connect()` kết nối vào `source`
    // Ở đây thì push những giá trị lấy ra từ source sẽ được push vào hot observer
    hot.connect();

    console.log('Current Time after connect: ' + Date.now());

    // sau đấy 3 giây tiếp theo
    setTimeout(function () {
        console.log('Current Time after 2nd subscription: ' + Date.now());
        var subscription2 = hot.subscribe(
            function (x) {
                console.log('Observer 2: onNext: %s', x);
            }
        );
    }, 3000);
}, 3000);

// => Current Time after 1st subscription: 1425834043641
// => Current Time after connect: 1425834046647
// => Observer 1: onNext: 0
// => Observer 1: onNext: 1
// => Current Time after 2nd subscription: 1425834049649
// => Observer 1: onNext: 2
// => Observer 2: onNext: 2
// => Observer 1: onNext: 3
// => Observer 2: onNext: 3
// => Observer 1: onNext: 4
// => Observer 2: onNext: 4
// => Observer 1: onNext: 5
// => Observer 2: onNext: 5
// => Observer 1: onNext: 6
// => Observer 2: onNext: 6
```

**About Subject **##

Subject về cơ bản là 1 Class của RX nhưng đây là 1 Class rất quan trọng. Class này kế thừa từ cả Observable và Observer. Do đó, nếu Subject mà nằm trong Observable thì nó cũng nằm trong Observer.

Subject kết hợp giữa Observer và Observable###

Theo như ví dụ dưới đây, tạo 1 Subject và sau đầy thì Subscribe nó. Tiếp tục, lại sư dụng lại Subject ây và Push giá trị vào Observer mà nó được tạo ra.

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

var subscription = subject.subscribe(
    function (x) { console.log('onNext: ' + x); },
    function (e) { console.log('onError: ' + e.message); },
    function () { console.log('onCompleted'); });

subject.onNext(1);
// => onNext: 1

subject.onNext(2);
// => onNext: 2

subject.onCompleted();
// => onCompleted

subscription.dispose();
```

VIBLO Search Viblo Sign In/Sign up

Một trong số những mục đích của Subject là lắng nghe Broadcast. Subject cũng giống như Observable, có interface `subscribe()` nhưng có 1 chút khác biệt là `subscribe()` của Subject thì không phải để ý gì đến scheduler.

```
// Observable khởi tạo giá trị mỗi 1 giây
var source = Rx.Observable.interval(1000);

var subject = new Rx.Subject();

// truyền vào `source`
var subSource = source.subscribe(subject);

// Broadcast 1
var subSubject1 = subject.subscribe(
    function (x) { console.log('Value published to observer #1: ' + x); },
    function (e) { console.log('onError: ' + e.message); },
    function () { console.log('onCompleted'); });

// Broadcast 2
var subSubject2 = subject.subscribe(
    function (x) { console.log('Value published to observer #2: ' + x); },
    function (e) { console.log('onError: ' + e.message); },
    function () { console.log('onCompleted'); });

setTimeout(function () {
    // cho kết thúc sau 5 giây sau đó
    subject.onCompleted();
    subSubject1.dispose();
    subSubject2.dispose();
}, 5000);

// => Value published to observer #1: 0
// => Value published to observer #2: 0
// => Value published to observer #1: 1
// => Value published to observer #2: 1
// => Value published to observer #1: 2
// => Value published to observer #2: 2
// => Value published to observer #1: 3
// => Value published to observer #2: 3
// => onCompleted
// => onCompleted
```

Subject đảm nhận chức năng Proxy###

Hãy tham khảo đoạn code dưới đây.

+10

```
setTimeout(function () {
    // khởi tạo giá trị cho subscriber của subject sau 2 giây sau đó
    subject.onNext('from SUBJECT');
}, 2000);

// => Value published to observer #1: 0
// => Value published to observer #2: 0
// => Value published to observer #1: from SUBJECT
// => Value published to observer #2: from SUBJECT
// => Value published to observer #1: 1
// => Value published to observer #2: 1
// => Value published to observer #1: 2
// => Value published to observer #2: 2
// => Value published to observer #1: 3
// => Value published to observer #2: 3
// => onCompleted
// => onCompleted
```

About Scheduler##

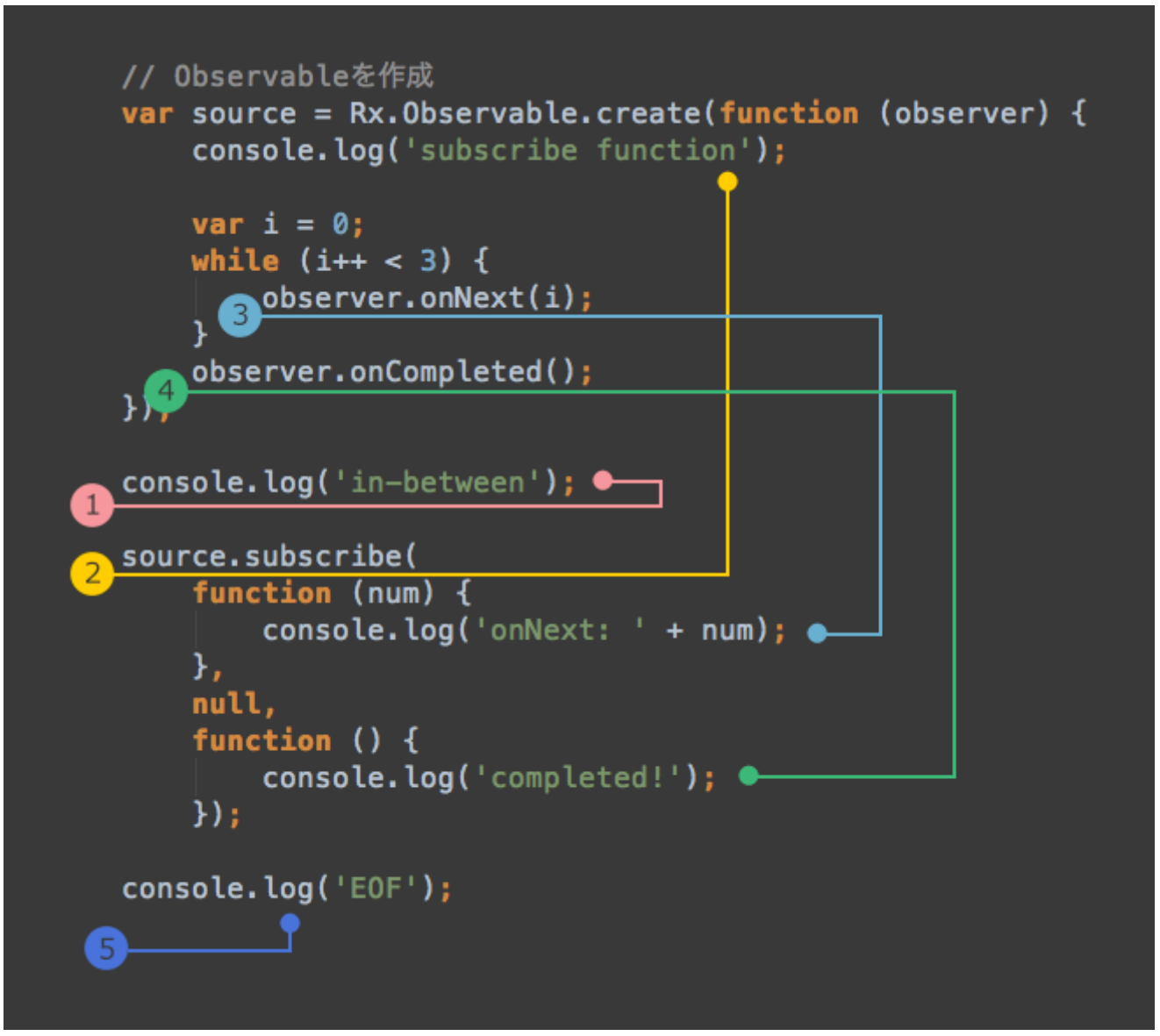
Scheduler là 1 trong những Class của RX.

Class này làm nhiệm vụ quyết định「Lúc nào có thể bắt đầu Subscribe ?」 「lúc nào giá trị được khởi tạo ?」

Hãy tham khảo đoạn code đưới đây.

```
// Tạo mới Observable
var source = Rx.Observable.create(function (observer) {
    console.log('subscribe function');

    var i = 0;
    while (i++ < 3) {
        observer.onNext(i);
    }
    observer.onCompleted();
});

// source = source.subscribeOn(Rx.Scheduler.timeout);
// source = source.observeOn(Rx.Scheduler.timeout);

console.log('in-between');

source.subscribe(
    function (num) {
        console.log('onNext: ' + num);
    },
    null,
    function () {
        console.log('completed!');
    });

console.log('EOF');

// => in-between
// => subscribe function
// => onNext: 1
// => onNext: 2
// => onNext: 3
// => completed!
// => EOF
```

Thứ tự Output hẳn là đúng với mọi người dự đoán. Dưới đây là hình ảnh minh họa cho thứ thự Output.



Bỏ phần comment ở đoạn `source = source.subscribeOn(Rx.Scheduler.timeout);` kết quả sẽ như thế này

```
// => in-between
// => EOF
// => subscribe function
// => onNext: 1
// => onNext: 2
// => onNext: 3
// => completed!
```

Các hàm sẽ được ưu tiên sử dụng theo thứ tự như dưới đây.

- setImmediate
- nextTick
- postMessage
- MessageChannel
- script readystatechanged
- setTimeout

Ví dự như nếu trong Node.js thì sẽ không phải là `setTimeout` mà sẽ sử dụng `setImmediate` hoặc là `nextTick` . Việc làm này sẽ giúp tránh được việc block UI khi có những xử lý nặng.

Tiếp tục, bạn hãy thử comment out dòng `source = source.subscribeOn(Rx.Scheduler.timeout);` , bỏ comment ở đoạn `source = source.observeOn(Rx.Scheduler.timeout);`

```
// => in-between
// => subscribe function
// => EOF
// => onNext: 1
// => onNext: 2
// => onNext: 3
// => completed!
```

Nguyên lý ở đây cũng tương tự như trên. Đoạn xử lý `onNext` sẽ được cho vào Queue và gọi Loop Event tiếp theo từ Queue.

** Các chủng loại Scheduler **###

Có 3 loại Scheduler

- timeoutScheduler
- immediateScheduler
- currentThreadScheduler

** Một số ví dụ minh họa **##

Tạm thời đến đây thì các bạn đã nắm được hết các khái niệm cơ bản của RxJS. Dưới đây mình sẽ cung cấp một số ví dụ mẫu để các bạn có thể hiểu thêm về RxJS.

** Ví dụ 1 : **###

Những Operator được sử dụng:

- flatMap
- takeUntil

Thuyết minh : sử dụng `flatMap` , chuyển từ event `mousedown` về event `mousemove` liên tục cho đến khi `mouseup`

http://jsbin.com/giqayi/1/edit

```
(function() {
    var $box, box_width, mousedown_events, mousemove_events, mouseup_events, source;

    $box = $('#box');
    mouseup_events = Rx.Observable.fromEvent($box, 'mouseup');
    mousemove_events = Rx.Observable.fromEvent(document, 'mousemove');
    mousedown_events = Rx.Observable.fromEvent($box, 'mousedown');

    source = mousedown_events.flatMap(function(event) {
        var start_left, start_pageX, start_pageY, start_top;
        start_pageX = event.pageX;
        start_pageY = event.pageY;
        start_left = parseInt($box.css('left'));
        start_top = parseInt($box.css('top'));
        $box.addClass('hovering');
        return mousemove_events.map(function(e) {
            return {
                left: start_left + (e.pageX - start_pageX),
                top: start_top + (e.pageY - start_pageY)
            };
        }).takeUntil(mouseup_events);
    });

    mouseup_events.subscribe(function() {
        $box.removeClass('hovering');
    });

    source.subscribe(function(pos) {
        TweenLite.set($box, {
            left: pos.left,
            top: pos.top
        });
    });

})();
```

Ví dụ 2 :##

Những Operator được sử dụng:

- combinelLatest

Thuyết minh : sử dụng `BehaviorSubject` để thay đổi UI. `BehaviorSubject` sẽ lấy ra giá trị được khởi tạo gần nhất. Sau đó sử dụng `combineLatest` để tổng hợp các giá trị của `BehaviorSubject` . Nếu so sánh với jQuery thì cũng công việc nhưng với RxJs việc xử lý sẽ dễ hơn nhiều.

http://jsbin.com/zoceme/1/edit

```
(function () {
    var $color, $combined, $h1, $size, $text, bind, color, size, text;

    $h1 = $('h1');
    $text = $('.text>input');
    $size = $('.size>input');
    $color = $('.color>input');
    $combined = $('#combined');

    text = new Rx.BehaviorSubject($text.val());
    size = new Rx.BehaviorSubject($size.val());
    color = new Rx.BehaviorSubject($color.val());

    text.subscribe(function (val) {
        $h1.text(val);
    });

    size.subscribe(function (val) {
        $h1.css('font-size', val + 'px');
    });

    color.subscribe(function (val) {
        $h1.css('color', val);
    });

    bind = function (eType, elem, subject) {
        Rx.Observable.fromEvent(elem, eType).subscribe(function (e) {
            subject.onNext(e.target.value);
        });
    };

    text.combineLatest(size, color, function (text, size, color) {
        return "text: " + text + "<br>Size: " + size + "px<br>Color: " + color;
    }).subscribe(function (val) {
        return $combined.html(val);
    });

    bind('keyup', $text, text);
    bind('keyup change', $size, size);
    bind('change', $color, color);

})();
```

** Ví dụ 3 😗*##

Những Operator được sử dụng:

- filter
- concat
- timeout
- take

Thuyết minh : Ở đây thì tất cả những phần quan trọng nhất sẽ nằm ở hàm `createCommand` .

http://jsbin.com/mocoma/1/edit

```
(function () {
    var $ken, $stage, createCommand, keydowns, keys, skill;

    keys = {
        left: 37,
        right: 39,
        up: 38,
        down: 40,
        a: 65,
        s: 83
    };

    keydowns = Rx.Observable.fromEvent(document, 'keydown');

    /**
     * hàm helper để tạo command
     * @param {Array} array lưu các key được tạo thành nên từ command combination_key theo thứ tự
     * @param {Function}
     hàm callback thực hiện khi mà có 1 command callback được ấn vào thành công
    **/
    createCommand = function (combination_keys, timeout, callback) {
        var get_source, watch;

        get_source = function () {
            var source;
            source = Rx.Observable.empty();
            combination_keys.forEach(function (keyCode, index) {
                var this_key;
                this_key = keydowns.filter(function (e) {
                    var is_correct;
                    is_correct = e.keyCode === keyCode;
                    if (is_correct === false) {
                        throw Error('incorrect key pressed');
                    } else {
                        return is_correct;
                    }
                }).take(1);
                if (index > 0) {
                    this_key = this_key.timeout(timeout);
                }
                source = source.concat(this_key);
            });
            return source;
        };

        watch = function () {
            var source;
            source = get_source();
            source.subscribe(function () {
                console.log('correct');
            }, function (e) {
                console.log(e.message);
                watch();
            }, function () {
                console.log('completed');
                watch();
                callback();
            });
        };
        watch();
    };

    // xử lý animation
    $ken = $('.ken');
    $stage = $('.stage');
    skill = {
        hadoken: function () {
            var $fireball, tl;
            tl = new TimelineLite();
```

```
            $stage.append($fireball);

            tl.add(function () {
                $fireball.addClass('moving').animate({
                    left: '+=250'
                }, 3000, 'linear');
            }, 0.33).add(function () {
                $ken.removeClass('hadoken');
                console.log('yes');
            }, 0.5).add(function () {
                $fireball.addClass('explode');
            }, 3.3);
        },
        senpukyaku: function () {
            var tl;
            $ken.addClass('tatsumaki');
            tl = new TimelineLite();
            tl.add((function () {
                $ken.removeClass('tatsumaki');
            }), 2);
        }
    };

    // setting command
    createCommand([keys.left, keys.down, keys.right, keys.a], 500, skill.hadoken);
    createCommand([keys.right, keys.down, keys.left, keys.s], 500, skill.senpukyaku);

})();
```

Các thư viện đối thủ##

Tại thời điểm hiện nay thì Bacon.js là 1 thư viên khá giống với RxJS và cũng có thể xem là đối thủ của RxJS. Dưới góc nhìn cá nhân thì thư viện này dễ hiểu hơn RxJS về các thuật ngữ. Nếu bạn nào đã nắm được RxJS thì việc chuyển qua sử dụng thư viện này cũng khá đơn giản.

Bạn có thể tham khảo link sau đây để có thêm thông tin về việc so sánh giữa các thư viện.
http://qiita.com/kondei/items/17e5d4867a0652911e52

Lời kết##

Bài viết lần này là với chủ để là "nhập môn" vì bản thân người viết (cũng như người dịch) cũng là những người rất mới đối với RxJS. Đơn thuần là cảm thấy có hứng thú với Reactive Programming nên tìm hiểu. Một mục đích chính khác nữa mà mình viết bài viết này cũng là muốn phổ cập RxJS, đưa RxJS đến với nhiều người hơn.

Tài liệu tham khảo##

- RxJS https://github.com/Reactive-Extensions/RxJS
- RxJS GitBook http://xgrommx.github.io/rx-book/
- Operators By Category http://reactivex.io/documentation/operators.html
- Rx Marbles http://rxmarbles.com/
- The introduction to Reactive Programming you've been missing https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
- Learn RxJS https://github.com/jhusain/learnrx
- The Reactive Extensions for JavaScript (RxJS) Examples https://github.com/Reactive-