Reactive programming met RxJava

I will get back to you, I promise...

Reactive programming heeft het afgelopen jaar flink aan populariteit gewonnen. En terecht, want het concept lost een hoop traditionele problemen voor webapplicaties op. Toch is het niet alles goud wat er blinkt. Het asynchrone programmeermodel heeft ook nadelen. Het lijkt vaak ingewikkelder dan synchrone code, foutafhandeling is lastiger en "callback hell" is een veel gehoorde klacht. Is daar dan niets aan te doen? Jawel! RxJava biedt hulp.

Inleiding

RxJava is een library voor het bouwen van asynchrone en event-based programmatuur. Het is een Java-implementatie van ReactiveX, een API voor asynchroon programmeren. ReactiveX leunt op de observable en iterable patterns van de Gang of Four en principes van functioneel programmeren. RxJava biedt een abstractielaag waardoor de aanroepende code zich niet druk hoeft te maken om zaken als threading, synchronisatie en non-blocking I/O.

ReactiveX (Rx) is ontwikkeld door het Cloud Programmability team bij Microsoft met als doel om programmeren voor grootschalige asynchrone en data-intensieve internetarchitecturen te vereenvoudigen. De eerste implementatie was dan ook voor .NET in november 2009. Nadat Microsoft Rx.NET open source heeft gemaakt, is ook actief gewerkt aan implementaties in andere talen. Zo hebben ontwikkelaars van Netflix de Java-implementatie gemaakt: RxJava.

RxJava wordt gebruikt door bedrijven als Netflix, Soundcloud en The New York Times. Er zijn daarnaast ook andere producten en libraries die RxJava hebben omarmd, waaronder Couchbase, Apache Camel en Vert.x.

Basisprincipes

RxJava werkt met Observables en Subscribers. Een object dat de Observer interface implementeert, kan zichzelf subscriben op een Observable. De subscriber reageert vervolgens wanneer de Observable een object of een reeks objecten uitzendt (emit). Hierin zien we elementen uit het Observer pattern terug: de Observable houdt de subscribers bij en notificeert deze automatisch bij het beschikbaar komen van nieuwe informatie. Dit faciliteert concurrency, omdat het niet nodig is om de huidige thread te laten wachten totdat de Observable data uitzendt. Een Observable lijkt op een stream, maar biedt asynchrone afsluiting en foutafhandeling. In de interface van Observable zien we elementen uit het Iterable pattern terug (zie **tabel 1**).

Belangrijk verschil is dat Observables een API bieden voor het aan elkaar koppelen van operaties (Observables zijn *composable*). Deze eigenschappen maken Observables een ideale manier om reeksen van meerdere objecten

	Iterable	Observable
model	*pull*	*push*
lezen van data	T next()	onNext(T)
foutafhandeling	throws Exception	onError(Exception)
klaar?	!hasNext()	onCompleted()

Tabel 1

Bert Jan Schrijver is Software craftsman bij JPoint. Hij is erg geïnteresseerd in Java, open source en continuous delivery.

Tim van Eijndhoven is Software craftsman bij JPoint. Hij houdt zich vooral bezig met Java, applicatie-architectuur, software security en open source.

asynchroon te benaderen. **Listing 1** illustreert dit. Hierin zie je 3 manieren om een Observable aan te maken:

1. De eenvoudigste manier is Observable.*from*(), die direct een Observable maakt op basis van een eenvoudig object.
2. Je kunt een timed Observable aanmaken, die een event na een bepaalde tijdseenheid afvuurt.
3. Een wat flexibelere manier is Observable.*create*(), die in het voorbeeld events emit op het moment dat een subscriber zich aanmeldt.

Dit voorbeeld geeft de volgende output:

```
Observable 1: Hello
Observable 1: world
Observable 1: complete!
Observable 3: Hello
Observable 3: world
Observable 3: bye!
```

En na 1 seconde:

```
Observable 2
```

Dit ogenschijnlijk eenvoudige principe is enorm krachtig en heel breed toepasbaar. Verderop volgt een wat uitgebreider voorbeeld van de toepassingsmogelijkheden. Excepties die in de observable optreden worden aan een subscriber doorgegeven met een aanroep naar de *onError*() functie van de subscriber, zodat deze afhandeling kan doen op basis van de fout.
Er zijn twee soorten observables. De ene is passief en genereert notificaties op aanvraag, bijvoorbeeld wanneer een subscriber zich aanmeldt. Dit zijn *cold observables*: elke subscriber krijgt de volledige reeks objecten te verwerken. Dit is in het voorbeeld te zien bij

```
String[] input = { "Hello", "world" };
Observable<String> observable1 = Observable.from(input);
observable1.subscribe(
       message -> System.out.println("Observable 1: "+ message),
       error -> System.err.println("Observable 1: " + error.getMessage()),
       () -> System.out.println("Observable 1: complete!")
);

Observable<Long> observable2 = Observable.timer(1, TimeUnit.SECONDS);
observable2.subscribe(message -> {
   System.out.println("Observable 2");
});

Observable<String> observable3 = Observable.create(subscriber -> {
   subscriber.onNext("Hello");
   subscriber.onNext("world");
   subscriber.onError(new Exception("bye!"));
   subscriber.onCompleted();
});

observable3.subscribe(
       message -> System.out.println("Observable 3: " + message),
       error -> System.err.println("Observable 3: " + error.getMessage()),
       () -> System.out.println("Observable 3: complete!")
);
```

Listing 1: basiswerking van Observables

observable3 waarbij iedere subscriber "Hello" en vervolgens "World" binnenkrijgt. De andere soort is actief en genereert notificaties ongeacht of er subscribers zijn. Dit zijn *hot observables*, die gegevens uitzenden in hun eigen tempo en volgens hun eigen schema. Een subscriber ontvangt alleen notificaties, die verzonden zijn na zijn aanmelding.

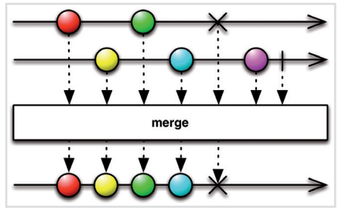
Samenstellen van Observables

RxJava biedt de mogelijkheid om observables te combineren. Hiermee kun je op een aantal manieren de emits van 2 of meer observables combineren tot 1 nieuwe observable.Zo voegt merge() de output van meerdere Observables samen tot één Observable.
mergeDelayError() doet hetzelfde, maar spaart eventuele opgetreden fouten op en emit die als laatste.
Operatie zip() combineert Observables via een vooraf gedefinieerde functie en emit de resultaten van deze functie. Een voorbeeld van de toepassing hiervan vind je in de voorbeelden verderop. Operatie combineLatest() combineert de twee laatst ge-emitte items van twee Observables via een vooraf gedefinieerde functie en emit de resultaten hiervan.

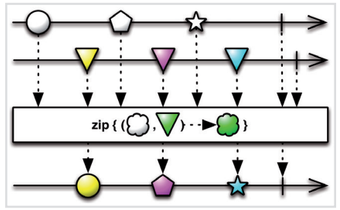
Een compleet overzicht van manieren om Observables samen te stellen en uitgebreide codevoorbeelden vind je op [2]. Dit is tevens de bron van de gebruikte afbeeldingen.

Threads en (a)synchroniciteit

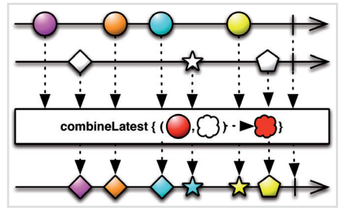
Wanneer je met RxJava een API aanbiedt, geldt een belangrijk uitgangspunt: alle interacties met de API zijn asynchroon en declaratief. De implementatie van de API kiest zelf of iets blocking of non-blocking is en wat het concurrency-gedrag is. Dit is te sturen met een *Scheduler*: een thread of



Merge



Zip



CombineLatest

threadpool voor het afhandelen van een bepaalde bewerking. Zo is er een 'computation' scheduler voor event-loops en callback-verwerking, een io-scheduler, een 'immediate' scheduler die direct begint in de huidige thread en een 'trampoline' scheduler die de te verwerken actie achteraan de queue voor de huidige thread plaatst. Operaties op Observables hebben een standaard scheduler, maar in veel gevallen kun je ook een specifieke scheduler kiezen om de manier van verwerking te beïnvloeden.

Backpressure

Het reactief verwerken van data is eenvoudig wanneer de subscriber de aangeboden gegevens sneller kan verwerken dan dat de observable ze aanbiedt. Het wordt een uitdaging wanneer dit niet zo is. In dat geval vindt er een steeds grotere opbouw van nog onverwerkte data plaats. Om ervoor te zorgen dat er geen gegevens verloren gaan, zou het systeem de onverwerkte data moeten bijhouden in een eindeloos uitbreidende buffer, waardoor het potentieel enorme resources zou gebruiken. Om dit tegen te gaan, moet de subscriber tegendruk op de observable kunnen uitoefenen om minder events uit te zenden. Dat wordt backpressure genoemd. RxJava biedt verschillende manieren om met backpressure om te gaan.

Lossy

De eerste manier is door aan de observable aan te geven dat we niet elk element in de stream kunnen verwerken, maar dat we slechts periodiek iets willen ontvangen, oftewel throttling. Er zijn verschillende mogelijkheden om te bepalen welke elementen dan worden uitgezonden:

- *Sampling* kijkt periodiek naar de stream en emit het laatste element.
- *ThrottleFirst* kijkt ook periodiek en emit het eerste element uit de stream.
- *Debounce* emit alleen elementen, die binnen een tijdsinterval niet gevolgd worden door een ander element.

Lossless

Een andere manier om met backpressure om te gaan, is om gegevens te aggregeren en vervolgens te emitten als collectie. Dit reduceert de totale hoeveelheid gegevens niet, maar geeft de mogelijkheid aan de gebruiker om zelf te beslissen wat hij doet met de gegevens (in plaats van dit over te laten aan de observable, zoals bij throttling). Dit kan door gebruik te maken van:

- een *buffer*, die werkt met een interval op basis van een opgegeven aantal elementen of op basis van een eigen functie, die aangeeft of de buffer vol is.
- een *window*: vergelijkbaar met een buffer, maar geeft een nieuwe Observable door in plaats van een collectie.

Voor een complete uitleg van backpressure en de mogelijkheden, zie [3].

RxJava in de praktijk

Tijdens de ontwikkeling van RxJava door Netflix is de populariteit van Netflix flink gestegen. Al het dataverkeer dat door de Netflix API gaat, wordt door RxJava afgehandeld. Hetzelfde geldt voor de Hystrix-library voor fouttolerantie, ook door Netflix ontwikkeld. Op deze pijlers heeft Netflix een compleet reactive stack gebouwd, waarmee alle interne services met elkaar verbonden zijn. Deze stack handelt zo'n 30% van al (!) het internetverkeer in de VS af.

Als praktijkvoorbeeld voor het gebruik van RxJava nemen we een eenvoudige webserver gemaakt met Vert.x, een reactief applicatieplatform voor de JVM. Vert.x biedt een microservices-architectuur, waarbij services met elkaar verbonden zijn via een eventbus. Typisch handel je met Vert.x een request af door één of meerdere services via de eventbus aan te roepen, daar wat logica op los te laten en vervolgens een response terug te sturen. **Listing 2** geeft een eenvoudige implementatie van dit concept.

Het voorbeeld illustreert dat je al vrij snel in een situatie belandt, waarin je steeds dieper in inner classes terecht komt. Java 8 lambda's verlichten de pijn enigszins, maar fraai is het niet.

In **Listing 3** herschrijven we de service calls en het samenstellen van de HTTP-response naar RxJava Observables.

```
// Create a handler for HTTP requests.
Handler<HttpServerRequest> requestHandler = request -> {

   // Create dummy service requests (parameters for the service calls).
   JsonObject service1Request = new JsonObject().put("key", "value");
   JsonObject service2Request = new JsonObject().put("key", "value");

   // Call service 1.
   vertx.eventBus().send(Service1.ADDRESS, service1Request, service1Response -> {

       // Call service 2.
       vertx.eventBus().send(Service2.ADDRESS, service2Request, service2Response -> {

           // Combine results from both service responses.
           String combinedResult = service1Response.result().body() + " - "
                   + service2Response.result().body();

           // Respond with combined result from service 1 and 2.
           request.response().end("combined result: " + combinedResult);
       });
   });
};

// Create a basic webserver.
vertx.createHttpServer(new HttpServerOptions().setPort(8080))
.requestHandler(requestHandler).listen();
```

Listing 2: Webserver die 2 asynchrone services aanroept

```
// Call service 1.
Observable<Message<String>> service1Response =
    vertx.eventBus().sendObservable(Service1.ADDRESS, service1Request);

// Call service 2.
Observable<Message<String>> service2Response =
    vertx.eventBus().sendObservable(Service2.ADDRESS, service2Request);

// Combine results from both service responses:
// Zip both observables into a single one (which concats the results from the
// individual observables).
Observable<String> combined = Observable.zip(service1Response, service2Response,
    (Message resp1, Message resp2) -> resp1.body() + " - " + resp2.body());

// Respond with combined result from service 1 and 2.
combined.subscribe(response -> {
   request.response().end("combined result: " + response);
});
```

Listing 3: Dezelfde service-calls, maar nu met hulp van RxJava

```
// Send two service calls and combine the result.
Observable.zip(
   vertx.eventBus().sendObservable(Service1.ADDRESS, service1Request),
   vertx.eventBus().sendObservable(Service2.ADDRESS, service2Request),
   (Message res1, Message res2) -> res1.body() + " - " + res2.body())
       .subscribe(resp -> request.response().end("combined result: " + resp));
```

Listing 4: Compacte notatie met RxJava.

Handigheid hierbij is dat Vert.x vanaf versie 3.0 integratie met RxJava biedt en je een eventbus-response als Observable kunt modelleren.

Dit is al een stuk leesbaarder, zeker wanneer je meer dan 2 services na elkaar moet aanroepen. Maar echt veel korter is het nog niet. Daarom volgt **Listing 4** met dezelfde service-aanroepen en response met RxJava, maar dan compacter geschreven.

Als je dit vergelijkt met de callback-brij van het eerste voorbeeld worden een aantal van de voordelen van RxJava duidelijk.

Vergelijking met andere technologieën

Er zijn een aantal andere Java-technologieën waar je aan kunt denken wanneer het gaan om zaken als asynchrone verwerking, een reactief model en het verwerken van streams van data. Zo kun je voor asynchrone verwerking gebruik maken van Java Futures of de CompletableFutures in Java 8 en voor het verwerken van streams is er in Java 8 natuurlijk de stream API. Toch zijn deze alternatieven geen vervangers voor Rx. De streaming API is in de basis bedoeld voor data die zonder latency gegenereerd wordt (op basis van bestaande collecties of functies), terwijl Rx juist gemaakt is voor het verwerken van een continue stroom aan data. Java Futures kunnen eenvoudig gebruikt worden bij een enkel niveau van asynchrone verwerking, maar zodra je ze gaat nesten wordt het al snel minder triviaal. Met CompletableFutures gaat het verwerken en combineren van meerdere niveaus van asynchrone verwerking al beter, maar het blijft een constructie die bedoeld is voor het eenmalig uitvoeren van een functie. Met Rx kun je juist ook reeksen van asynchrone data verwerken.

Een andere speler op het reactive vlak is Akka, met Composable Futures en Actor message passing. De Rx-aanpak is ook hier vaak krachtiger, omdat die niet alleen op enkelvoudige waarden werkt, maar ook op reeksen van waarden en zelfs op oneindige streams van data. Bijkomend voordeel is dat je met Rx de bron van de asynchroniciteit scheidt van datgene dat je gebruikt om die reeksen samen te stellen. Zowel Akka als RxJava bieden implementaties van de reactive streams specificatie, die bedoeld is om het verwerken van grote hoeveelheden data tussen verschillende asynchrone systemen goed te reguleren [6].

Conclusie

RxJava is een library voor de JVM waarmee je asynchroon en event-driven kunt programmeren door het gebruik van Observables, die zowel scalaire data als (oneindige) reeksen van data kunnen verwerken. RxJava biedt een krachtige set operaties waarmee je data kunt aggregeren, foutafhandeling kunt regelen en de verwerking van services en streams op een declaratieve, functionele manier kunt aanpakken. Het feit dat Netflix volledig op RxJava draait, geeft aan dat het een library is die in een serieuze productieomgeving is in te zetten. Wanneer je een reactive applicatie bouwt die grote hoeveelheden gebruikers en data aan moet kunnen, dan is RxJava één van de beste opties van het moment. ■

REFERENTIES

[1] https://github.com/ReactiveX/RxJava
[2] https://github.com/ReactiveX/RxJava/wiki/Combining-Observables
[3] https://github.com/ReactiveX/RxJava/wiki/Backpressure
[4] http://techblog.netflix.com/2013/02/rxjava-netflix-api.html
[5] http://www.reactivemanifesto.org
[6] http://www.reactive-streams.org/