ReactiveX mit RxJava

Roman Roelofsen - W11K GmbH / theCodeCampus

Twitter & GitHub: **romanroe**

Über mich

- Alpha Geek, Entwickler, Trainer
- W11K GmbH - *The Web Engineers*
- Individualsoftware
- theCodeCampus
- Schulungsanbieter Angular & TypeScript

Reaktive Programmierung

*... a programming paradigm oriented around **data flows** and the **propagation of change**."*

Im Kleinen

- Strom von Daten: Liste
- Änderungen verfolgen: Events (Mouse-Clicks, ...)

Im Großen

- Strom von Daten: Web-Sockets
- Änderungen verfolgen: Message Bus

Iterator

- `java.util.Iterator`
- Synchron, Pull
- Keine Fehler-Konzept

Callback

- `java.util.function.Funtion<T, R>`
- Asynchron, Push
- Kein standardisiertes Fehler-Konzept

Reactive Programming = Iterator + Callback

(+ Fehlerbehandlung)

ReactiveX

- RxJava
- **http://reactivex.io**
- Implementierungen für
- Java, JavaScript/TypeScript, .NET, Scala, Clojure, Swift, etc.

Java Flow API

- ab Java 9
- Reactive Programming basierend auf http://www.reactive-streams.org/
- Observable -> Flowable
- *Subject -> *Processor

API

- Observable
  - Liefert Daten

	Single return value	Multiple return values
Pull/Synchronous/Interactive	Object	Iterables(Array | Set | Map)
Push/Asynchronous/Reactive	~~Promise~~ Future	Observable

- Observer
  - Bekommt Daten
  - `subscribe` am Observable -> `Disposable`

Operatoren

- Methoden am Observable
  - `map/filter/...`
- Kombination von Observables
  - `flatMap/withLatestFrom/...`
- Operatoren erzeugen immer neue `Observable`s

Demo - Operatoren

API - cold/synchron

```
Observable<Integer> observable = Observable.create(e -> {
        e.onNext(1);
        e.onNext(2);
        e.onComplete();
});
```

```
observable.subscribe(new Observer<Integer>() {
    public void onSubscribe(Disposable d) {
    }
    public void onNext(Integer i) {
    }
    public void onError(Throwable e) {
    }
    public void onComplete() {
    }
});
```

Fehlerbehandung

Observer

```
Observable<Integer> observable = Observable.create(e -> {
        e.onNext(1);
        e.onNext(2);
        e.onError(new RuntimeException("error"));
        e.onNext(3); // wird nicht "gesendet"
});
```

- Stream termininert bei einem Fehler!

Use Cases

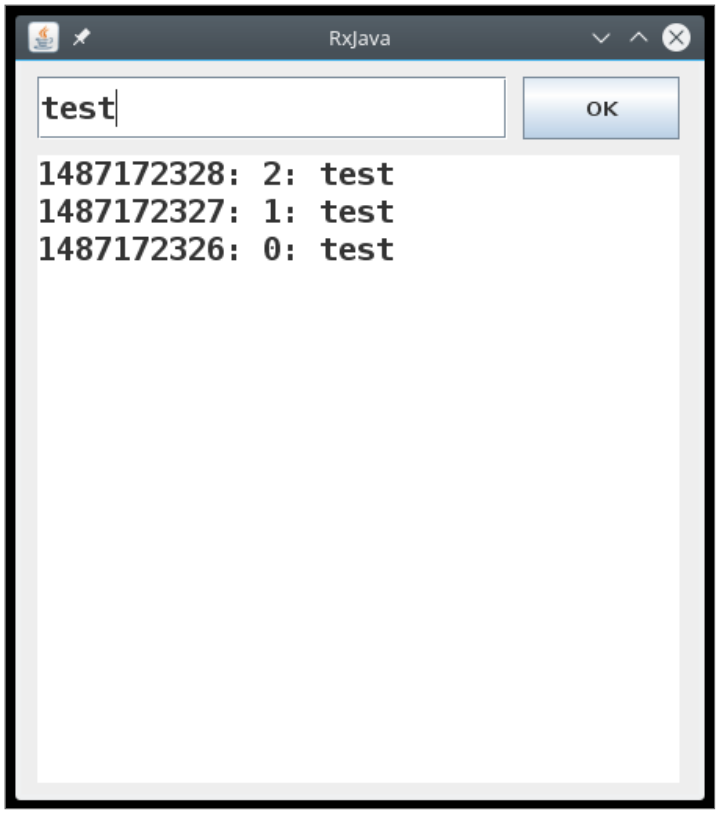
Http Client - RxNetty

```
HttpClient.newClient(serverAddress)
    .enableWireLogging("hello-client", LogLevel.ERROR)
    .createGet("/hello")
    .doOnNext(resp -> logger.info(resp.toString()))
    .flatMap(resp -> resp.getContent()
                         .map(bb ->
                            bb.toString(Charset.defaultCharset())))
```

Http Client - Netflix Ribbon

```
HttpResourceGroup httpResourceGroup = Ribbon.createHttpResourceGroup("movieServiceClient",ClientOptions.create()
                    .withMaxAutoRetriesNextServer(3)
                    .withConfigurationBasedServerList("localhost:8080,localhost:8088"));

HttpRequestTemplate<ByteBuf> recommendationsByUserIdTemplate =
                                        httpResourceGroup.newTemplateBuilder("recommendationsByUserId", ByteBuf.class)
            .withMethod("GET")
            .withUriTemplate("/users/{userId}/recommendations")
            .withFallbackProvider(new RecommendationServiceFallbackHandler())
            .withResponseValidator(new RecommendationServiceResponseValidator())
            .build();

Observable<ByteBuf> result = recommendationsByUserIdTemplate.requestBuilder()
                        .withRequestProperty("userId", “user1")
                        .build()
                        .observe();
```

User Interface



Subject

- Observable und Observer
- Multiplexer
- Puffer

- Nützlich, wenn Datenquelle nicht verschachtelt werden kann

```
Observable.create(e -> {
        ...
        ...
});
```

- z.B. Servlet -> Subject -> Observer

```
public class Servlet extends HttpServlet {

    protected void doGet(HttpServletRequest req,
                         HttpServletResponse res) {

        subject.onNext(Pair.of(req, res));

    }
}
```

PublishSubject

```
Subject<Integer> sub1 = PublishSubject.create();
  sub1.onNext(1);
  sub1.onNext(2);
  sub1.subscribe(System.out::println);
  sub1.onNext(3);
```

Ausgabe

```
3
```

ReplaySubject

```
Subject<Integer> sub1 = ReplaySubject.createWithSize(3);
sub1.onNext(1);
sub1.onNext(2);
sub1.onNext(3);
sub1.onNext(4);
sub1.onNext(5);
sub1.subscribe(System.out::println);
sub1.onNext(6);
```

Ausgabe

```
3
4
5
6
```

ReplaySubject

```
Subject<Integer> sub1 = BehaviorSubject.createDefault(99);
sub1.subscribe(System.out::println);
sub1.onNext(1);
sub1.onNext(2);
sub1.onNext(3);
```

Ausgabe

```
99
1
2
3
```

Java 9 Flow API

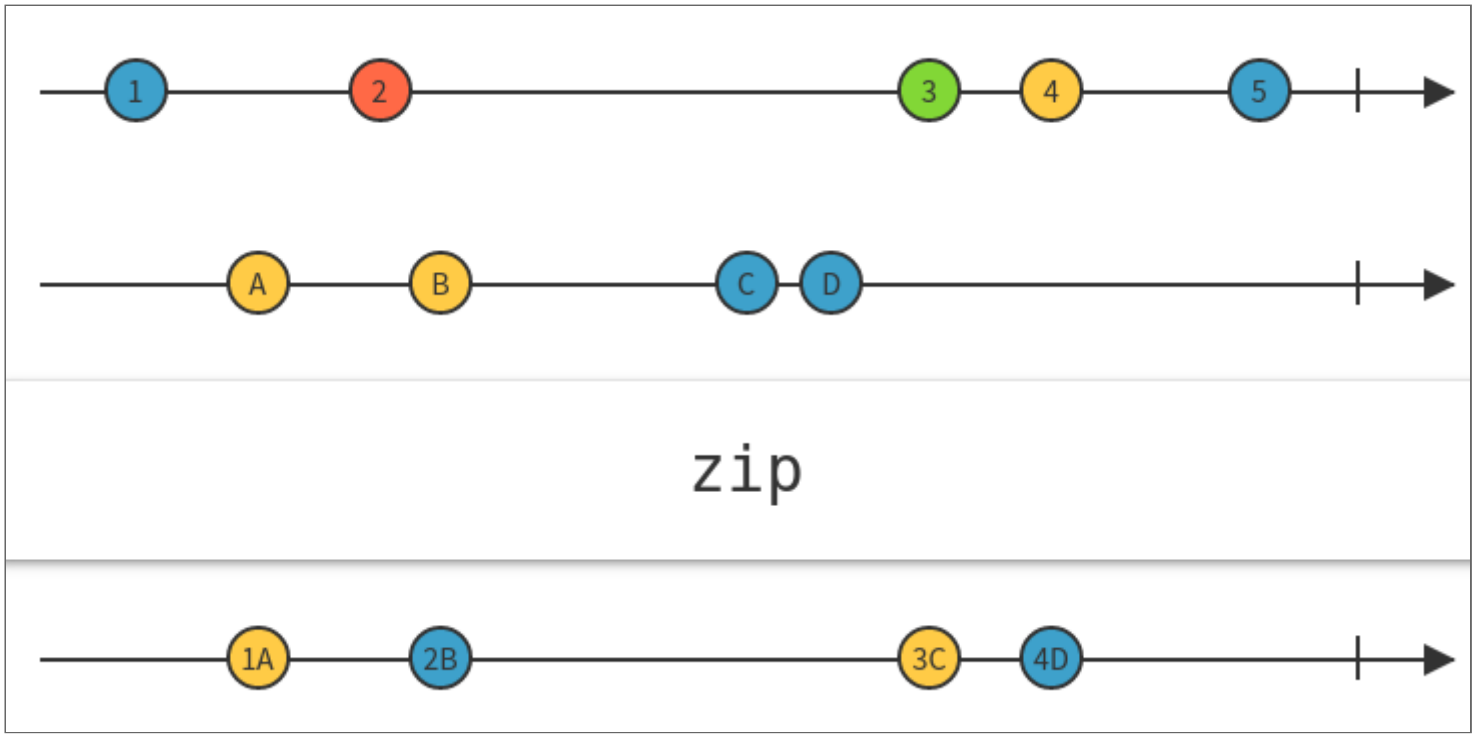
```
Flowable.create(subscriber -> {
    int count = 0;

        while (true) {
            subscriber.onNext(count++);
        }

    }, BackpressureStrategy.DROP)
            .observeOn(Schedulers.newThread(), false, 1)
            .subscribe(val -> {

                        Thread.sleep(1000);
                        System.out.println(val);

                    }
            );
```

```
Flowable.create(subscriber -> {
    int count = 0;

        while (true) {
            subscriber.onNext(count++);
        }

    }, BackpressureStrategy.MISSING)
            .onBackpressureDrop()
            .observeOn(Schedulers.newThread(), false, 1)
            .subscribe(val -> {

                        Thread.sleep(1000);
                        System.out.println(val);

                    }
            );
```

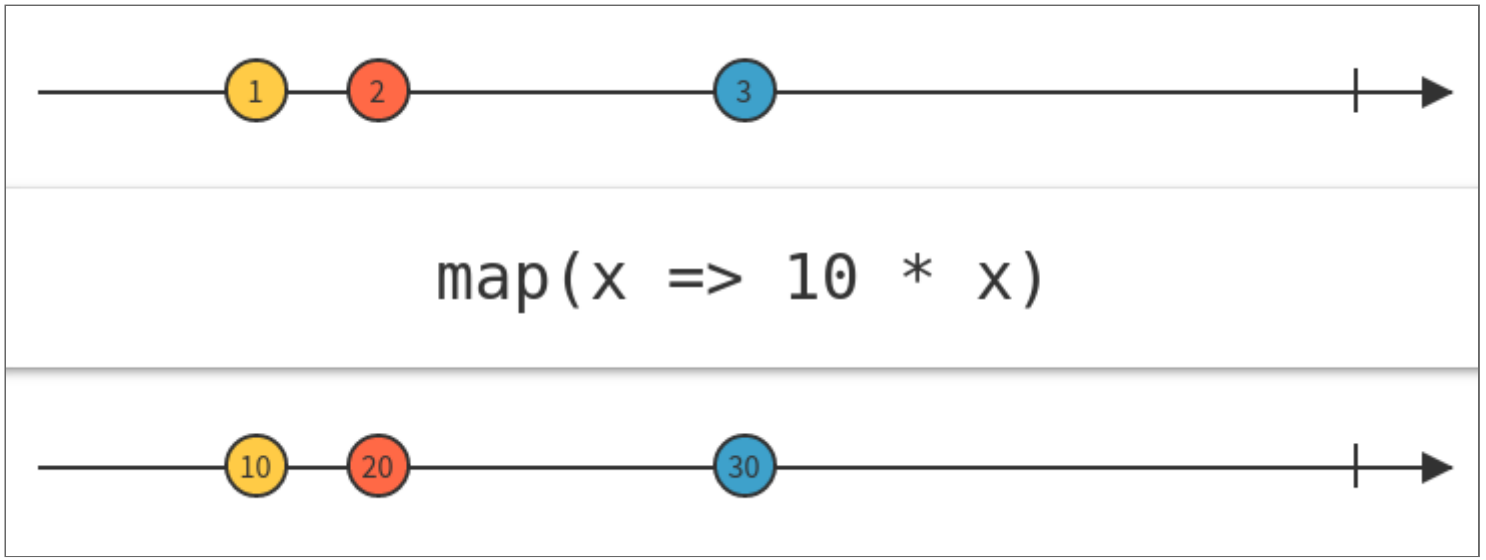
```
Flowable.create(subscriber -> {
    int count = 0;

        while (true) {
            subscriber.onNext(count++);
        }

    }, BackpressureStrategy.MISSING)
            .onBackpressureBuffer(10)
            .observeOn(Schedulers.newThread(), false, 1)
            .subscribe(val -> {

                        Thread.sleep(1000);
                        System.out.println(val);

                    }
            );
```

Marble Diagramme

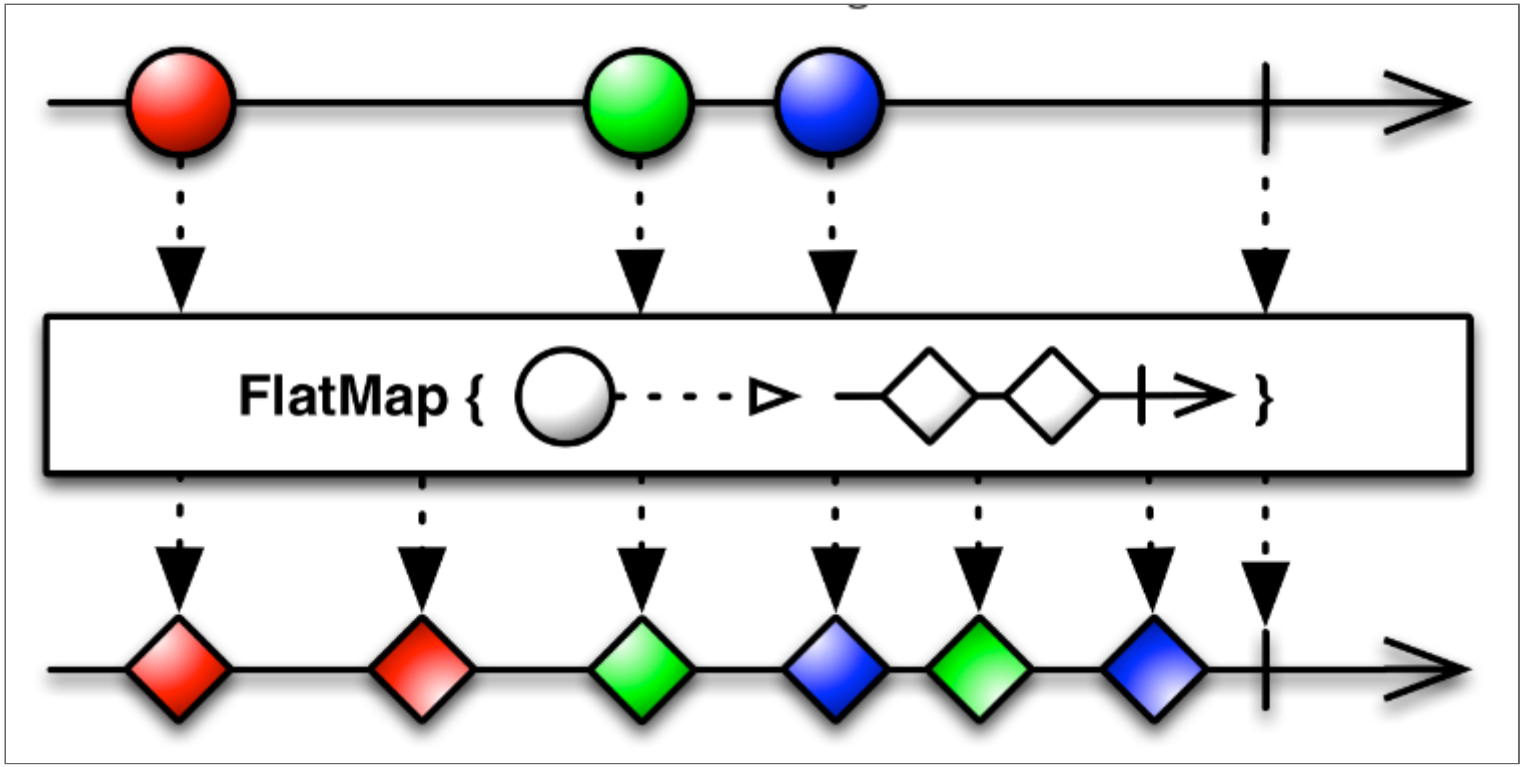
zip



map



flatMap



Schedulers

- Observable Streams sind nicht grundsätzlich asynchron!
- Schedulers verlagern den Observer und die Operatoren in Threads

```
obs$.observeOn(Schedulers.io()).map(i -> ioBound(i));
obs$.observeOn(Schedulers.computation()).map(i -> heavyOnCpu(i));
```

Integrationsmöglichkeiten

- Mit `Observable`/`Observer` lassen sich alle Kommunikationsszenarien abbilden
- Lokal
  - JDBC
  - Thread Kommunikation
- Remote
  - REST
  - WebServices

Servlets

JDBC

Diverses

- Ratpack - https://ratpack.io/
- Akka - http://akka.io/
- RxJS

Fazit

- Lernkurve:
  - kurz flach
  - dann lange steil
    - dann wieder flach
- ReactiveX macht komplexe Datenflüsse "einfach"
  - Keine komplexe Datenflüsse? Dann ggf. overkill
- Kein echtes Projekt zum Lernen nehmen
- **http://rxmarbles.com**

Roman Roelofsen - w11k GmbH

@romanroe