MENU

Grokking RxJava, Part 3: Reactive with Benefits

30 SEPTEMBER 2014 on rxjava

In part 1, I went over the basic structure of RxJava. In part 2, I showed you how powerful operators could be. But maybe you're still not sold; there isn't quite enough there yet to convince you. Here's some of the other benefits that come along with the RxJava framework which should seal the deal.

Error Handling

Up until this point, we've largely been ignoring `onComplete()` and `onError()`. They mark when an `Observable` is going to stop emitting items and the reason for why (either a successful completion, or an unrecoverable error).

Our original `Subscriber` had the capability to listen to `onComplete()` and `onError()`. Let's actually do something with them:

```
Observable.just("Hello, world!")
    .map(s -> potentialException(s))
    .map(s -> anotherPotentialException(s))
    .subscribe(new Subscriber<String>() {
        @Override
```

```
        public void onNext(String s) {
System.out.println(s); }

        @Override
        public void onCompleted() {
System.out.println("Completed!"); }

        @Override
        public void onError(Throwable e) {
System.out.println("Ouch!"); }
    });
```

Let's say `potentialException()` and `anotherPotentialException()` both have the possibility of throwing `Exceptions`. Every `Observable` ends with either a single call to `onCompleted()` or `onError()`. As such, the output of the program will either be a String followed by "Completed!" or it will just be "Ouch!" (because an `Exception` is thrown).

There's a few takeaways from this pattern:

1. **`onError()` is called if an `Exception` is thrown at *any* time.**

   This makes error handling much simpler. I can just handle every error at the end in a single function.

2. **The operators don't have to handle the `Exception`.**

   You can leave it up to the `Subscriber` to determine how to handle issues with any part of the `Observable` chain because `Exceptions` skip ahead to `onError()`.

3. **You know when the `Subscriber` has finished receiving items.**

Knowing when a task is done helps the flow of your code. (Though it is possible that an `Observable` may never complete.)

I find this pattern a lot easier than traditional error handling. With callbacks, you have to handle errors in each callback. Not only does that lead to repetitious code, but it also means that *each* callback must know how to handle errors, meaning your callback code is tightly coupled to the caller.

With RxJava's pattern, your `Observable` doesn't even have to know what to do with errors! Nor do any of your operators have to handle error states - they'll be skipped in cases of critical failure. You can leave all your error handling to the `Subscriber`.

Schedulers

You've got an Android app that makes a network request. That could take a long time, so you load it in another thread. Suddenly, you've got problems!

Multi-threaded Android applications are difficult because you have to make sure to run the right code on the right thread; mess up and your app can crash. The classic exception occurs when you try to modify a View off of the main thread.

In RxJava, you can tell your `Observable` code which thread to run on using `subscribeOn()`, and which thread your `Subscriber` should run on using `observeOn()`:

```
myObservableServices.retrieveImage(url)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
```

```
    .subscribe(bitmap ->
myImageView.setImageBitmap(bitmap));
```

How simple is that? Everything that runs before my `Subscriber` runs on an I/O thread. Then in the end, my View manipulation happens on the main thread[1].

The great part about this is that I can attach `subscribeOn()` and `observeOn()` to any `Observable`! They're just operators! I don't have to worry about what the `Observable` or its previous operators are doing; I can just stick this at the end for easy threading[2].

With an `AsyncTask` or the like, I have to design my code around which parts of the code I want to run concurrently. With RxJava, my code stays the same - it's just got a touch of concurrency added on.

Subscriptions

There's something I've been hiding from you. When you call `Observable.subscribe()`, it returns a `Subscription`. This represents the link between your `Observable` and your `Subscriber`:

```
Subscription subscription = Observable.just("Hello,
World!")
    .subscribe(s -> System.out.println(s));
```

You can use this `Subscription` to sever the link later on:

```
subscription.unsubscribe();
System.out.println("Unsubscribed=" +
```

```
subscription.isUnsubscribed());
// Outputs "Unsubscribed=true"
```

What's nice about how RxJava handles unsubscribing is that it stops the chain. If you've got a complex chain of operators, using `unsubscribe` will terminate wherever it is currently executing code[3]. No unnecessary work needs to be done!

Conclusion

Keep in mind that these articles are an introduction to RxJava. There's a lot more to learn than what I presented and it's not all sunshine and daisies (for example, read up on backpressure). Nor would I use reactive code for everything - I reserve it for the more complex parts of the code that I want to break into simpler logic.

Originally, I had planned for this post to be the conclusion of the series, but a common request has been for some practical RxJava examples in Android, so you can now continue onwards to part 4. I hope that this introduction is enough to get you started on a fun framework. If you want to learn more, I suggest reading the official RxJava wiki. And remember: the infinite is possible.

Many thanks to all the people who took the time to proofread these articles: Matthias Käppler, Matthew Wear, Ulysses Popple, Hamid Palo and Joel Drotos (worth the click for the beard alone).

---

[1] This is one reason why I try to keep my `Subscriber` as lightweight as possible; I want to minimize how much I block the main thread.

[2] Deferring calls to `observeOn()` and `subscribeOn()` is good practice because it gives the `Subscriber` more flexibility to handle processing as it wants. For instance, an `Observable` might take a while, but if the `Subscriber` is already in an I/O thread you wouldn't need to observe it on a new thread.

[3] In part 1 I noted that `Observable.just()` is a little more complex than just calling `onNext()` and `onComplete()`. The reason is subscriptions; it actually checks if the `Subscriber` is still subscribed before calling `onNext()`.

Dan Lew

Read more posts by this author.

Minneapolis *http://blog.danlew.net/about/*

Share this post

Greg Loesch • 3 years ago

Thanks for spending the time to write these up. As others have said, your posts (particularly the first one) finally got something to click after - somewhat casually - reading several posts by others who I think lost sight of the initial learning curve or Rx. Or perhaps they all loosened the tightly screwed on lid just to that point :)

4 • Reply • Share ›

Vijay • 3 years ago

Thanks for this article! I have a question around error handling. What if you create an observable around an event bus, and you want to continue receiving events indefinitely (i.e. never call onComplete). However, for whatever reason, you get an exception. This exception will kill your subscription. How would you gracefully handle and keep the subscription alive? Would you take the exception, and then somehow resubscribe? Would you instead *never* throw an exception from the source, and instead pass some intermediate object that encapsulates the error and returned object and let that flow through the pipeline? Would you use one of these onErrorResumeNext? if you do use onErrorResumeNext, the error gets swallowed, and your subscription is still killed..

3 • Reply • Share ›

Dan Lew Mod ➚ Vijay • 3 years ago

A common dilemma. One thing worth thinking about is that Exceptions are supposed to be that: an exception. In the reactive world, it means you're in a state where your Observable no longer works. When viewed from that perspective, it makes sense why RxJava does what it does by default.

Thus, I'd get around it by doing something like what you proposed - encapsulate the message in a way where you can detect errors and handle them properly. It's not that there's been an unrecoverable error - it's just another execution path you expect.

2 • Reply • Share ›

David COHEN ➚ Dan Lew • 2 years ago

Thank you for that :) I was trying to force rx to continue on error but you're right, it's so much cleaner to keep default rx pattern...

• Reply • Share ›

Vijay ➚ Dan Lew • 3 years ago

I can live with that :) I guess if you see things as messages vs data, then you have to ask yourself if what the definition of an exception in that case really is. Thanks for your help!

^ | v • Reply • Share ›

Artyom Gapchenko • 2 years ago

I did my best to understand what this paragraph means, but it seems that I failed:

> Deferring calls to observeOn() and subscribeOn() is good practice because it gives the Subscriber more flexibility to handle processing as it wants. For instance, an Observable might take a while, but if the Subscriber is already in an I/O thread you wouldn't need to observe it on a new thread.

What did you mean here? Does it mean that sometimes it's better not to use observeOn()/subscribeOn() at all because it's good (in someparticular case) to perform both Observable and Subscriber on the same thread?
P.S. English isn't my native language, so please, don't think that I'm mocking you.

1 ^ | v • Reply • Share ›

Dan Lew Mod ➚ Artyom Gapchenko • 2 years ago

Yes, that's what I meant. It depends on the context. If you don't need a new thread, don't use one.

2 ^ | v • Reply • Share ›

Justin Hong • 3 years ago

For error handling, how are checked exceptions propagated to the on error callback? The compiler will complain about needing a try/catch or throws declaration, so do we need to do something like try, catch, and rethrow the exception as something else?

1 ^ | v • Reply • Share ›

Dan Lew Mod ➚ Justin Hong • 3 years ago

I think it's best to think of how the Subscriber will react to an exception being thrown.

Imagine I'm calling map(), and inside of map() there's a function which could thrown an exception. If that exception grinds my call to a halt, then yes, I'd rethrow it so the Subscriber sees it. I think that's most common. But occasionally the method you're calling just means you have to use an alternative, at which point you could just keep processing even though an exception was thrown.

It gets more complicated, depending on how you want to do it. It may be easiest if your map() *always* rethrows the exception, so that it's obvious what happens, and then some other code will handle the exception explicitly.

1 • Reply • Share ›

Justin Hong ➚ Dan Lew • 3 years ago

Thanks for the answer. I found what I was looking for, which was re-throwing the exception as an OnErrorThrowable. Thanks again.

• Reply • Share ›

Luke Sleeman • 3 years ago

I suspect there is a bug in the code under:

```
Observable.just(retrieveImage(url))
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.subscribe(bitmap -> myImageView.setImageBitmap(bitmap));
```

If the activity containing bitmap is destroyed before the call to retreiveImage completes, then you will have all kinds of problems. This could happen pretty easily if the user got sick of waiting for her image and decided to press home and load another app. The OS may keep your activity around in the background or it may destroy it to reclaim memory.

Developers phones typically are high end and we only run one app on them, so you won't see it much in testing. But once your app hits the play store and gets used by people with crappy phones on slow network connections, who run loads of different apps, the error crash logs will start piling up!

Correctly dealing with background threads and the Android activity lifecycle is one of the hardest problems in Android. I would love to see some discussion of how RxJava can help!

1 • Reply • Share ›

Dan Lew Mod ➚ Luke Sleeman • 3 years ago

I'm working on part 4 which will cover this exact problem. In the meantime, cliff notes are to check out the RxAndroid project: AndroidObservable.bindActivity() helps prevent this problem. Also unsubscribe from Subscriptions properly.

• Reply • Share ›

Richa Khanna • 5 months ago

Great post Dan. I just saw something at a particular line, where I feel there is a typo mistake. You said "In RxJava, you can tell your Observer code which thread to run on using subscribeOn(), and which thread your Subscriber should run on using observeOn():" Did you mean "In RxJava, you can tell your Observable code..." Please let me know.

• Reply • Share ›

Dan Lew Mod ➚ Richa Khanna • 5 months ago

Yeah, that's a bit of a typo. I'll fix it.

Reply • Share ›

Anas Ismail • 10 months ago

Hi Dan Lew,
At first, I want to thank you for these posts. They were so valuable for me. I think there's a typo within this post. It's in the third paragraph of the "Schedulers" part. You've said there: "In RxJava, you can tell your Observer code which thread to run on using subscribeOn() ...", I think you wanna say "Observable" instead of "Observer".

Reply • Share ›

vivek • a year ago

One of the best article on Rx I have read big thanks to Dan!

Reply • Share ›

Alen Siljak • a year ago

Hi, Dan. Thanks for the handy introduction series. Any opinion on RxLifecycle and the approach of ending the sequence instead of unsubscribing?

Reply • Share ›

Dan Lew Mod ➚ Alen Siljak • a year ago

Funny you should ask that, since I basically wrote and maintain RxLifecycle.

I think it's a legitimate way to clean up memory and stop sequences. Unsubscription is only necessary in the case where you need to handle `onCompleted()` yourself.

Reply • Share ›

Alen Siljak ➚ Dan Lew • a year ago

Ahah, right! Found that out today through another article. Thanks for the contribution!

Reply • Share ›

Prasanth • a year ago

Awesome post

Reply • Share ›

Matt • 3 years ago

Thanks Dan, both for this excellent guide and linking the life-changing Zombo com... :) I'm not able to build code equivalent to your first example due to an "unhandled exception" error around potentialException()/anotherUnhandledException(). I was able to work around it by wrapping the method call in a new observable and calling onError(), but this adds a bunch of ugly code - I'd much rather have any exceptions passed along to the subscriber at the end of the chain as you've said. Is there something wrong with my setup? This is using RetroLambda for Android development if that's

pertinent.

Reply • Share ›

Dan Lew Mod → Matt • 3 years ago

If your method throws a checked exception you still have to handle it within `map()`.

It's just runtime exceptions which will automatically be forwarded to onError().

Reply • Share ›

Mohamad Atie • 3 years ago

I think it is a good idea to highlight that operators can also return super classes when it's "? super String". I found it useful when handling errors if the mapped item like in this example is not a String, but instead of the Throwable type

Reply • Share ›

Andy Dennie • 3 years ago

Not 100% sure, since I'm a total RxNoob at this point, but I think that where you say "In RxJava, you can tell your Observer code which thread to run on using subscribeOn()...", you mean: "In RxJava, you can tell your Observable code ...:

READ THIS NEXT

Grokking RxJava, Part 4: Reactive Android

In parts 1, 2, and 3 I covered how RxJava works (in a general sense). But as an Android…

YOU MIGHT ENJOY

Grokking RxJava, Part 2: Operator, Operator

In part 1 I went over the basic structure of RxJava, as well as introducing you to the map(…