

RXSwift

Is it really for **Aliens**?

About me

Stefano Mondino



iOS Technical Leader @ Synesthesia

Contatti

stefano.mondino.dev@gmail.com

stefano.mondino@synesthesia.it

github: stefanomondino

twitter: @puntoste

Menu del giorno

- RxSwift : a cosa serve?
- Conoscere l'Observable
- Manipolazione e combinazione di observables
- Trasformazione del proprio codice in RX
- Cenni di MVVM
- Pro e contro
- Esempi pratici

Rx

Reactive Extensions

Progetto di ReactiveX

Sintassi di programmazione reattiva comune a più
linguaggi

es: RxJava, RxJS, Rx.NET

FRP

Functional Reactive Programming

Programmazione di flussi di valori nel tempo

Limitazione delle variabili di stato

Migliore astrazione e riutilizzo del codice

Rx + RxSwift

Vantaggi di linguaggio:

- Closures: *first class citizens*
- Generics<T>
- Estensioni di classi
- Protocol Oriented Programming

Observable

Oggetto principale

Racchiude il lavoro da eseguire e le callbacks da invocare

Può emettere 0, 1 o più valori o emettere UN (eventuale) errore.

Non esegue NULLA fino a quando *non esplicitamente richiesto*

Subscription

Dichiarazione di interesse

- Metodo `subscribe()` attiva l'observable che inizierà a emettere
- Callback con ogni tipologia di evento (`next`, `error`, `complete`)
- `subscribe()` restituisce un oggetto di tipo `Disposable`

Disposable

E' un riferimento a una subscription Può essere annullata con il metodo `dispose()` La callback definita nella subscription non verrà mai più invocata L'`Observable` può eventualmente annullare il lavoro in corso

Esempio : l'antifurto

Sistema complesso composto da più sensori

Si attiva esplicitamente uscendo di casa

Se è attivo E rileva un intrusione, suona

- Antifurto -> Observable
- Attivazione dell'antifurto -> subscribe()
- Intrusione e allarme che suona -> onNext()
- Chiamata di emergenza -> *side effect*
- Tentativo di manomissione -> onError() da gestire
- Non c'è onComplete()
- Disattivazione dell'antifurto -> dispose()

Operatori (introduzione)

Funzioni che applicate su un Observable che ne restituiscono una versione "arricchita"

Live demo: RXMarbles.com

```
.map { input in return output(input) }
```

```
.filter { input in return  
isValid(input) }
```

Threading

È possibile specificare i thread su cui eseguire le operazioni e/o ricevere i risultati

Scheduler : oggetto che rappresenta una coda di esecuzione

Operatori subscribeOn (poco usato) e observeOn

Esempio:

.observeOn(MainScheduler.instance)

Observable semplici

Utili per convertire valori semplici in observable

- `.just(value)` -> Alla subscription, restituisce immediatamente value e completa
- `.empty()` -> Alla subscription, completa immediatamente
- `.error(error)` -> Alla subscription, emette un errore e completa

Esempio pratico

Chiamata di rete Rx

API JSON di **TVMaze** (servizio per info su serie tv)

Effettuare chiamata di rete nativa al servizio
schedule e mostrare il risultato in una `UILabel`

Gestire correttamente i thread

Soluzione :

```
URLSession.shared.rx
    .json(url: URL(string:"https://api.tvmaze.com/schedule"))
    .subscribeOn(MainScheduler.instance)
    .subscribe(onNext:{[weak self] json in
        let array = json as? [[String:Any]]
        if let first = array?.first,
            let title = first["name"] as? String
        {
            self?.lbl_text.text = "Next episode is \(title)"
        }
    })
    .disposed(by: self.disposeBag)
```

Operatori

A.k.a. : la vera ragione di tutto questo

map

Trasforma **ogni** input applicando logica indicata in una closure



`map(x => 10 * x)`



L'output può avere tipo diverso dall'input Esempi :

- da stringa a stringa maiuscola
- da data a stringa (data formattata)
- da numero a stringa (o viceversa)

filter

Condiziona l'invio di ogni next in base a una condizione booleana



```
filter(x => x > 10)
```



L'elemento viene propagato solo se la condizione è verificata

Esempi :

- filtrare solo gli Int minori di 10
- filtrare le stringhe con almeno 3 caratteri
- filtrare gli elementi non nulli

distinctUntilChanged()

Propaga l'elemento solo se è diverso dal precedente



distinctUntilChanged



take(n)

Propaga i primi n elementi e poi completa



take(2)



skip(n)

Ignora i primi n elementi



skip(2)



delay(t)

Ritarda l'invio di ogni elemento di t secondi.



delay

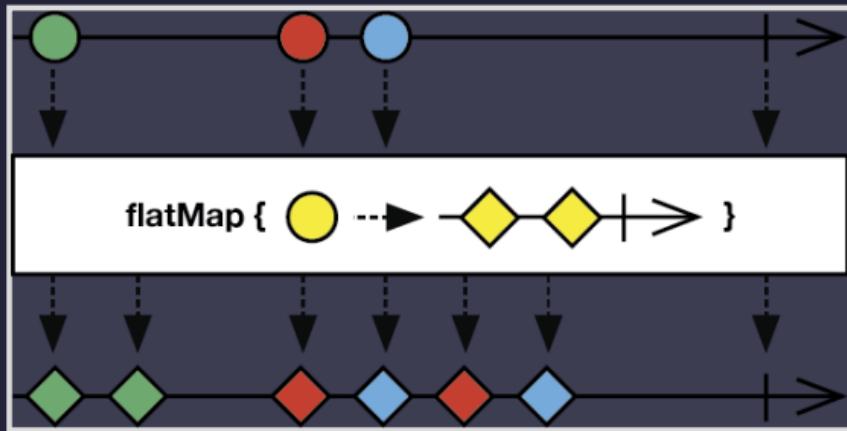


flatMap

Come `map`, ma per ciascun `next` restituisce un `Observable` (e non un elemento)

Per ogni `Observable` risultante viene creata una sottoscrizione automatica

Il risultato, dal punto di vista dell'observer, è un flusso *piatto* di elementi provenienti da ciascun observable generato.



Caso pratico:

1. Chiamata di rete di login. In next l'accessToken

```
func login(user:String,  
password:String) -> Observable<String>
```

2. Chiamata di rete (sotto autenticazione) per
ottenere una lista di dati.

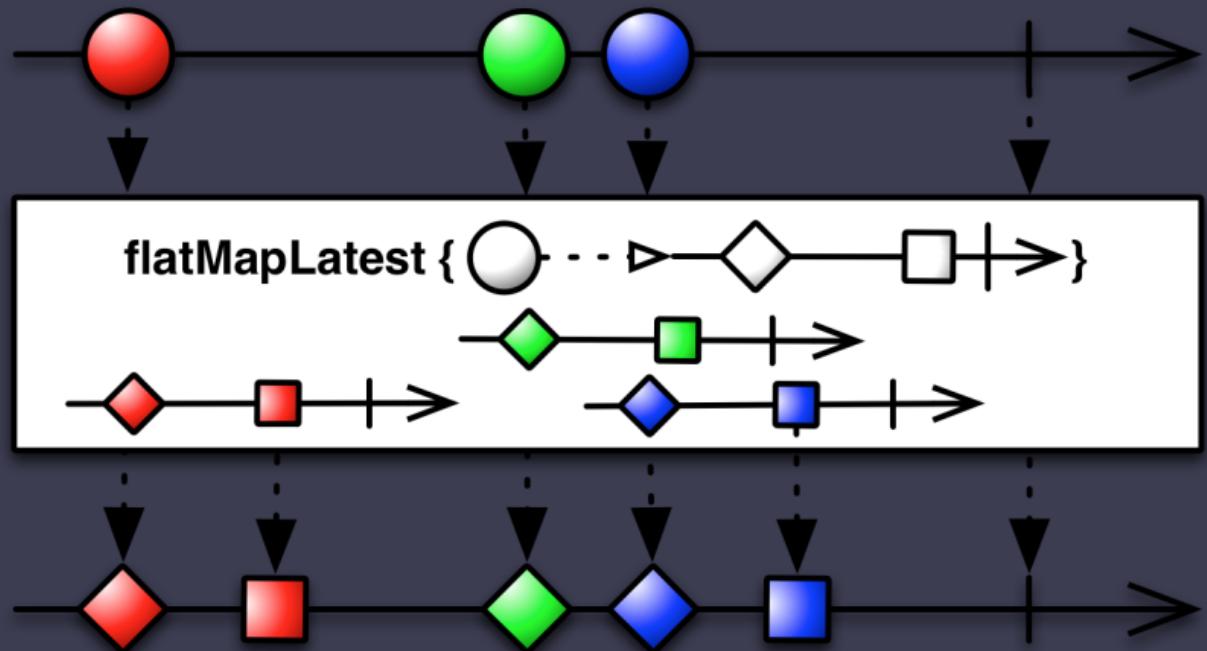
```
func getList(with token:String) ->  
Observable<[Any]>
```

Soluzione :

```
let observable = login(user:"user", password:"pwd")
    .flatMap { token in
        return getList(with:token)
    }
```

flatMapLatest

Come flatMap, ma a fronte di un next vengono
annullate tutte le precedenti subscription



Esempio:

Autocomplete in un campo di ricerca: per ogni modifica in un campo di testo parte una chiamata di rete. Se il contenuto del campo di testo cambia, le chiamate precedenti eventualmente in corso vengono annullate.

debounce(t)

Invia l'elemento corrente solo se nei t secondi successivi non ne arriva un altro.



debounce

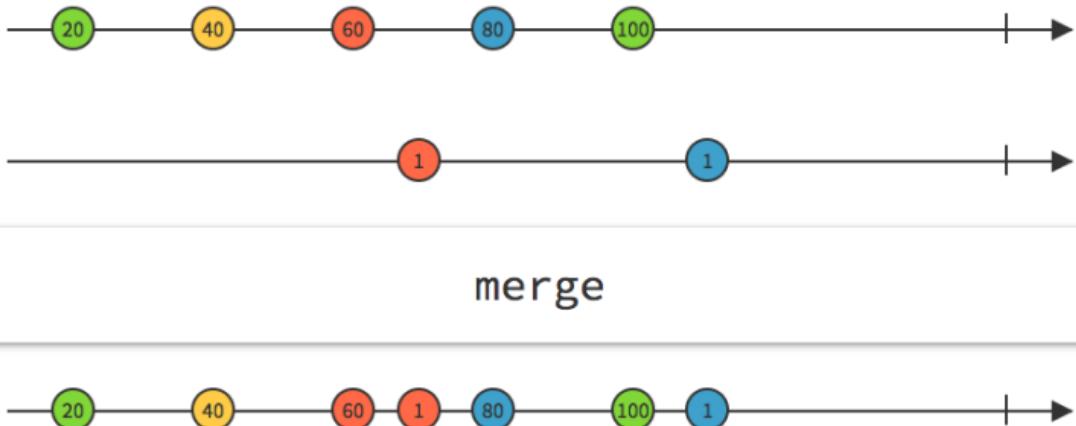


Esempio:

API di ricerca collegata a textfield: evita che vengano avviate chiamate superflue mentre l'utente sta ancora digitando.

merge

Lavora su un observable di observables





```
let toMerge:Observable<Observable<Any>>  
    = ...
```

Il risultato è un observable che propaga in next
ogni elemento di ciascun observable interno

Caso d'uso : stessa azione da eseguire al verificarsi di più di un evento, indipendentemente dall'evento

Esempio: emettere un suono ogni volta che scuoto il telefono O premo un pulsante O modifco un testo.

In output ho l'ultimo valore utile, non riesco a distinguerne la provenienza

combineLatest



```
combineLatest((x, y) => "" + x + y)
```



Simile a merge, ma :

- non emette valori fino a quando tutti gli observables interni non hanno emesso il primo next
- in output ha un array (o una tupla) con l'elenco ordinato degli ultimi next emessi

zip

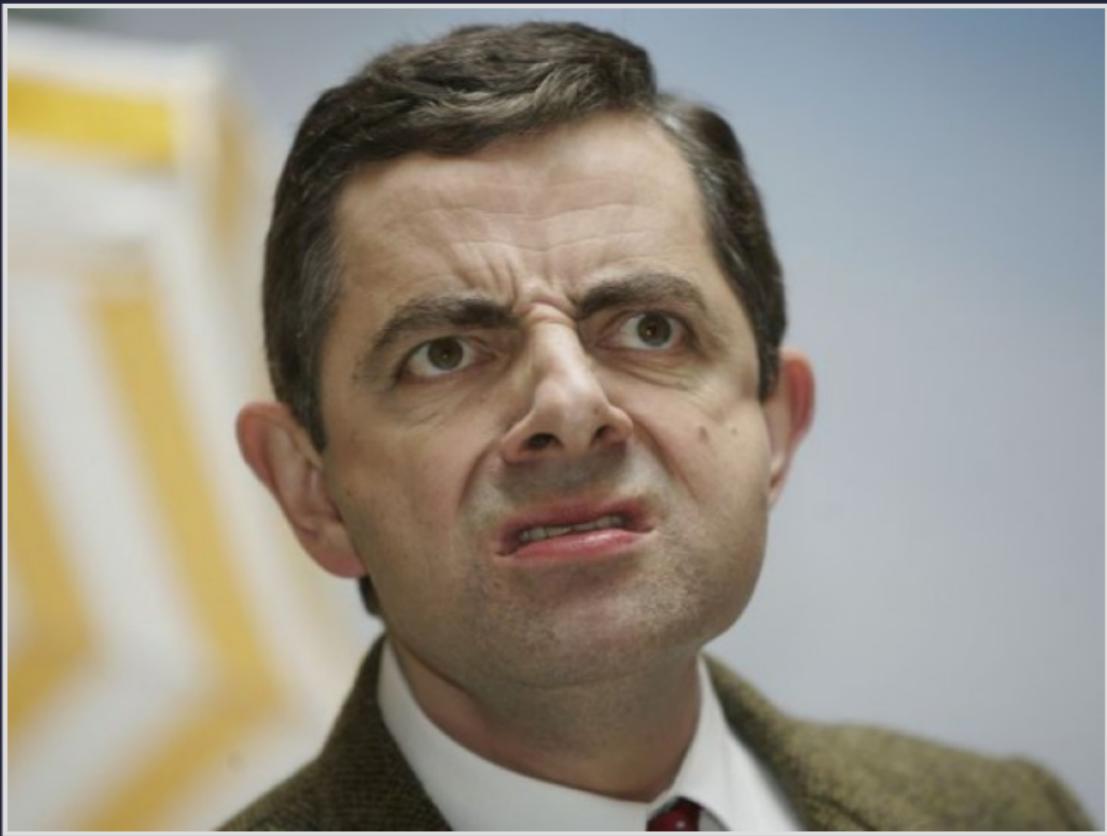


zip



Emette in next tuple di oggetti (come `combineLatest`), ma ciascun membro della tupla è l'n-esima emissione dei rispettivi observables.

In altre parole, il primo next è l'unione di tutti i primi next degli observables sottostanti.





concat

Concatena una lista di Observable

La subscription al successivo avviene solo al complete del precedente

Ogni next interno viene propagato all'observer

Caso d'uso : downloader di oggetti

toArray

Accumula in un array **TUTTI** i valori in next fino al complete.

All'observer arriva un unico next con l'array di tutti i next interni

Caso d'uso : concat di più chiamate e unico array di risultati.

RxCocoa

API per i componenti Apple nativi

Quasi tutti gli oggetti acquisiscono una variabile
 `.rx`

Al suo interno ci sono proprietà osservabili o
 "osservanti" (*bindable*)

Esempi:

```
let someTextObservable:Observable<String> = ...  
someTextObservable  
    .bind(to:label.rx.text)  
    .disposed(by:disposeBag)
```

```
let observable = textField.rx  
    .text  
    .asObservable()  
    .flatMapLatest { someQueryApi(with:$0) }
```

Objc runtime

Tramite il runtime di Objective-C è possibile osservare keypaths e chiamate a metodi.

Esempio :

```
viewController.rx  
.methodInvoked(#selector(UIViewController.viewDidAppear(_:)))
```

Variables

Sono oggetti che contengono valori osservabili
`(.asObservable())`.

Accesso diretto in lettura e scrittura in qualunque momento
`(.value)`

Molto utili per la gestione di form

Esempio:

```
let variableString = Variable<String>("Test")
variableString.asObservable().subscribe(onNext:{print ($0)}).disp
//in console : "Test"
...
// da qualche altra parte nel codice

variableString.value = "Change!"
//in console : "Change!"
```

Data binding

Associazione di una proprietà a un observable

Ogni `next` si riflette sulla proprietà

- metodo `Observable.bind(to:)`

Esempio:

```
let v = Variable("")  
let textField = UITextField()  
textField.rx.text.bind(to:v)
```

Two-way data binding

Associazione vicendevole di due proprietà osservabili

Nell'esempio precedente, se cambia la variable, il cambiamento non si rispecchia sul text field

Problemi di deadlock : A aggiorna B che riaggiorina A ecc...

Soluzione:

Operatore <-> come extension

Il risultato è un Disposable (come subscribe())

```
self.txt_field.rx.textInput <->  
    variable
```

Esempio Pratico

Form

Creare un form di login : 2 textfield e 1 pulsante Il pulsante deve essere abilitato solo se :

- username ha almeno 5 caratteri
- password ha almeno 8 caratteri

Creazione di observable

Metodo `.create` fornisce una closure che :

- come input ha l'observer a cui inviare gli eventi (`next,error e complete`)
- come output deve restituire un `Disposable`

All'interno della closure, si implementa il "lavoro" da svolgere

Solitamente, sono API di framework o librerie che a loro volta forniscono callback di completamento o errore

Al loro interno, invocare
`observer.onNext(value)` e simili

In output, restituire un disposable che si occupi di
annullare l'operazione iniziata

```
Disposables.create {  
    //operation è un ipotetico riferimento all'operazione in corso  
    operation.cancel()  
}
```

Condivisione di observable

Ogni sottoscrizione implica l'esecuzione del blocco
di osservazione.

In alcuni casi, questo può essere un effetto
indesiderato.

Esempio :

Esempio

```
let o = Observable<String>.create { observer -> Disposable
    operation = someAsyncOperation.getString {
        (result:String? , error:Error?) in
        if let error = error { observer.onError(error) }
        else {
            if let result = result { observer.onNext(result) }
            observer.onCompleted()
        }
    }
    return Disposables.create { operation.cancel() }
}
```

Side effects

Effettuano operazioni aggiuntive al di fuori della catena di eventi

La closure non ha valore di ritorno

Esempi :

```
.do(onNext:{ print $0 })
.do(onSubscribed:{ print "Subscribed!" })
.do(onComplete:{ print "Finished!" })
.do(onDispose:{ print "Cancelled!" })
```

Gestione errori

Se un observable va in errore, l'intera catena viene interrotta.

Come nei normali `try/catch`, si possono gestire le "eccezioni".

`catch` : prevede una closure di recupero che converte l'errore in un nuovo observable.

`catchErrorJustReturn(value)` : se arriva un errore, ritorna value al suo posto

Utile nella concatenazione/unione di più observable

Esempio Pratico

RxLocationManager

- Observable che in next fornisca ogni aggiornamento di posizione dell'utente
- Deve chiamare startUpdatingLocation() e stopUpdatingLocation() automaticamente
- Suggerimento : metodo do() per i side effects

RxGeocoder

- Costruire un observable intorno a CLGeocoder
- Estendere l'oggetto CLLocation con un Observable<CLPlacemark> che fornisce le informazioni testuali sulla posizione.

Considerazioni

Pro

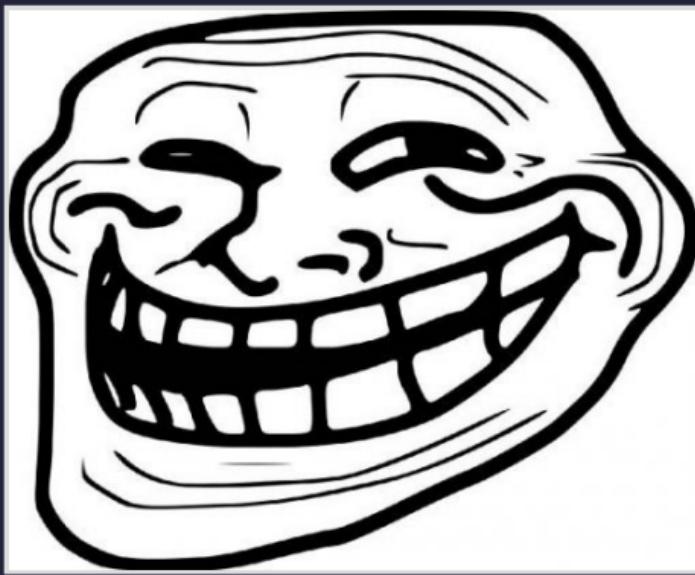
- Riorganizzazione del codice in singoli "blocchi"
- Riutilizzo tra più progetti
- Migliore testabilità

Contro

- Framework molto "ingombrante"
- È una libreria : potrebbe avere conflitti con altri SDK
- Curva di apprendimento ripida

Alternative : ReactiveSwift

Alla fine, è uguale



- Observable = Signal
- Signal ha due elementi generici (), è più scomodo
- Non ha una visione comune cross-platform

Pattern di sviluppo Rx

MVVM

- Alternativa a MVC (*Massive View Controller*)
- Praticamente inutile senza data-binding
- Sfrutta appieno RxSwift

Funzionamento



Progetto

<https://github.com/stefanomondino/RxSwiftExample>



```
.just("Fine!")
```