Reactive Functional Programming con RxJS

Chi sono?

Maksim Sinik

- Fullstack developer
- Software e Cloud Architect
- DevOps lover
- Functional Programming enthusiast

@maksimsinik

Agenda

1. Introduzione alla FRP
2. Che cos'è RxJS?
3. Composizione ed esecuzione di uno stream
4. Marble Diagrams
5. Obsevable Operators
6. Subjects
7. Higher Order Observables

1. Che cos'è la FRP?

1. Che cos'è la FRP?

Functional Programming + **Reactive** Programming

Functional Reactive Programming

FRP è un **paradigma di programmazione** che permette di gestire un **flusso di dati asincrono**, servendosi dei capisaldi della **programmazione funzionale** (ad esempio dei metodi `map`, `reduce`, `filter`). La FRP è molto usata nella programmazione di *interfacce grafiche*, in robotica e in musica e ha come scopo quello di rendere più semplice la modelazione degli **eventi che succedono nel tempo**.

Callback

"Una **funzione**, o un "blocco di codice" che viene passata come **parametro** ad un'altra funzione"

Callback

"Una **funzione**, o un "blocco di codice" che viene passata come **parametro** ad un'altra funzione"

async vs **sync**

Imperative vs Declarative (1)

Imperativo

Paradigma di programmazione secondo cui un programma viene inteso come un **insieme di istruzioni**, ciascuna delle quali può essere pensata come un "*ordine*" che viene impartito alla macchina virtuale del linguaggio di programmazione utilizzato.

Dichiarativo

Paradigma di programmazione che si focalizza sulla **descrizione** delle proprietà della soluzione desiderata (**il cosa**), lasciando indeterminato l'algoritmo da usare per trovare la soluzione (**il come**).

Imperative vs Declarative (2)

Imperativo

```
const doubled = []

const doubleMap = numbers => {
  for (let i = 0 i < numbers.length i++) {
    doubled.push(numbers[i] * 2)
  }
  return doubled
}

console.log(doubleMap([2, 3, 4])) // [4, 6, 8]
```

Imperative vs Declarative (2)

Imperativo

```
const doubled = []

const doubleMap = numbers => {
  for (let i = 0 i < numbers.length i++) {
    doubled.push(numbers[i] * 2)
  }
  return doubled
}

console.log(doubleMap([2, 3, 4])) // [4, 6, 8]
```

Dichiarativo (Funzionale)

```
const doubleMap = numbers => numbers.map(n => n * 2)

console.log(doubleMap([2, 3, 4])) // [4, 6, 8]
```

Functional Programming

Functional Programming

- è un paradigma di pogrammazione **dichiarativo**

Functional Programming

- è un paradigma di pogrammazione **dichiarativo**
- il programma è l'**esecuzione** di una serie **di funzioni**

Functional Programming

- è un paradigma di pogrammazione **dichiarativo**
- il programma è l'**esecuzione** di una serie **di funzioni**
- usa tipi di dato che **non sono mutabili**

Functional Programming

- è un paradigma di pogrammazione **dichiarativo**
- il programma è l'**esecuzione** di una serie **di funzioni**
- usa tipi di dato che **non sono mutabili**
- **non cambia il valore di variabili** definite fuori dallo scope in esecuzione (*funzioni pure*)

Functional Programming

- è un paradigma di pogrammazione **dichiarativo**
- il programma è l'**esecuzione** di una serie **di funzioni**
- usa tipi di dato che **non sono mutabili**
- **non cambia il valore di variabili** definite fuori dallo scope in esecuzione (*funzioni pure*)
- Ogni chiamata succesiva a **una stessa funzione**, con gli stessi argomenti, **produce lo stesso output**

Array methods(1)

```
const source = ['1', '1', 'foo', '2', '3', '5', 'bar', '8', '13']

const result = source
  .map(x => parseInt(x)) // applica la funzione parseInt a ogni elemento
  //.filter(x => !isNaN(x))
  //.reduce((x, y) => x + y)

console.log(result)

// logs: [1, 1, NaN, 2, 3, 5, NaN, 8, 13]
```

Array methods(2)

```
const source = ['1', '1', 'foo', '2', '3', '5', 'bar', '8', '13']

const result = source
  .map(x => parseInt(x))
  .filter(x => !isNaN(x)) // filtra solo quelli che sono numeri
  //.reduce((x, y) => x + y)

console.log(result)

// logs: [1, 1, 2, 3, 5, 8, 13]
```

Array methods(3)

```
const source = ['1', '1', 'foo', '2', '3', '5', 'bar', '8', '13']

const result = source
  .map(x => parseInt(x))
  .filter(x => !isNaN(x))
  .reduce((x, y) => x + y) // addiziona tutti gli elementi

console.log(result)

// logs: 33
```

2. Che cos'è RxJS?

2. Che cos'è RxJS?

ReactiveX è una **libreria** per comporre *programmi asincorni ed "event-based"*, usando una sequenza "**osservabile**" di **valori**.

Reactive Programming

"Programming with **asynchronous observables data streams**"

André Staltz

ReactiveX

http://reactivex.io/

- specifiche reactive
- implementazioni nei principiali linguaggi (> 15) e piattaforme
- noi ci occuperemo di **RxJS**

3. Composizione ed esecuzione di uno stream

Uno *stream* è composto da:

- `Observable`
- `Operators`
- `Subscription`

Uno *stream* è composto da:

- `Observable`
- `Operators`
- `Subscription`

Uno stream si basa, quindi, su **flussi di dati** emessi, sugli **operatori** per modificarli e gli **observer** che li ascoltano.

Observable

Observable

- **oggetto** (javascript) che possiede dei metodi (chiamati **operatori**) e delle proprietà

Observable

- **oggetto** (javascript) che possiede dei metodi (chiamati **operatori**) e delle proprietà
- emettitore di eventi o valori in un certo lasso di tempo

Observable

- **oggetto** (javascript) che possiede dei metodi (chiamati **operatori**) e delle proprietà
- emettitore di eventi o valori in un certo lasso di tempo
- creato a partire da strutture "simili" (isomorfe)

Observable

- **oggetto** (javascript) che possiede dei metodi (chiamati **operatori**) e delle proprietà
- emettitore di eventi o valori in un certo lasso di tempo
- creato a partire da strutture "simili" (isomorfe)

```
const numberStream = Observable.of(1, 2, 3, 4, 5, 6, 7, 8)
numberStream.subscribe({
  next(x) { console.log(x) },
  error(e) { console.error(e) },
  complete() { console.log('done') }
})
// => 1
// => 2
// => 3
// => 4
// => 5
// => 6
// => 7
// => 8
// => done
```

Operators

- sono **metodi** che si applicano a uno stream

- possono essere **concatenati** se è necessario applicare più di un operatore

- ogni **operatore modifica** lo stream dei valori emmessi dall'operatore che lo precede

- ogni operatore produce un **nuovo Observable** (stream)

Operators(2)

API prende spunto dai metodi degli array

```
const array$ = Observable.from([1, 2, 3, 4, 5, 6, 7, 8, 9])
array$
  .map(x => x * 2)
  .filter(x => x > 6)
  .subscribe(x => console.log(x))

// 8
// 10
// 12
// 14
// 16
// 18
```

Observer (o subscriber)

- Possiede **tre metodi**: prossimo elemento, errore, fine dello stream

```
const example$ = Observable.create(observer => observer.next(1))
const observer = {
  next: next => {
    console.log(next)
  },
  error: err => {
    console.log(err)
  },
  complete: () => {
    console.log('done')
  }
}
example$.subscribe(observer)
```

- E' un **Oggetto javascript** passato come argomento a `.subscribe()`

Esecuzione di uno stream

- Comicncia solo quando **invochiamo** il methodo `.subscribe(observer)`

Esecuzione di uno stream

- Comicncia solo quando **invochiamo** il methodo `.subscribe(observer)`
- Vengono eseguiti in **sequenza** ordinata tutti gli operatori

Esecuzione di uno stream

- Comicncia solo quando **invochiamo** il methodo `.subscribe(observer)`
- Vengono eseguiti in **sequenza** ordinata tutti gli operatori
- Ogni operatore torna un **nuovo Observable**

Esecuzione di uno stream

- Comicncia solo quando **invochiamo** il methodo `.subscribe(observer)`
- Vengono eseguiti in **sequenza** ordinata tutti gli operatori
- Ogni operatore torna un **nuovo Observable**
  - Eccetto `.subscribe()`!

Unsubscribe

```
const randomNumber$ = Observable.create((observer) => {
  const id = setInterval(() => {
    observer.next(Math.random())
  }, 500)
  return () => clearInterval(id)
})

const sub = randomNumber$.subscribe({
  next(x) { console.log(x) },
  error(e) { console.error(e) },
  complete() { console.log('done') }
})

setTimeout(() => {
  sub.unsubscribe()
}, 2000)

// 0.10430196667680214
// 0.4141351814554881
// 0.5761438321958294
```

Più di una subscription

```
const array$ = Observable.from([1, 2, 3, 4, 5, 6, 7, 8, 9])
  .map(x => x * 2)
  .filter(x => x > 6)

array$
  .subscribe(x => console.log(`First: ${x}`))
array$
  .subscribe(x => console.log(`Second: ${x}`))

// First: 8
// First: 10
// First: 12
// First: 14
// First: 16
// First: 18
// Second: 8
// Second: 10
// Second: 12
// Second: 14
// Second: 16
// Second: 18
```

Gli stream si possono comporre

```
const array$ = Observable.from([1, 2, 3, 4])

const array2$ = Observable.from([ 5, 6, 7, 8, 9])

array$
  .merge(array2$)
  .subscribe(x => console.log(`First: ${x}`))

// First: 1
// First: 2
// First: 3
// First: 4
// First: 5
// First: 6
// First: 7
// First: 8
// First: 9
```

Gli stream si possono anche spezzare

```
const array$ = Observable.from([1, 2, 3, 4, 5, 6, 7, 8, 9])

const [evens, odds] = array$.partition(val => val % 2 === 0)

const subscribe = Observable.merge(
  evens
    .map(val => `Even: ${val}`),
  odds
    .map(val => `Odd: ${val}`)
  ).subscribe(val => console.log(val))
```

HOT VS COLD

- **COLD**: Observable che cominciano l'esecuzione dopo che viene invocato il metodo subscribe
- **HOT**: Observable che producono i valori anche prima che ci sia una subscription attiva in ascolto

Cold

A ogni `.subscribe()` corrisponde una nuova esecuzione.

Cold

A ogni `.subscribe()` corrisponde una nuova esecuzione.

```
const obs = Observable.create(observer => {
  observer.next(1)
  setTimeout(() => {
    observer.next(2)
    setTimeout(() => {
      observer.complete()
    }, 1000)
  }, 1000)
})

obs.subscribe(
  v => console.log("1st subscriber: " + v),
  err => console.log("1st subscriber error: " + err),
  () => console.log("1st subscriber complete ")
)
setTimeout(() => {
  obs.subscribe(
    v => console.log("2nd subscriber: " + v),
    err => console.log("2nd subscriber error: " + err),
    () => console.log("2nd subscriber complete ")
  )
}, 2000)
```

Hot

L'esecuzione è condivisa tra tutti gli observer.

```
const obs = Observable
  .interval(1000)
  .publish()

obs.connect() // avvia l'esecuzione, è come chiamare .subscribe su un cold

setTimeout(() => {
  obs.subscribe(v => console.log("1:" + v))
  setTimeout(
    () => obs.subscribe(v => console.log("2:" + v)), 1000)
}, 2000)

// 1:1
// 1:2
// 2:2
// 1:3
// 2:3
// 1:4
// 2:4
// ...
```

4. Marble Diagram

Marble Diagram

Marble Diagram

I marble diagram sono il modo che abbiamo di **rappresentare in modo visivo** gli stream reattivi di dati (asincroni).

Marble Diagram

I marble diagram sono il modo che abbiamo di **rappresentare in modo visivo** gli stream reattivi di dati (asincroni).

In un marble diagram, **l'asse X rappresenta il tempo**.

Marble Diagram

I marble diagram sono il modo che abbiamo di **rappresentare in modo visivo** gli stream reattivi di dati (asincroni).

In un marble diagram, **l'asse X rappresenta il tempo**.

L' asse Y invece rappresenta i diversi observable che interagiscono tra di loro, anche tramite operatori.

Marble diagrams (ASCII form)

Marble diagrams (ASCII form)

- Il **tempo** è rappresentato da: ------

Marble diagrams (ASCII form)

- Il **tempo** è rappresentato da: ------
- I **valori** sono rappresentati da: [0-9] oppure [a-z]

Marble diagrams (ASCII form)

- Il **tempo** è rappresentato da: ------
- I **valori** sono rappresentati da: [0-9] oppure [a-z]
- Il **completamento** è rappresentato da: |

Marble diagrams (ASCII form)

- Il **tempo** è rappresentato da: ------
- I **valori** sono rappresentati da: [0-9] oppure [a-z]
- Il **completamento** è rappresentato da: |
- L'eccezione è rappresentata da: X

Marble Diagram di map

```
first:  ---0---1---2---3-|

operator:  map( x => x * 2)

second: ---0---2---4---6-|
```

- *First* è lo stream in **input**
- *Second* è lo stream in **output**
- *Operator* indica il **metodo applicato**

5. Obsevable Operators

Catergorie di Operatori

- Creation
- Transformation
- Filtering
- Combination
- Multicasting
- Error Handling
- Utility
- Conditional and Boolean
- Mathematical and Aggregate

***9 categorie** diverse e più di **120 operatori** totali*

120?????

Creation Operators

create

```
create(subscribe: (observer) => subscription): Observable<T>
```

create

```
create(subscribe: (observer) => subscription): Observable<T>
```

- Crea un Observable dalla funzione subscribe, passata come parametro.

create

```
create(subscribe: (observer) => subscription): Observable<T>
```

- Crea un Observable dalla funzione subscribe, passata come parametro.
- Alias del costruttore

from

```
from(ish: ObservableLike<T>): Observable<T>
```

from

```
from(ish: ObservableLike<T>): Observable<T>
```

- Facade per creare Observable

from

`from(ish: ObservableLike<T>): Observable<T>`

- Facade per creare Observable
  - Array

from

`from(ish: ObservableLike<T>): Observable<T>`

- Facade per creare Observable
  - Array
  - array-like object

from

`from(ish: ObservableLike<T>): Observable<T>`

- Facade per creare Observable
  - Array
  - array-like object
  - Promise

from

```
from(ish: ObservableLike<T>): Observable<T>
```

- Facade per creare Observable
  - Array
  - array-like object
  - Promise
  - iterabli

from

`from(ish: ObservableLike<T>): Observable<T>`

- Facade per creare Observable
  - Array
  - array-like object
  - Promise
  - iterabli
  - Observable

fromPromise e fromEvent

fromPromise e fromEvent

- Crea un observable a partire da una Promise.

```
fromPromise(promise: Promise): Observable
```

fromPromise e fromEvent

- Crea un observable a partire da una Promise.

```
fromPromise(promise: Promise): Observable
```

- Crea un observable a partire da un evento.

```
fromEvent(target: EventTargetLike, eventName: string): Observable
```

interval

```
interval(period: number): Observable
```

interval

```
interval(period: number): Observable
```

Crea un Observable che emmette valori ogni 'period' (ms)

of

`of(...values): Observable`

of

`of(...values): Observable`

Crea un Observable i cui valori sono i suoi argomenti e completa immediatamente.

Operators

Operatori Famosi

```
//Combination
.concat
.concatAll
.merge
.mergeAll
.zip

// Transformation
.scan
.buffer
.map/.mapTo
.groupBy
.mergeMap
.switchMap

// Filtering
.filter
.first/.last
.take
.skip
.throttle
```

concat

`concat(observables: ...*): Observable`

Al **completamento** del primo observable, viene **concatenato il secondo**, al quale viene passato lo stesso subscribe.

```
first   --0--1--2--3|
second (a|)
          first.concat(second)
result  a-0--1--2--3|
```

merge

`merge(input: Observable): Observable`

Unisce più Observable in un unico Observable.

```
first:  ----0----1----2----(3|)
second: --0--1--2--3--(4|)
            first.merge(second)
result: --0-01--21-3--(24)-(3|)
```

zip

`zip(observables: *): Observable`

Zip sottoscrive tutti gli observable che gli vengono passati come argomenti. Appena viene emesso un valore da ognuno di loro, zip **emette un array contentente i valori** emessi dagli singoli observable.

```
first:  ----0----1----2----(3|)

second: --0--1--2--3--(4|)

            Observable.zip(first, second)

result: ----00---11---22----33|
```

switch

`switch(): Observable`

Fa in modo che il subscribe sia fatto solo sul Observable emesso come ultimo.

```
first:  -0--1--2-..
second: ---------1-2-3-4-5
            switch()
result: ---------1-2-3-4-5
```

map (mapTo)

`map(project: Function, thisArg: any): Observable`

Applica la funzione a ogni valore emesso dall'Observable.

```
first:  ---0---1---2---3-|
operator:  map( x => x * 2)
second: ---0---2---4---6-|
```

filter

`filter(select: Function, thisArg: any): Observable`

Emette solo i valori che passano la condizione.

```
first:  --0--1--2--3--4--5--6--7-

          filter(x => x % 2 === 0)

result: --0-----2-----4-----6----
```

Altri operatori di filtering

Altri operatori di filtering

```
take(count: number): Observable
```

Altri operatori di filtering

```
take(count: number): Observable
```

```
first(predicate: function, sel: function)
```

Altri operatori di filtering

```
take(count: number): Observable

first(predicate: function, sel: function)

skip(the: Number): Observable
```

Altri operatori di filtering

```
take(count: number): Observable

first(predicate: function, sel: function)

skip(the: Number): Observable

last(predicate: function): Observable
```

debounce

`debounce(durationSelector: function): Observable`

Scarta i valori che sono stati emessi in un tempo minore rispetto a quello specificato.

```
--0--1--2--3--4|

  debounceTime(1000) // simile a debounce

------------------------4|
```

throttle

`throttle(duration: function(value): Observable | Promise): Observable`

Emette valori solo quando è passata la durata specificata.

```
--0--1--2--3--4|
  throttleTime(1000)
--0-----2-----4|
```

scan

`scan(accumulator: function, seed: any): Observable`

Riduce i valori emessi nel tempo fino a che non viene emesso il complete.

```
-----h-----e-----l-----l-----o|

  scan((acc, x) => acc+x, '')

-----h-----(he)--(hel)-(hell)(hello|)
```

buffer

`buffer(closingNotifier: Observable): Observable`

Aggrega i valori emesssi finchè l'observable passato come argomento non emette. Emette un array.

```
-----h-----e-----l-----l-----o|

        bufferCount(2)

-----------he----------ll----o|
```

do

```
do(nextOrObserver: function, error: function, complete: function):
Observable
```

Esegue le funzioni passate come argomenti senza modificare in alcun modo l'observable in ingresso.

```
---0---1---2---3--...

  do(x => console.log(x))

---0---1---2---3--...
```

share

```
share(): Observable
```

Condivide l'Observable sorgente con più subscriber.

share

```
share(): Observable
```

Condivide l'Observable sorgente con più subscriber.

è uguale a `publish().refCount()`

share

`share(): Observable`

Condivide l'Observable sorgente con più subscriber.

è uguale a `publish().refCount()`

E' un multicast operator.

6. Subjects

Subject

- E' un observable: possiede tutti gli operatori
- E' un observer: quando viene usato come subscription emette i valori che gli vengono passati nel next
- Può usare il multicast: se passato nel subscribe viene aggiunto alla lista di observer
- Quando è completo, in errore o non più subscribed non può più essere usato
- Può passare valori a se stesso chiamando la sua funzione next

Subject vs Observable

La differenza più grande tra Subject e Observable è che il Subject ha uno stato interno: salva la lista degli observers.

Subject vs Observable

La differenza più grande tra Subject e Observable è che il Subject ha uno stato interno: salva la lista degli observers.

```
const tick$ = Observable.interval(1000);
const subject = new Subject();

subject.subscribe(observer1);
subject.subscribe(observer2);

tick$.subscribe(subject);
```

Subject vs Observable

La differenza più grande tra Subject e Observable è che il Subject ha uno stato interno: salva la lista degli observers.

```
const tick$ = Observable.interval(1000);
const subject = new Subject();

subject.subscribe(observer1);
subject.subscribe(observer2);

tick$.subscribe(subject);
```

In questo esempio vediamo che tick$ viene multicastato in due observer distinti. Questo è l'uso primario che ha il Subject in Rx.

Il **Subject** è, quindi, un **proxy/bridge**.

BehaviorSubject (the current value)

- Rappresenta valori che cambiano nel tempo
- Ogni BehaviorSubject ha un valore iniziale oppure l'ultimo valore emesso

BehaviorSubject (the current value)

- Rappresenta valori che cambiano nel tempo
- Ogni BehaviorSubject ha un valore iniziale oppure l'ultimo valore emesso

Nei **Service** di Angular si usa spesso il behavior subject per la gestione dei dati. Infatti, il servizio spesso si inzializza prima del component e il behavior subject ci garantisce che ci sarà un valore inziiale che poi verrà aggiornato appena ce ne sarà disponbile uno più recente.

7. Higher Order Observables

Higher Order Observables

Higher Order Observables

```
const numObservable = Rx.Observable.interval(1000).take(4)

const higherOrderObservable = numObservable
  .map(x => Rx.Observable.of(1,2))

higherOrderObservable
  .subscribe(obs =>s
    obs.subscribe(x => console.log(x))
  )
```

Higher Order Observables

```
const numObservable = Rx.Observable.interval(1000).take(4)

const higherOrderObservable = numObservable
  .map(x => Rx.Observable.of(1,2))

higherOrderObservable
  .subscribe(obs =>s
    obs.subscribe(x => console.log(x))
  )
```

```
const clickObservable = Rx.Observable
  .fromEvent(document, 'click')

const clockObservable = clickObservable
  .map(click => Rx.Observable.interval(1000))

clockObservable
  .subscribe(clock =>
    clock.subscribe(x => console.log(x))
  )
```

Flattening operators

- Si applicano ad **Observable di Observable**
- Tornano i **valori** dell'Observable interno, rispettandone il tipo

Flattening operators

- Si applicano ad **Observable di Observable**
- Tornano i **valori** dell'Observable interno, rispettandone il tipo

```
in:     Observable<Observable<number>>
method:       flatten()
out:    Observable<number>
```

Esempio con switch

```
const clickObservable = Rx.Observable
  .fromEvent(document, 'click')

const clockObservable = clickObservable
  .map(click => Rx.Observable.interval(1000))
  .switch()

// a: --------+--------+------------------------
//             \        \
//              -0-1-2-3 -0-1-2-3-4-5-6

//           switch(a, b)

// b: ----------0-1-2-3--0-1-2-3-4-5-6

clockObservable
  .subscribe(x => console.log(x))
```

Esempio con mergeAll

```
const clickObservable = Rx.Observable
  .fromEvent(document, 'click')

const clockObservable = clickObservable
  .map(click => Rx.Observable.interval(1000))
  .mergeAll(3)

// --------+--------+------------------------
//          \        \
//           -0-1-2-3 -0-1-2-3-4-5-6

//           mergeAll

// ----------0-1-2-3-405162738495...

clockObservable
  .subscribe(x => console.log(x))
```

Esempio con concatAll

```
const clickObservable = Rx.Observable
  .fromEvent(document, 'click')

const clockObservable = clickObservable
  .map(click => Rx.Observable.interval(1000).take(5))
  .concatAll() // megeAll(1)

// --------+---------------+-+----
//          \
//           -0-1-2-3-4|

//           concatAll

// ----------0-1-2-3-4-----0-1-2-3-4--0-1-2-3-4

clockObservable
  .subscribe(x => console.log(x))
```

Operatori composti

`.map()` + `.concatAll()` = `.concatMap()`

Operatori composti

`.map()` + `.concatAll()` = `.concatMap()`

`.map()` + `.mergeAll()` = `.mergeMap()` // (`flatMap`)

Operatori composti

`.map()` + `.concatAll()` = `.concatMap()`

`.map()` + `.mergeAll()` = `.mergeMap()` // (`flatMap`)

`.map()` + `.switch()` = `.switchMap()`

switchMap

```
const clickObservable = Rx.Observable
  .fromEvent(document, 'click')

function performRequest() {
  return fetch('https://jsonplaceholder.typicode.com/users/1')
    .then(res => res.json())
}

// Observable<Event> ---> Observable<Response>
const responseObservable = clickObservable
  .switchMap(click => performRequest())

// switchMap = map ... + ... switch

responseObservable
  .subscribe(x => console.log(x.email))
```

mergeMap

```
const clickObservable = Rx.Observable
  .fromEvent(document, 'click')

function performRequest() {
  return fetch('https://jsonplaceholder.typicode.com/users/1')
    .then(res => res.json())
}

const emailObservable = clickObservable
  .mergeMap(click => performRequest(),
            (click, res) => res.email,
            3)

// mergeMap = map ... + ... mergeAll

emailObservable
  .subscribe(email => console.log(email))
```

concatMap

```
const clickObservable = Rx.Observable
  .fromEvent(document, 'click')

function performRequest() {
  return fetch('https://jsonplaceholder.typicode.com/users/1')
    .then(res => res.json())
}

const emailObservable = clickObservable
  .concatMap(click => performRequest(),
             (click, res) => res.email)

// concatMap = map ... + ... concatAll

emailObservable
  .subscribe(email => console.log(email))
```

Esempio di typeahead

```
const obs1 = Observable.fromEvent(input, 'keyup')
    .map(e => e.target.value)
    .filter(value => value.length > 2)
    .distinctUntilChanged()
    .debounceTime(500)
    .mergeMap(word => this.http.get('...')) // Angular 2 http observable
    .retry(2)
    .subscribe(res => console.log(res))
```

Grazie!