PCP Reactive Programming

From imperative to functional to reactive

Robin Bürgi & Dorus Janssens

19.12.2017

Inhaltsverzeichnis

Einleitung

ReactiveX ist eine Bibliothek zum Erstellen von asynchronen und Ereignis basierten Programmen unter Verwendung beobachtbarer Sequenzen.

Diese Sequenzen können aus Daten oder Events jeglicher Art bestehen. Der Vorteil vom Observer/Subscribe Pattern besteht darin, dass nur bestimmt Daten/Ereignise aus einer Sequenz entnommen werden können und der ursprüngliche Stream nicht mutiert wird. Die ReactiveX Bibliothek bietet zudem mehr als 60 Operatoren, die das Mutieren, Filtern oder Selektieren von Daten aus den Streams möglich machen. Der Entwickler muss sich ebenfalls keine Gedanken über Low-Level Threading, Synchronisation, Thread-Sicherheit oder blockierende I/O machen. Dies wird von der Bibliothek übernommen.

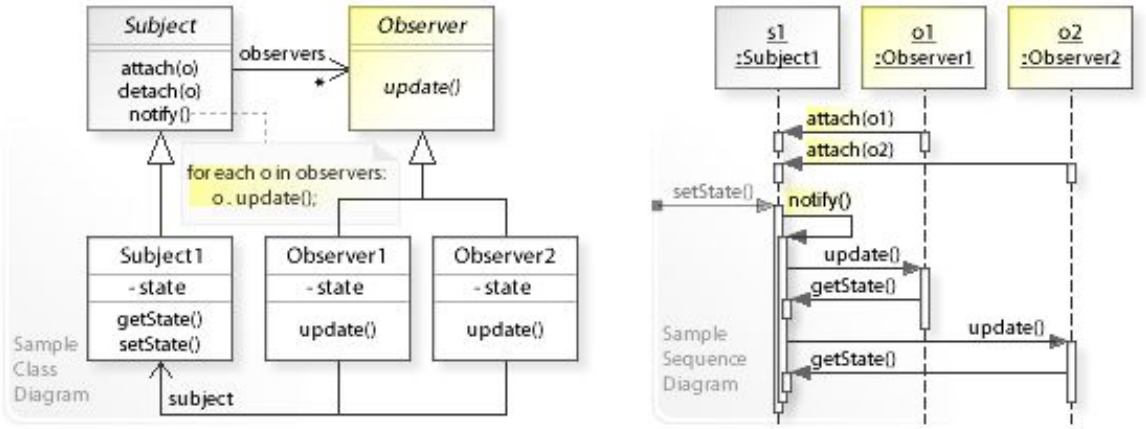
Observable füllen die Lücke, indem sie die ideale Möglichkeit darstellen, auf asynchrone Sequenzen mehrerer Elemente zuzugreifen.

	Einzelne Artikel	Mehrere Artikel
Synchron	T getData()	Iterable<T> getData()
Asynchron	Future<T> getData()	Observable<T> getData()

ReactiveX wird manchmal "funktionale reaktive Programmierung" genannt. Dies ist jedoch eine falsche Bezeichnung. ReaktiveX kann funktional und/oder reaktiv sein, wobei es ein anderes Problem löst als "funktionale reaktive Programmierung". Ein Hauptunterschied besteht darin, dass die funktionale reaktive Programmierung sich auf **kontinuierliche** Werte beschränkt, die sich im Laufe der Zeit ändern. ReactiveX beschränkt sich auf **diskrete** Werte, die im Laufe der Zeit ausgegeben werden.

Observable/Observers Pattern

Das Observer Pattern ist ein Software Design Pattern in welchem ein Objekt, genannt das Subjekt, eine Liste von Abhängigkeiten, Observers genannt, enthält und diese automatisch über jegliche Statusänderungen informiert, normalerweise durch den Aufruf einer ihrer Methoden.[1]



[2]

Cold Observables

Ein Observable wird Cold Observable genannt, wenn der Producer vom Observable erstellt wird. Dies wird führt zu einer eins-zu-eins Beziehung zwischen Observable und Producer, was auch unicast genannt wird. Folgender Code erzeugt ein Cold Observable von einem Websocket:

```
const source = new Observable((observer) => {
 const socket = new WebSocket('ws://someurl');
 socket.addEventListener('message', (e) => observer.next(e));
 return () => socket.close();
});
```

[1] https://en.wikipedia.org/wiki/Observer_pattern
[2] https://commons.wikimedia.org/wiki/File:W3sDesign_Observer_Design_Pattern_UML.jpg

Hot Observables

Im Gegensatz zu einem Cold Observable wird bei einem Hot Observable nur die Referenz eines Producers an das Observable übergeben. Dies führt dazu, dass ein Producer mehrmals verwendet werden kann.

```
const socket = new WebSocket('ws://someurl');
const source = new Observable((observer) => {
 socket.addEventListener('message', (e) => observer.next(e));
});
```

Subjects

Ein Subject[3] ist ein spezieller Typ von Observable, der es ermöglicht, Werte mit mehreren Observern zu verknüpfen. Während einfache Observables Unicast sind (jeder abonnierte Observer besitzt eine unabhängige Ausführung des Observable), sind Subjects Multicast.

Async Subject

Ein AsyncSubject leitet nur den letzten Wert an seine Observer weiter, wenn der Stream beendet wird.

Behavior Subject

BehaviorSubjects sind nützlich für die Darstellung von "Werten im Laufe der Zeit". Zum Beispiel ist ein Ereignisstrom von Geburtstagen ein Subjekt, aber der Strom des Alters einer Person wäre ein BehaviorSubject. Es speichert den letzten Wert, der an seine Konsumenten gesendet wird, und wenn ein neuer Observer abonniert wird, erhält er sofort den "aktuellen Wert" vom BehaviorSubject.

Replay Subject

Ein ReplaySubject speichert eine definierte Anzahl Werte und gibt diese bei neue Abonnenten wieder. Mann kann sich dies wie ein Cache vorstellen.

[3] http://reactivex.io/rxjs/manual/overview.html#subject

Operatoren

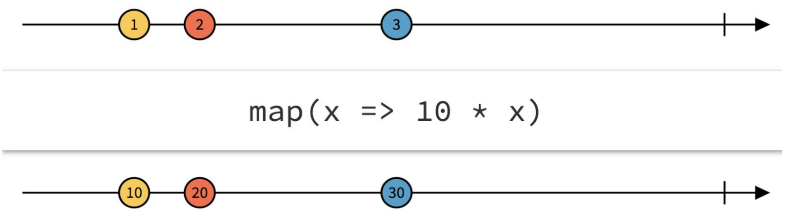
Murmel Diagramme

Asynchrone Daten Sequenzen können hervorragend mit Murmel Diagrammen[4] visualisiert werden. Dies unterstützt den Entwickler, das Verhalten der Operatoren zu verstehen. Viele Operatoren besitzen eine ähnliche Funktionalität und unterscheiden sich nur darin, wie die Sequenz weitergegeben wird oder welche Werte von der Input Sequenz verwendet werden.

Standard Operatoren[5]

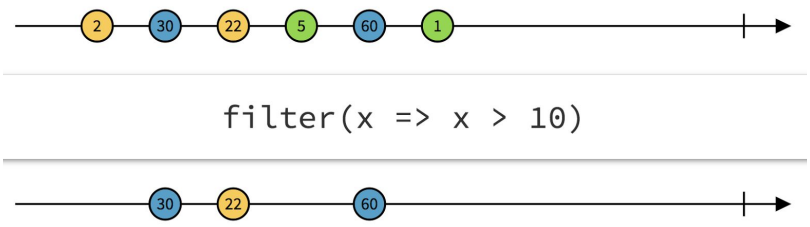
Map

Map kann verwendet werden, um bei jedem Event den Wert zu transformieren.



Filter

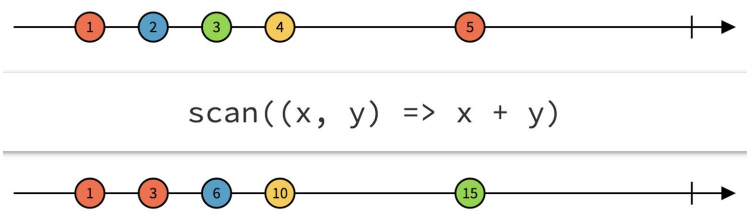
Filter kann verwendet werden, um nur Elemente, die eine gewisse Bedingung erfüllen, durchzulassen.



[4] http://rxmarbles.com
[5] http://reactivex.io/rxjs/manual/overview.html#operators

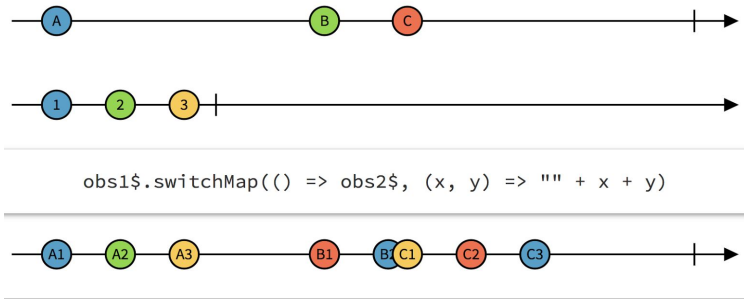
Scan

Scan kann verwendet werden, um Werte in einem Stream zu kumulieren. Der Unterschied zu reduce ist, dass das Zwischenresultat jedesmal weitergegeben wird. Reduce gibt den Wert nur weiter, wenn der Stream abgeschlossen wird.



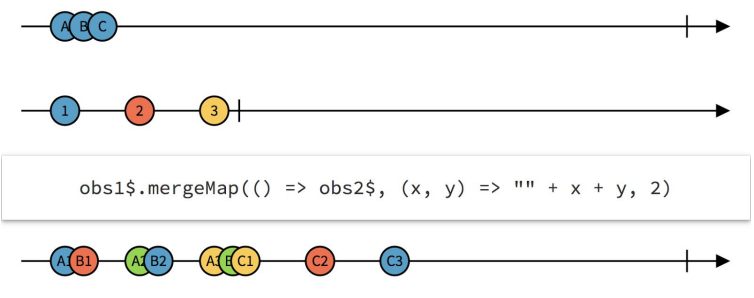
SwitchMap

SwitchMap kann verwendet werden, um einen Stream mit einem zweiten Stream zu kombinieren. Dabei wird der aktuellste Wert von Stream 1 mit den Werten von Stream 2 multipliziert.



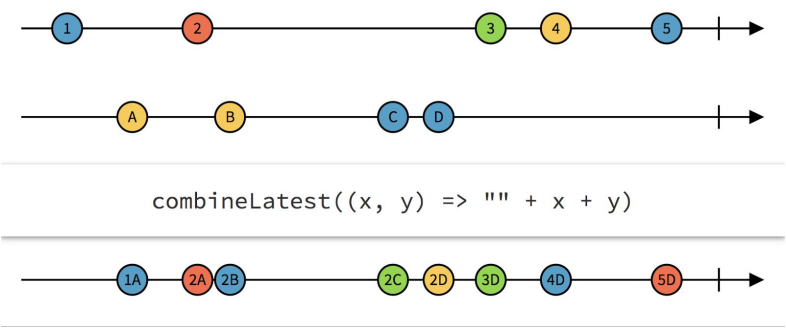
MergeMap

Mergemap kann verwendet werden, um einen Stream mit einem zweiten Stream zu kombinieren. Hierbei werden sämtliche Werte miteinander multipliziert.



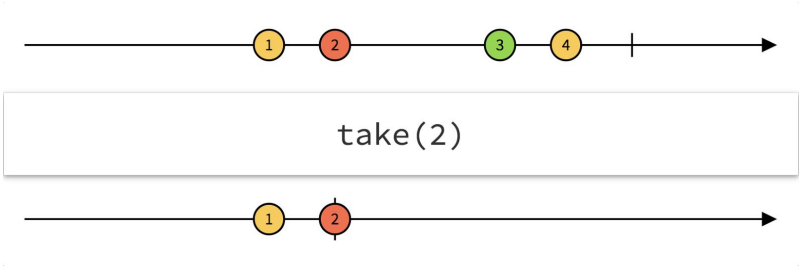
combineLatest

CombineLatest kann verwendet werden, um die letzten Werte von zwei Streams zu kombinieren.



Take

Take kann verwendet werden, um die aktuellen letzten (n) Werte auszulesen. Take schliesst den Stream anschliessend ab.

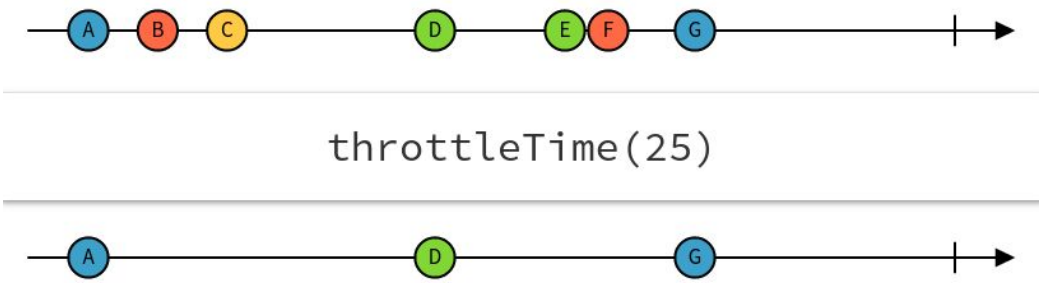


Backpressure Operators

Falls ein Stream zu viele Ereignisse produziert, bzw. der Konsument diese nicht schnell genug abarbeiten kann, gibt es verschiedene Operatoren, die angewendet werden können um diese Problem zu reduzieren. Dabei wird zwischen Lossy und Lossless Operationen unterschieden. Lossy Operatoren verlieren Daten, Lossless Operatoren nicht.

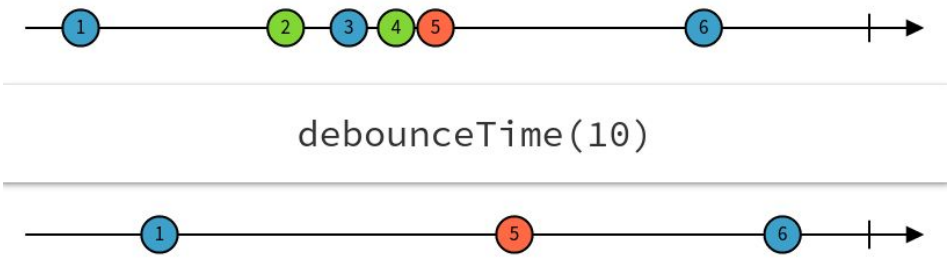
Throttling

Die Funktion Throttle gibt immer nur das erste Element eines Streams, für einen vordefinierten Zeitintervall zurück. Throttle alternativ, anstelle des Zeitintervalls, auch auf ein anderes Observable warten.



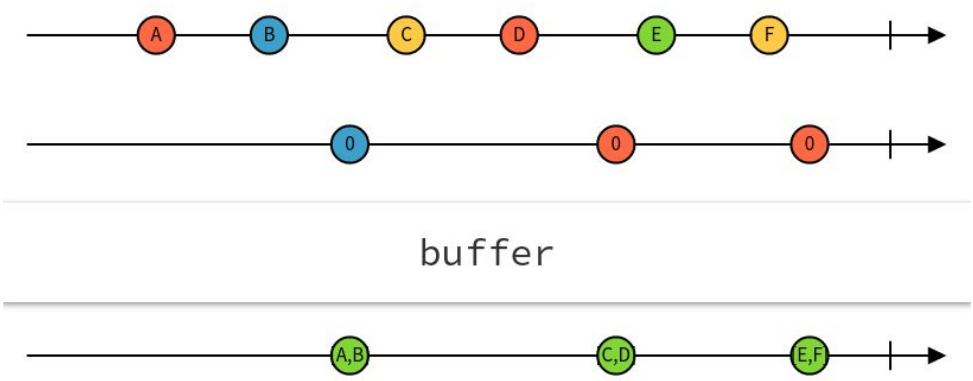
Debounce

Der Debounce-Operator sendet immer nur dann, wenn eine definierte Zeit verstrichen ist, in der kein Event aufgetreten ist. Dies kann zum Beispiel verwendet werden, um nicht auf jede einzelne Eingabe eines Users einzugehen, sondern erst, wenn er fertig geschrieben hat.



Buffering

Dieser Operator ist im Gegensatz zu Debounce und Throttle verlustfrei. Die Buffer-Funktion wartet auf ein anderes Observable oder einen vordefinierten Zeitintervall und sendet danach alle Elemente als Liste zusammengefasst weiter.



Experiment

Ausgangslage

Bitcoin ist die bekannteste[6] Kryptowährung zur Zeit dieses Experiments. Eine der Vorteile, die Kryptowährungen gegenüber herkömmlichen Zahlungsmitteln haben, ist, dass die Transaktionen transparent sind. Das heisst, dass jeder die Möglichkeit besitzt, die Transaktionen auf der Blockchain zu begutachten. Unser Experiment verwendet den Websocket von Blockchain.info, auf der die aktuellen Transaktionen ausgegeben werden.

Anhand von diesem Stream an Transaktionen kann RxJS seine Funktionalität hervorragend beweisen.

Websocket: https://blockchain.info/api/api_websocket

Ziel

Ziel ist es mittels einer Web App die aktuellen Transaktionen aufzuzeigen. Anhand von RxJS Operatoren soll gezeigt werden, wie der Transaktion Stream verändert werden kann, um Aussagen über den Datenfluss zu machen.

Folgende Aussagen wollen wir machen:

- Wie viele Bitcoin seit dem Aufruf der Webseite den Besitzer gewechselt haben.
- Wie viele Transaktionen in dieser Zeit verarbeitet wurden.

Zusätzlich wollen wir die letzten zehn Transaktionen anzeigen, die mehr als 5 Bitcoin fassen.

Vorgehen

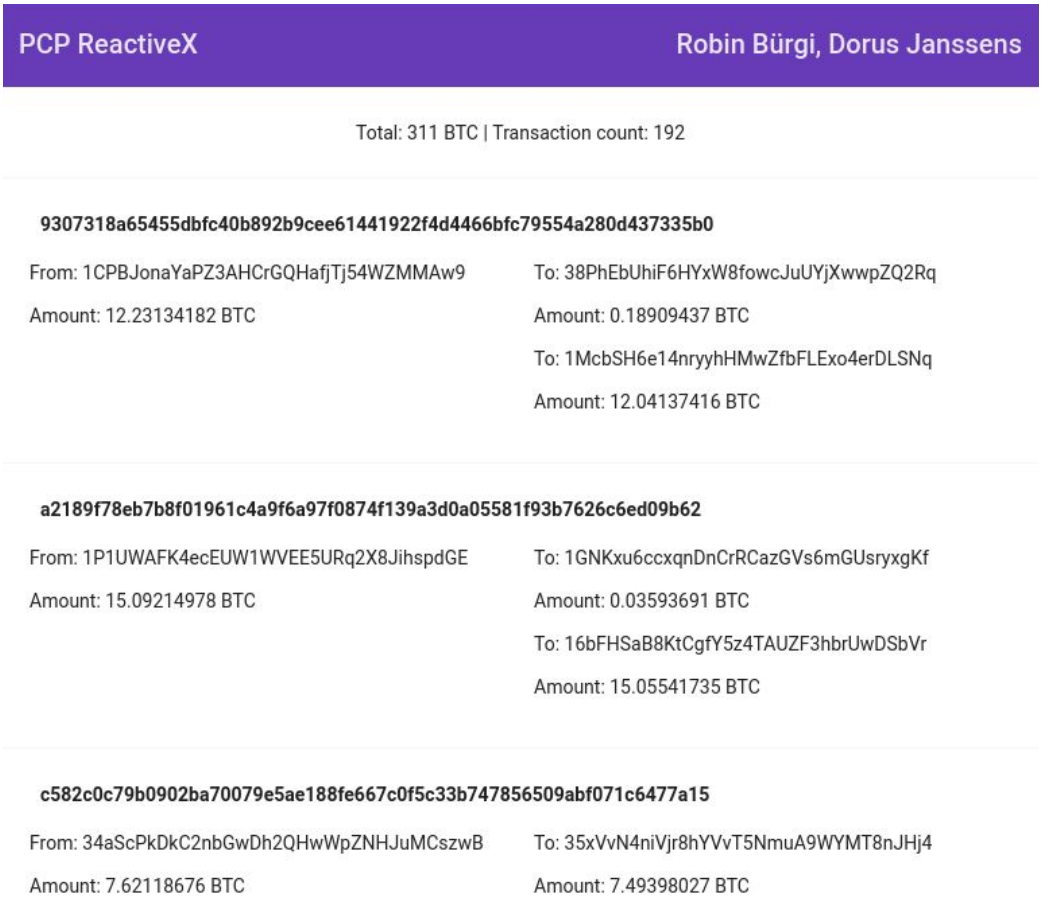
Um eine Web App bereit zu stellen, wurde ein Angular 5 Projekt im Ordner “demo” initialisiert. Die RxJS Bibliothek muss nicht noch explizit installiert werden, da Angular 5 diese bereits in der aktuellen Version(5.5.5) beinhaltet.

In der Hauptkomponente wurde der Websocket instanziert und die Events mittels der Helper Function “fromEvent” zu einem Observable transformiert. Damit die Daten an mehreren Stellen im GUI verfügbar sind, wurde der Stream an ein Subject weitergeleitet. Daraus können mehrere Variabeln sich beim Subject subscriben und die erhaltenen Werte wie benötigt filtern, kummulieren, debouncen etc.

[6] https://blockchain.info

Resultat

Wir haben eine Web App programmiert, die sich auf ein Transaktions Websocket verbindet und die Daten anschliessend mit RxJS in Echtzeit verarbeitet. Die Applikation ist unter https://coin-hard-or-coin-home.firebaseapp.com/ verfügbar.



Mit dieser Applikation konnte die “Power” von RxJS demonstriert werden. Sämtliche Daten sind auf Github öffentlich verfügbar[7] und können als Referenz verwendet werden.

Fazit

ReactiveX ist eine sehr mächtige Bibliothek, wenn Daten in Echtzeit verarbeitet werden sollen. Die Lernkurve von ReactiveX ist steil, aber wenn Konzepte der funktionalen Programmierung bereits vorhanden sind, kann das Potential schnell ausgeschöpft werden.

[7] https://github.com/code-inflation/pcp-reactive

Unterstützte Sprachen[8]

Sprache	ReactiveX Bibliothek
Java	RxJava
JavaScript	RxJS
C#	Rx.NET
C#(Unity)	UniRx
Scala	RxScala
Clojure	RxClojure
C++	RxCpp
Lua	RxLua
Ruby	Rx.rb
Python	RxPY
Go	RxGo
Groovy	RxGroovy
JRuby	RxJRuby
Kotlin	RxKotlin
Swift	RxSwift
PHP	RxPHP
Elixir	reaxive
Dart	RxDart

[8] http://reactivex.io/languages.html