



Grokking RxJava, Part 2: Operator, Operator

22 SEPTEMBER 2014 on rxjava

In [part 1](#) I went over the basic structure of RxJava, as well as introducing you to the `map()` operator. However, I can understand if you're still not compelled to use RxJava - you don't have much to work with yet. But that will change quickly - a big part of the power of RxJava is in all of the operators included in the framework.

Let's go through an example to introduce you to more operators.

The Setup

Suppose I have this method available:

```
// Returns a List of website URLs based on a text search
Observable<List<String>> query(String text);
```

I want to make a robust system for searching text and displaying the results. Given what we know from the last article, this is what one might come up with:

```
query("Hello, world!")
    .subscribe(urls -> {
```

```
        for (String url : urls) {  
            System.out.println(url);  
        }  
    });
```

This answer is highly unsatisfactory because I lose the ability to transform the data stream. If I wanted to modify *each* URL, I'd have to do it all in the `Subscriber`. We're tossing all our cool `map()` tricks out the window!

I could create a `map()` from `urls -> urls`, but then *every* `map()` call would have a for-each loop inside of it - ouch.

A Glimmer of Hope

There is a method, `Observable.from()`, that takes a collection of items and emits each them one at a time:

```
Observable.from("url1", "url2", "url3")  
    .subscribe(url -> System.out.println(url));
```

That looks like it could help, let's see what happens:

```
query("Hello, world!")  
    .subscribe(urls -> {  
        Observable.from(urls)  
            .subscribe(url -> System.out.println(url));  
    });
```

I've gotten rid of the for-each loop, but the resulting code is a *mess*. I've got multiple, nested subscriptions now! Besides being ugly and hard to modify, it also breaks some critical as-yet undiscovered features of RxJava¹. Ugh.

A Better Way

Hold your breath as you view your savior: `flatMap()`.

`Observable.flatMap()` takes the emissions of one `Observable` and returns the emissions of another `Observable` to take its place. It's the ol' switcheroo: you thought you were getting one stream of items but instead you get another. Here's how it solves this problem:

```
query("Hello, world!")
    .flatMap(new Func1<List<String>, Observable<String>>
() {
    @Override
    public Observable<String> call(List<String> urls)
    {
        return Observable.from(urls);
    }
})
    .subscribe(url -> System.out.println(url));
```

I'm showing the full function just so you can see exactly what happened, but simplified with lambdas it looks awesome:

```
query("Hello, world!")
    .flatMap(urls -> Observable.from(urls))
```

```
.subscribe(url -> System.out.println(url));
```

`flatMap()` is weird, right? Why is it returning *another* `Observable`? The key concept here is that the new `Observable` returned is what the `Subscriber` sees. It doesn't receive a `List<String>` - it gets a series of individual `Strings` as returned by `Observable.from()`.

For the record, this part was the hardest for me to understand, but once I had the "aha" moment a lot of RxJava clicked.

It Gets Even Better

I can't emphasize this idea enough: `flatMap()` can return *any* `Observable` it wants.

Suppose I've got a second method available:

```
// Returns the title of a website, or null if 404  
Observable<String> getTitle(String URL);
```

Instead of printing the URLs, now I want to print the title of each website received. But there's a few issues: my method only works on a single URL at a time, and it doesn't return a `String`, it returns an `Observable` that emits the `String`.

With `flatMap()`, solving this problem is easy; after splitting the list of URLs into individual items, I can use `getTitle()` in `flatMap()` for each url before it reaches the `Subscriber`:

```
query("Hello, world!")
    .flatMap(urls -> Observable.from(urls))
    .flatMap(new Func1<String, Observable<String>>() {
        @Override
        public Observable<String> call(String url) {
            return getTitle(url);
        }
    })
    .subscribe(title -> System.out.println(title));
```

And once more, simplified via lambdas:

```
query("Hello, world!")
    .flatMap(urls -> Observable.from(urls))
    .flatMap(url -> getTitle(url))
    .subscribe(title -> System.out.println(title));
```

Crazy, right? I'm composing multiple independent methods returning `Observables` together! How cool is that!

Not only that, but notice how I'm combining two API calls into a single chain. We could do it for any number of API calls. You know how much of a pain in the ass it is to keep all your API calls synced, having to link their callbacks together before presenting the data? We've skipped the trip to callback hell; all that same logic is now encased in this short reactive call².

Operators Galore

We've only looked at two operators so far, but there are so many more! How else can we improve our code?

`getTitle()` returns null if the URL 404s. We don't want to output "null"; it turns out we can filter them out!

```
query("Hello, world!")
    .flatMap(urls -> Observable.from(urls))
    .flatMap(url -> getTitle(url))
    .filter(title -> title != null)
    .subscribe(title -> System.out.println(title));
```

`filter()` emits the same item it received, but only if it passes the boolean check.

And now we want to only show 5 results at most:

```
query("Hello, world!")
    .flatMap(urls -> Observable.from(urls))
    .flatMap(url -> getTitle(url))
    .filter(title -> title != null)
    .take(5)
    .subscribe(title -> System.out.println(title));
```

`take()` emits, at most, the number of items specified. (If there are fewer than 5 titles it'll just stop early.)

Now we want to save each title to disk along the way:

```
query("Hello, world!")  
    .flatMap(urls -> Observable.from(urls))  
    .flatMap(url -> getTitle(url))  
    .filter(title -> title != null)  
    .take(5)  
    .doOnNext(title -> saveTitle(title))  
    .subscribe(title -> System.out.println(title));
```

`doOnNext()` allows us to add extra behavior each time an item is emitted, in this case saving the title.

Look at how easy it is to manipulate the stream of data. You can keep adding more and more ingredients to your recipe and not mess anything up.

RxJava comes with a **ton** of operators. It is intimidating how many operators there are, but it's worth reviewing so you know what's available. It will take a while to internalize the operators but you'll have true power at your fingertips once you do.

On top of all that's provided, you can even write your own custom operators! That's outside the scope of this article, but basically, if you can think it, you can do it³.

So What?

Alright, so you're a hard sell. You're a skeptic. Why should you care about all these operators?

Key idea #3: Operators let you do anything to the stream of data.

The only limit is yourself.

You can setup complex logic using nothing but chains of simple operators. It breaks down your code into composable bits and pieces. That's functional reactive programming. The more you use it, the more it changes the way you think about programs.

Plus, think about how simple our data was to consume once transformed. By the end of our example we were doing two API calls, manipulating the data, then saving it to disk. But our `Subscriber` doesn't know that; it just thinks it's consuming a simple `Observable<String>`. Encapsulation makes coding easier!

In part 3 we'll cover some of the other cool features of RxJava that aren't as directly involved with manipulating data, like error handling and concurrency.

Continue onwards to part 3

¹ The way RxJava does error handling, threading, and subscription cancellation wouldn't work at all with this code. I'll get to that in part 3.

² You may be wondering about the other part of callback hell that is error handling. I'll be addressing that in part 3.

³ If you want to implement your own operators, [check out this wiki page](#), though some implementation details won't make sense until part 3.

Dan Lew




Read [more posts](#) by this author.

Share this post



📍 *Minneapolis* 🔗 <http://blog.danlew.net/about/>

68 Comments Dan Lew Codes

 Login ▾ Recommend 19 Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS **Dominic Fui Dodzi-Nusenu** • 3 years ago

OMG! This is by far the best, most interactive tutorial I have read about RxJava since I started my research, and boi am I happy. Now I completely understand it and plan to implement it soonest. :-)

15 ^ | ▾ • Reply • Share ▸

**Justin Hong** • 3 years ago

Could you have used
.map(url -> getTitle(url))
instead of
.flatMap(url -> getTitle(url))
in your example?

If not, why not? If so, what's the difference?

6 ^ | ▾ • Reply • Share ▸

**Dan Lew** Mod ➔ Justin Hong • 3 years ago

They're not interchangeable, and the reason why took me a while to understand, so I get the confusion.

Think of it this way: what you want is to convert String -> String (the url into a title). map() is a simple type -> type conversion. getTitle() returns Observable<string>, so if you were to use map() you're converting String -> Observable<string>, which is not what you want.

flatMap(), on the other hand, is more of the form String -> Observable<string> -> String. In other words, it takes a String, which is converted into some Observable<string>, which is then used to emit the String from inside the Observable.

5 ^ | ▾ • Reply • Share ▸

**Justin Hong** ➔ Dan Lew • 3 years ago

Ah, I missed the return type of getTitle. Makes sense, thanks!

3 ^ | ▾ • Reply • Share ▸



Ciprian Grigor → Dan Lew • 3 years ago

Can the new observables be executed on another thread so we create several request in parallel that get the titles? And if so I guess the order will not be preserved; title will be in other order than urls?

^ | v • Reply • Share ›



Ivan → Justin Hong • 2 years ago

What is the difference between:

1) getTitle having signature "Observable<string> getTitle(String url)" and using flatMap

2) getTitle having signature "String getTitle(String url)" and using map?

3 ^ | v • Reply • Share ›



Dan Lew Mod → Ivan • 2 years ago

Technically, (1) could return multiple Strings per URL whereas (2) could only return one. Generally speaking, though, if (1) is only expected to return a single result, there isn't much of a difference.

Whether you use map() or flatMap() depends on what you're working with.

3 ^ | v • Reply • Share ›



Mike Mannon • 3 years ago

Could part 3 also mention how you work around configuration changes (orientation, etc), please?

I saw that Novoda are proposing a "solution" (<https://github.com/Reactive...>, although the reviewers aren't in favor.

2 ^ | v • Reply • Share ›



Mike Mannon → Mike Mannon • 3 years ago

it seems the formatter broke the link:

<https://github.com/Reactive...>

the reason I mention this is that further down the review mttkay mentions: "We use retained fragments and the replay operator." - Which again I am not sure if its the best approach

2 ^ | v • Reply • Share ›



Dan Lew Mod → Mike Mannon • 3 years ago

I'm trying to figure out the best way to do Rx w/ the lifecycle as well. I think that so far it's an unsolved problem; as mttkay says, the issue is that RxJava is **not** opinionated, so there are many potential solutions.

I've got one friend who had a decent solution which involved using replays but without the retained fragments, but instead storing the

Observables somewhere outside of the lifecycle. Just so long as you can un/re-subscribe to an Observable and have it work the way you expect...

1 ^ | v • Reply • Share ›



Mike Mannon → Dan Lew • 3 years ago

I have just stumbled onto this:

<https://github.com/evant/rx...> (by Evan Tatarka, of retro-lambda fame)

I've not tried it, as I am still trying to understand the Retrofit & RxJava mix, but it looks like it might warrant a deeper investigation as it is "Handles Android's activity lifecycle for rxjava's Observable".

2 ^ | v • Reply • Share ›



s73v3r → Dan Lew • 3 years ago

Would a Loader be an appropriate solution? Or perhaps the Rx stuff stays out of the Activity/Fragment altogether, and lives inside a controller object, which is just a Java object owned by the Activity?

^ | v • Reply • Share ›



Andy800 • 3 years ago

Dan, I've just spent a weekend refactoring a project to try RxJava. I am questioning its error handling, and wondering how you are dealing with it. What if getTitle throws an error because the web server never responds, or it can't handle non-Unicode text, or the response is a 302 with no title, etc etc.? Once any part of the Observable throws onError, the entire rest of the stream stops. What if a page title is crazy long, so you get an IO error in saveTitle? Another error that kills the entire Observable. What if you decide you'd rather return the original URL instead of filtering out the null result? You no longer have the reference to the URL (I'm assuming you can't recode the getTitle function itself). On Page 1 you state "The Observable and Subscriber are independent of the transformational steps in between them." -- this is not true, if a transformation raises an error, the entire Observable is stopped.

You've created a great tutorial, however I am wondering how you are dealing with Rx in real life code. Thanks.

1 ^ | v • Reply • Share ›



Dan Lew Mod → Andy800 • 3 years ago

The point of `onError()` is to represent an unrecoverable state in your stream. If there is an operation which might throw an exception but you don't actually care if it does, don't let it throw `onError()`!

Another way to think of "independent transformational steps" is that, at any given moment, you could store the Observable in a variable. For example, this

is the same as some of the sample code above:

```
Observable<list<string>> step1 = query("Hello, world!");
Observable<string> step2 = step1.flatMap(urls -> Observable.from(urls));
Observable<string> step3 = step2.flatMap(url -> getTitle(url));
Observable<string> step4 = step3.filter(title -> title != null);
Observable<string> step5 = step4.take(5);
```

The examples use chaining, but really they are independent operations. As long as the type matches, I could replace any step with another step. The initial query could come from an in-memory list instead of this API call. Retrieving the title could be a hash instead of an API call. I could have one subscriber start halfway through, another subscribe at the end.

That said: I assume you're using Retrofit? This is actually somewhat of a design snafu in the library. It really ought to return `Observable<response<data>>` instead of `Observable<data>`, because non-200s are not really error states. I'm hoping that Retrofit upgrades to this at some point because right now people are having to use `onErrorResumeNext()` to solve the problem, which is not great.

^ | v • Reply • Share ›



Andy800 → Dan Lew • 3 years ago

Thanks. I do care that the operation throws an error -- but I would like to handle it (or allow the Subscriber to handle it) and have the Observable to continue to the next item, instead of shutting itself down completely. Your example simply filters out null titles instead of dealing with them, which works fine in a hypothetical, but might be unacceptable in real life.

^ | v • Reply • Share ›



Dan Lew Mod → Andy800 • 3 years ago

If you need to deal with them, you can make the example more complex. You could simply take that step out and change how it renders on screen based on null values. Or you could map null values to the empty string, so as to avoid NPEs. Or you could split your Observable in two, one with null titles, and with titles, and subscribe to that in two different ways.

It's a simple example but can be expanded upon. :)

^ | v • Reply • Share ›



Giorgos Kylafas • 3 years ago

Hi, Daniel. Great article series and one of the best I have read on the subject, particularly because you write both the full and lambda-simplified versions of the code. It also helps that you introduce new concepts very gently.

I also have a question: is the code

```
Observable.from("Hello, world!")
```

in section "It Gets Even Better" correct?
I think it should be

```
query("Hello, world!")
```

(just like in every other example).

1 ^ | v • Reply • Share >



Dan Lew Mod → Giorgos Kylafas • 3 years ago

Whoops! Thank you for catching that. Fixed now!

^ | v • Reply • Share >



Deckard 06 → Dan Lew • 2 years ago

are `Observable.from("Hello, world!")` and `query("Hello, world!")` the same thing? Are they both Observables?

^ | v • Reply • Share >



Dan Lew Mod → Deckard 06 • 2 years ago

They're both returning Observables in this case, but they are not returning the same thing - `query()` is assumed to be some sort of Observable that returns results, whereas `from()` just returns the data input.

^ | v • Reply • Share >



PnLinh • 4 days ago

Thank you so much!!! Awesome example about map and flatMap.
P/S: I bookmarked your blog

^ | v • Reply • Share >



iamtodor • a year ago

Thank you

^ | v • Reply • Share >



Aditya Pise • a year ago

Thanks.....

Really helpful. I saw 20 sites to get its basic idea. finally I understands what exactly it is.....

^ | v • Reply • Share >



shawnduan • 2 years ago

Hi Dan, thank you a lot for this awesome article! I have a question on using flatMap. Taking your code as an example, instead of printing out all titles, what if I want to build a HashMap with all (Uri, Title) pairs? Another way saying, is there a good way to refer

to both the original item (Uri) and the transformed item (title)? I know I can always build a Pair(Uri, Title) as the observable, but it getting more and more fussy when we use more and more operators during item transforming.

^ | v • Reply • Share ›



praveer09 • 2 years ago

Great article Dan... The link inside "The only limit is yourself." is awesome...

^ | v • Reply • Share ›



Zia Ulhaq • 2 years ago

thanks for the great articles. i have an question, for the example i have classes like this.

```
class Data{  
    int id  
    String name;  
    List<subdata> subdatalist;  
}
```

```
class Subdata{  
    int id;  
    String sudataName;  
    List<detail> detailList;  
}
```

```
class Detail{  
    int id;  
    String name;  
}
```

see more

^ | v • Reply • Share ›



Uk Jo • 2 years ago

```
Observable.from("url1", "url2", "url3")  
.subscribe(url -> System.out.println(url));  
this code is not working.
```

^ | v • Reply • Share ›



Dan Lew Mod ➔ **Uk Jo** • 2 years ago

I am pretty sure it does work, but only if you're compiling on Java8 or newer.

^ | v • Reply • Share ›



Darrell Merryweather ➔ **Dan Lew** • a year ago

This didn't work for me either, which is odd as I'm sure I've used something similar in the past. I'm using 1.8 and the latest RxJava libraries. I'm using eclipse, which is where is complaining. If I create a

String[] of the URLs then it works fine

^ | v • Reply • Share ›



eoin → Dan Lew • a year ago

works fine for me

^ | v • Reply • Share ›



Andrushka • 2 years ago

I guess there is a little typo:

```
.flatMap(urls -> Observable.from(urls))
```

should be

```
.flatMap(urls -> Observable.from((Iterable<string>) urls))
```

or it won't compile

^ | v • Reply • Share ›



Dan Lew Mod → Andrushka • 2 years ago

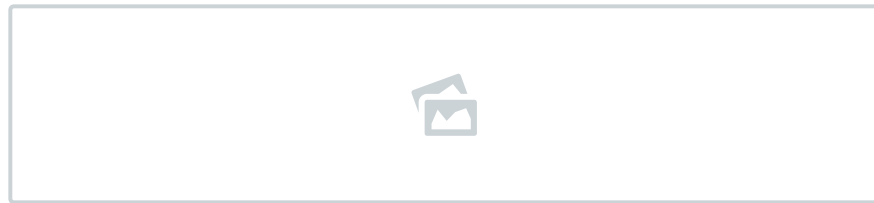
I'm pretty sure it works. This compiles just fine: <https://gist.github.com/dle...>

^ | v • Reply • Share ›



Andrushka → Dan Lew • 2 years ago

Yes. You are right. It is working. My Idea for some reason has gone crazy:



^ | v • Reply • Share ›



Dan Lew Mod → Andrushka • 2 years ago

Maybe you're compiling with the wrong version of RxJava or Java? Are you on RxJava 1.1.0 and Java 8?

^ | v • Reply • Share ›



Andrushka → Dan Lew • 2 years ago

It, despite on the warning, is compiling and working (yes I'm using fresh versions rx and java, but not last Idea version).

Thank you, Dan, for this articles!

^ | v • Reply • Share ›



shubh • 2 years ago

Why I need this:

```
getObservable(urlList).subscribe(new Action1<arraylist<string>>() {  
    public void call(ArrayList<string> urlList) {
```



```
Observable.from(urlList).subscribe(onNextAction); // Why it here!
}
});
```

instead we can simply use:

```
getObservable(urlList).subscribe(new Action1<arraylist<string>>() {
public void call(ArrayList<string> strings) {
// Do Here.
}
});
```

I mean what is use of Observable.from(..) here. Anything I'm missing at concept level !

^ | v • Reply • Share ›



Dan Lew Mod ➔ shubh • 2 years ago

I don't understand. I don't know what getObservable() is, nor what it does.

^ | v • Reply • Share ›



shubh ➔ Dan Lew • 2 years ago

I have update sample example, please have a look. thanks.

^ | v • Reply • Share ›



Dan Lew Mod ➔ shubh • 2 years ago

The point is not to use the first method NOR to use the second. The first one is bad due to unnecessary nested subscriptions, the second one *might* be appropriate but what if I want to handle one String at a time (instead of the entire list)?

^ | v • Reply • Share ›



shubh • 2 years ago

I'm bit new here...could you tell me difference between

```
Observable.from(new String[]{"url1", "url2"}).subscribe(onNextAction);
and
```

```
Observable observable = Observable.just("url1", "url2");
observable.subscribe(onNextAction);
```

Any specific purpose to use these two differently. Thanks :)

^ | v • Reply • Share ›



Dan Lew Mod ➔ shubh • 2 years ago

Those two are equivalent, though one is obviously a bit more convoluted than the other.

Generally speaking, one is used for individual items and the other for collections

collections.

  • Reply • Share ›**Daniel Gomez Rico** • 2 years ago

Is there a zip flatMap? Sometimes I needed to create a new observable from another 2 objects, getted from a zip, I mean, I have 2 web services responses needed to call another web service (retrofit), but if I use zip(w1, w2, (r1, r2 -> w3(r1, r2))) this will have the problem of the nested subscribers then... how can I do that?

  • Reply • Share ›**Dan Lew** Mod ➔ Daniel Gomez Rico • 2 years ago

I don't understand why zip(w1, w2) is bad.

  • Reply • Share ›**Daniel Gomez Rico** ➔ Dan Lew • 2 years ago

because w3 returns an observable then it needs a nested observable
zip().subscribe(observable -> observable.subscribe(desiredObject -> {}
)

  • Reply • Share ›**Dan Lew** Mod ➔ Daniel Gomez Rico • 2 years ago

Ah, I see.

It's a little weird, but you can use flatMap() to flatten the output of the Observable that was just passed in. See this sample for how to do it: <https://gist.github.com/dle...>

  • Reply • Share ›**Daniel Gomez Rico** ➔ Dan Lew • 2 years ago

will be great to have a flatZip :P but thanks

  • Reply • Share ›**Dan Lew** Mod ➔ Daniel Gomez Rico • 2 years ago

RxJava is all about composing operators together when

... ..

READ THIS NEXT

YOU MIGHT ENJOY

Grokking RxJava, Part 3: Reactive with Benefits

In part 1, I went over the basic structure of RxJava. In part 2, I showed you how powerful...

Dan Lew Codes © 2017

Grokking RxJava, Part 1: The Basics

RxJava is the new hotness amongst Android developers these days. The only problem is that it can be difficult...

Proudly published with **Ghost**