<https://yq.aliyun.com/articles/65027>

## rxjs简单入门

[雨异](https://yq.aliyun.com/users/1499314489531632) 2016-11-28 17:16:29 浏览25685 评论2

* [javascript](https://yq.aliyun.com/tags/type_blog-tagid_42/)
* [阿里技术协会](https://yq.aliyun.com/tags/type_blog-tagid_523/)
* [序列](https://yq.aliyun.com/tags/type_blog-tagid_1484/)
* [Create](https://yq.aliyun.com/tags/type_blog-tagid_2146/)
* [source](https://yq.aliyun.com/tags/type_blog-tagid_2206/)
* [数组](https://yq.aliyun.com/tags/type_blog-tagid_2795/)
* [rxjs](https://yq.aliyun.com/tags/type_blog-tagid_15966/)

摘要： # rxjs简单入门 > rxjs全名Reactive Extensions for JavaScript，Javascript的响应式扩展, 响应式的思路是把随时间不断变化的数据、状态、事件等等转成可被观察的序列(Observable Sequence)，然后订阅序列中那些Observable对象的变化，一旦变化，就会执行事先安排好的各种转换和操作 \*\*rxjs适用于异步场景，即前端

# rxjs简单入门

rxjs全名Reactive Extensions for JavaScript，Javascript的响应式扩展, 响应式的思路是把随时间不断变化的数据、状态、事件等等转成可被观察的序列(Observable Sequence)，然后订阅序列中那些Observable对象的变化，一旦变化，就会执行事先安排好的各种转换和操作

**rxjs适用于异步场景，即前端交互中接口请求、浏览器事件以及自定义事件。通过使用rxjs带给我们前所未有的开发体验。**

1. 统一异步编程的规范，不管是Promise、ajax还是事件，通通封装成序列(Observable Sequence)，一旦有异步环节发生变更，观察序列即可截获发生变更的信息。
2. 前端业务层和展现层解耦，比如展现层不需要关系指定事件触发时和DOM无关的处理逻辑。同时业务层也能组装异步操作中多个异步逻辑之间的关系，无需暴露给展现层。展现层关心的是：异步操作其中环节的数据变化。
3. rxjs开发业务层具有高弹性，高稳定性，高实时性等特点。

废话不多说，此篇文档结合模拟场景的例子，通过傻瓜式的描述来说明rxjs常用的方法以及组合关系。

### 1. Let's Go

rxjs应用观察者模式，其中包含2个重要的实例：Observer观察者和Subject被观察对象，多个Observer注册到Subject中，在Subject功能触发时，会通知注册好的Observab列表，逐一通知其响应观察变更信息。

#### 1.1 quick start

1. 先从官网搬来rxjs的几个实例概念
   * Observable: 可观察的数据序列.
   * Observer: 观察者实例，用来决定何时观察指定数据.
   * Subscription: 观察数据序列返回订阅实例.
   * Operators: Observable的操作方法，包括转换数据序列，过滤等，所有的Operators方法接受的参数是上一次发送的数据变更的值，而方法返回值我们称之为发射新数据变更.
   * Subject: 被观察对象.
   * Schedulers: 控制调度并发，即当Observable接受Subject的变更响应时，可以通过scheduler设置响应方式，目前内置的响应可以调用Object.keys(Rx.Subject)查看。
2. 我们最常用也最关心的Observable，四个生命周期：创建 、订阅 、 执行 、销毁。
   * 创建Obervable，返回被观察的序列源实例，该实例不具备发送数据的能力，相比之下通过new Rx.Subject创建的观察对象实例具备发送数据源的能力。
   * 通过序列源实例可以订阅序列发射新数据变更时的响应方法（回调方法）
   * 响应的动作实际上就是Observable的执行
   * 通过序列源实例可以销毁，而当订阅方法发生错误时也会自动销毁。
   * 序列源实例的catch方法可以捕获订阅方法发生的错误，同时序列源实例可以接受从catch方法返回值，作为新的序列源实例
3. 掌握最简单的例子
4. // 5.0.0-rc.1
5. import Rx from 'rxjs';
6. //emit 1 from promise
7. const source = Rx.Observable.fromPromise(new Promise(resolve => resolve(1)));
8. //add 10 to the value
9. const example = source.map(val => val + 10);
10. //output: 11

const subscribe = example.subscribe(val => console.log(val));

**通过代码掌握Observable, Observer, Subscription, Operators, Subject和Schedulers之间的关系**

import Rx from 'rxjs';

/\*\*

Rx.Observable是Observable

Rx.Observable.create创建序列源source，创建source的方法有多个，比如of, from, fromPromise等

observer是Observer观察者，只有在Rx.Observable.create创建方法可以获取，其他创建方法内置了observer且不可访问

observer.next发射数据更新

source.map其中map就是Operators的其中一个方法，方法调用返回新的source1

source1.subscribe是订阅，即数据更新时的响应方法。同时返回订阅实例Subscription

subscription.next立即响应（不同于发射）静态数据，此时不会经过`Operators`处理

! Rx.Observable.create或者Rx.Subject.create创建的source不会自动关闭，其他方式则当检测到没有序列发生变更会自动销毁source.

\*/

const source = Rx.Observable.create(observer => {

observer.next('foo');

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

});

const source1 = source.map(val => `hello ${val}`);

const subscription = source1.subscribe(value => console.log(value));

subscription.next('foo1');

// forEach和subscribe相似，同是实现订阅效果，等到promise可以监控subscription完成和失败的异常。

// 日志打印并没有comlete, 因为source并没有完成关闭，触发调用observer.complete()

const promise = source1.forEach(value => console.log(value))

promise.then(() => console.log('complete'), (err) => console.log(err));

/\*\*

output:

hello foo

foo1

hello foo

hello bar

hello bar

\*/

/\*\*

new Subject创建被观察者实例，同source一样都具备subscribe方法，表示的含义和作用也一样，即发射数据变更时响应方法。

subject.next立即发射数据变更，作用同observer.next

注意foo1是最后输出的，是因为在创建source时指定了Rx.Scheduler.async，是异步的调度器，表示在响应数据处理时是异步执行的。

\*/

Rx.Observable.of('foo1', Rx.Scheduler.async).subscribe(value => console.log(value));

const subject = new Subject();

const source2 = subject.map(val => `hello ${val}`);

const subscription = source1.subscribe(value => console.log(value));

subject.next('foo');

subscription.next('bar');

/\*\*

output:

hello foo

bar

foo1

\*/

#### 1.2 学会看rxjs交互图

交互图中每条连表示一个数据序列，每个球表示每次发射的变更，最后一条线表示最终产出的数据序列。

下图以combineLastest来举例：

* 方法之上的每条线都是一个source(数据序列实例)
* 方法之下方法调用后返回的新source
* combineLastest表示被组合的每个source，一旦发射数据变更，必须拿到其余的source的最新值（当异步时则等待，直到都拿到最新值），组合为新的数据，作为新source发射的数据变更。

source1: ————————①——————————②——————————③————————————④—————————⑤——————————|——>

source2: ———————————ⓐ————————ⓑ————————————ⓒ—————————————————————ⓓ—————————|——>

combineLastest(source1, source2, (x, y) => x + y)

source: ———————(①ⓐ)—(②ⓐ)—(②ⓑ)—————(③ⓑ)—(③ⓒ)———(④ⓒ)————(⑤ⓒ)—(⑤ⓓ)——|——>

### 2. 实例方法Operators

前面讲过Operators方法调用时，接收的参数是source，返回新的source, 以下是个人学习使用过程中，简单总结的rxjs各方法用法。

#### 2.1 创建

* 发射完数据更新自动关闭：from, fromPromise, of, from, range
* 不发射直接关闭：empty
* 抛出异常后关闭：throw
* 不发射数据也不关闭：never
* 保持发射数据且不自动关闭：timer, interval, fromEvent
* 需要手动发射数据且不自动关闭：create, (还有Rx.Subject.create)

#### 2.2 转换

1. 1:1效果：map, mapTo, flatMap, scan, expand, pluck
   * map，source = source1.map(func)表示source1每次发射数据时经过func函数处理，返回新的值作为source发射的数据
   * mapTo，不同于map，func改为静态值
   * flatMap，当发射的数据是一个source时，在订阅的响应方法中接收到的也是一个source（这是合理的，发射什么数据就响应什么数据嘛，但是如果我们想在响应方法收到的是source的发射数据），flatMap就是可以允许发射数据是一个source，同时在响应的时候接收的是source的发送数据，后面我们称之为**source打平**
   * scan，source = source1.scan(func, initialValue), source每次发射的数据是source前次发射数据和source1当前发射的数据 的组合结果（取决于func，一般是相加), initialValue第一次发射，source前次没发射过，采用initialValue作为前次发射的数据
   * expand，和scan不同的是当func返回值是一个source时，在func接收到的数据是source打平后的发射数据。**特别适用于polling长轮询**
   * pluck，每次发射数据时，获取数据中的指定属性的值作为source的发射数据
2. 1:N效果：concat, concatAll, concatMap, concatMapTo, merge, mergeAll, mergeMap, mergeMapTo, switchMap, switchMapTo
   * concat, concatAll和merge, mergeAll属于组合类型，放在这讲更好体现其效果。
   * concat，source = source1.concat(source2)表示source发射数组的顺序是，当source1或source2发射数据，source就发射。但是只有当source1发射完且关闭(source1不在发送数据)后，才触发source2发射数据。
   * concatAll，不同于concat，会把所有的发射的数据打平（如果数据为source时），然后在决定下次发射哪个数据。
   * concatMap，source = source1.concatMap(source2)表示source1每次发射数据时，获取source2的所有发射数据，map返回多个待发射数据，按顺序发射第一个数据变更。
   * concatMapTo, 不同于concatMap, map处理以source2的数据为返回结果
   * switchMap, 和concatMap不同的是在map之后的待发射数据排序上，concatMap中source1每次发射时source2的所有发射数据都接收，作为source1下一次发射前，之间的所有发射数据。switchMap则会判断source2的所有发射数据是否有数据的发射时间比source1下一次发射的时间晚，找出来去除掉。
   * switchMapTo对switchMap就好比concatMap对concatMapTo, mergeMap对比mergeMapTo的关系也是如此。
   * mergeMap相比于switchMap，找出的数据会打平到source中，不丢弃。
3. N:1效果：buffer, bufferCount, bufferTime, bufferWhen
   * buffer，source = source1.buffer(source2)表示source1以source2为参考，在source2的2次发射数据之间为时间段，source才发射一次数据，数据为该时间段内source1本该发射的数据的组合。
   * 比如source1原先每隔1秒发射一次数据，source2是每个2秒发射数据，source = source1.buffer(source2), 那么source会每隔2秒发射数据（source1的2秒内发射的2个数值组成的数组）
   * bufferCount，source = source1.bufferCount(count, start), count表示source1毎3次发射数据作为source的一次发射数据，发射完后，以source1当前组合的发射数据的第start个开始算下次发射数据需要组合的起始数据。
   * bufferTime，一段时间内的source1发射数据作为source的一次发射数据
   * bufferWhen, 以默认结果为准分成2段，分别作为source的每次发射数据
4. 1:source效果：groupBy, window, windowCount, windowTime, windowWhen
   * groupBy, source = source1.groupBy(func), 表示source1的所有发射数据，按func分成多段，每段作为source的每次发送的数据（这里数据只是新的source，你可以理解为inner Observable实例)
   * window和buffer不同的时，source每次发送的是innerObservable
   * window vs windowCount vs windowTime vs windowWhen 同 buffer相似
5. 1:sources效果：partition
   * partition，sources = source1.partition(func), 根据func吧所有的source1发射数据分段，每段组成一个source，最终得到sources数组

#### 2.3 过滤

source的过滤不会对发射数据做任何改变，只是减少source的发射次数，所以理解起来会简单很多，这里只做个简单分类

* 防抖动（一段时间内只取最新数据作为一次发射数据，其他数据取消发射）：debounce, debounceTime, throttle(和debounce唯一区别是debounce取一段时间内最新的，而throttle忽略这段时间后，发现新值才发送）, throttleTime
* 去重（重叠的发射数据只去第一数据作为发射数据，其他相同数据取消发射）：distinct, distinctUntilChanged
* 定位（根据条件值去一个或部分几个数据作为对应发射数据，其他取消发射）：elementAt, first, last, filter, take, takeLatst, takeUntil, takeWhile,
* 跳过（根据条件去除符合条件的，取剩下的值作为每次发射数据）：skip, skipUntil, skipWhile, ignoreElements(忽略所有的，等同于empty)
* 样本：sample, source=source1.sample(source2), 以source2发射数据时来发现最新一次source1发射的数据，作为source的发射数据，个人觉得应该属于**转换**分类，官网放到了**过滤**

#### 2.4 组合

做个source组合成新的souce

* concat, concatAll和merge, mergeAll，在**转换**分类讲过了
* combineLastest，source = source1.combineLastest(source2, func)，source1和source2一旦发射数据，func会触发，拿到source1和source2最新的发射数据，返回新的数据，作为source的发射数据。
* combineAll，同combineLastest，，source = sources.combineAll()
* forkJoin，source = Rx.Observable.forkJoin(sources), 所有的sources都关闭后，获取各自最新的发射数组组合为数组，作为source的发射数据
* zip和forkJoin的区别是，zip是sources都有发送数据时，组合为一个数组作为source的发送数据，而sources任一source关闭了，则取source最后发射的数值。
* zipAll，同concat对concatAll
* startWith，source = source1.startWith(value), 表示在source1的最前面注入第一次发射数据
* withLastestFrom, soruce = source1.withLastestFrom(source2, func), 表示source1每次发射数据时，获取source2最新发射的数据，如果存在则func处理得到新的数组作为source的发射数据

#### 2.5 判断

* find和findIndex分别是指定发射数据和发射数据的下标（第几次发送的），应该放到**过滤**分类才合理
* isEmpty, every, include等，判断是否为真，判断的结果当做是source的发射数据

#### 2.6 错误处理

* catch，source在Operators调用过程中出现的异常，都可以在catch捕获到，同时可以返回新的source，因为出现异常的当前source会自动销毁掉。
* retry，source = source.retry(times), source的所有发射，重复来几遍。
* retryWhen，根据条件来决定来几遍，只有当条件为false时才跳出循环。

#### 2.7 工具

* do，在每次响应订阅前，可以通过source.do(func)，做一些提前处理等任何动作，比如打印一下发射的数据等。
* delay, delayWhen，每次发送数据时，都延迟一定时间间隔后再发送。
* observeOn, 设置scheduler,即发射数据的响应方式，Schedulers详细查看[地址](http://reactivex.io/rxjs/manual/overview.html#scheduler), 这里不讲解了，项目中应用得不多。
* subcribeOn, timeInterval设置sheduler
* toPromise, source转成promise，可以通过promise.then达到source.subscribe的效果
* toArray，把source所有发射的数据，组成数组输出。

#### 2.8 计算

把source的所有发射数据进行指定计算后，得出的数据作为新source的发射数据，计算方法分别有：max, min, count, reduce, average等

#### 2.9 其他

* cache, source = source1.cache(1);共享source1的订阅结果，即不管source订阅几回，响应方法接收到的发射数据都是同一份。
* 共享source订阅结果很重要，因为**组合**等方法组合多个source时，其中包含sourceA，同时sourceA还需要单独订阅其结果，在不用cache情况下，sourceA会产生2个subscription，即2个订阅实例，但是我们更希望是能达到sourceA发生变化时，都能通知到所有的组合sourceA的source。
* publish，publishSource = source.publish(),让source的订阅的工作延后，即source不会发射数据，而是等到publishSource.connect()调用后才开发发射数据。效果和delay很相似，不同的是可以控制合适发射。
* share，当source订阅多次，那么每次响应时do都会调用多次，通过share合并响应，则source发射一次数据更新，多次响应当当一次响应处理，do也调用一次。

### 参考资料

1. rxjs官网 - <http://reactivex.io/rxjs/>
2. rxjs代码 - <https://github.com/ReactiveX/rxjs>
3. 常用rxjs方法的交互图 - <http://rxmarbles.com/>
4. rxhjs教程 - <http://xgrommx.github.io/rx-book/content/observable/observable_instance_methods/toarray.html>
5. Scheduler - <https://mcxiaoke.gitbooks.io/rxdocs/content/Scheduler.html>