RxJava/RxAndroid

Montag, 27. Dezember 2021 14:58

Links:

Author: Damir Ljubic
email: damirlj@yahoo.com

ReactiveX · GitHub
https://github.com/ReactiveX/RxJava
ReactiveX - Observable
GitHub - JakeWharton/RxBinding: RxJava binding APIs for Android's UI widgets.
CodingWithMitch.com
Running Android tasks in background threads | Android Developers

Intro

Imperative approach

```
public static List<User> filterAdmin() {
   List<User> admins = new ArrayList<>();
   final List<User> users = getUsers();
   for (User user : users) {
       if (user.userType == UserType.ADMIN) {
           admins.add(user);
       }
   }
   return admins;
}
```

The code has a lot of flows and violate a numerous rules of *SOLID* programming

- Filter functionality can be extracted into generic method, that accepts any iterable collection and unary predicate (**DRY**)
- It's difficult to extend this with another requirements, without direct changing the code (**OCP**)
- It's difficult to employ **a new thread context** in case that *getUsers*() call results in fetching the data from DB, especially if the DB is hosted on some network server (blocking network call)

Declarative approach
Java streams (Java SE8)

```
public static List<User> filterAdmin() {
  return getUsers().stream()
                   .filter(user -> user.userType == UserType.ADMIN)
                   .collect(Collectors.toList());
}
```

It's **Iterator** design pattern approach (the iterators are used to traversal over the underlying collection, without exposing the collection itself: the same algorithm applies on any collection that implements *Iterable* interface): the value will be pulled out from iterator and process.
It's ***Functional Programming*** *approach*, with fallowing characteristics:

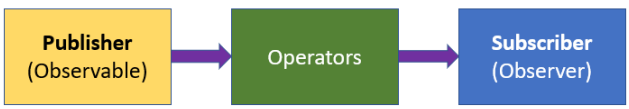
- Reach operators support, that act as a *pure functions*: without any side-effects.
  This means, they don't change neither the input arguments, nor they modify external - out of scope variables.
  Being immutable in that sense, make them thread-safe.
- The output of the one operation is the input into next one in operation chain, until terminal operation is reached.
- The code is therefore composable - easier to extend, in terms of adding new requirements (**OCP**): by converting the result back into the stream, and apply new operators

```
public static List<String> getAdminsNames() {
    return filterAdmin().stream()
                        .map(User::toString)
                        .collect(Collectors.toList());
}
```

ReactiveX

ReactiveX - *Reactive Extension* is the **library** that in essence implements the **Observer (Publisher-Subscriber) pattern**.
It provides the way of having **asynchronous event-based communication**, by completely decoupling the publisher - *Observable*: the side in communication which emits the items (events), from the subscriber - *Observer*: the **callbacks interface** that will be invoked upon receiving the emitted item (event).
Observer doesn't know who, when or how many times will emit the items - it only knows how to react on them.

Everything between the Publisher and Subscriber is reactive logic - **processing chain of operations** - *Operators*, that usually perform some kind of transformation on emitted items down to stream, until they reach the Subscriber.



RxJava

It comes in different flavors - there are different language specific implementation of the library.
RxJava is the Java-specific implementation of ReactiveX library.
It has the all benefits of Java stream (similar syntax), with additional advantages:

- integrated **threading support**
- laziness: you don't pay for something that you don't use
  - **lazy by emission** - the observable will start to emit the items (events) with first active subscription (Cold Observables)
  - **lazy by creation** - the observable will create the publishing sequence first upon subscription (Cold & lazy-by-creation Observables)
- almost anything can be turned into Observable (not only Iterable Collections), using one of the factory methods (https://github.com/ReactiveX/RxJava/blob/3.x/docs/Creating-Observables.md)
- **Error handling**: instead of using try/catch blocks as with imperative approach, the errors (exceptions) will be propagated down the stream to Observers

Getting back to our example, with Rx it becomes

```
private Observable<List<User>> fetchUsers() {
  return Observable.fromCallable(this::getUsers);
```

```
}


public Observable<List<String>> fetchUsersWithFilter(Predicate<User> condition) {
  return fetchUsers().subscribeOn(Schedulers.io()) // employing background thread for fetching/publishing items, including the entire processing chain of operations
          .doOnNext((users) -> Log.d(TAG, "<RxAndroid> Fetching users on thread: " + Thread.currentThread().getName())) // logging
          .flatMap(Observable::fromIterable) // converting the List<User> into Observable<User>, so that each iterable item can be processed separately down the stream
          .filter(condition)
          .map(User::toString)
          .toList() // aggregate operator: block until the all emitted items are processed, aggregating them into Single<List<String>>
          .toObservable(); // convert the Single<List<String>> into required, Observable<List<String>> more generic form
}

public Disposable subscribeToFetchUsers(Observable<List<User>> users) {
  return users.observeOn(AndroidSchedulers.mainThread())
              .subscribe(this::updateUsersList);
}
```

Laziness

There are two types of Observables, in terms of how they emit the items:

- **Cold Observables**: emit the items "lazy" - only after being subscribed for.
  Upon each subscription the complete sequence will be **reemitted** - each subscriber gets its own stream.
  Creating Observables · ReactiveX/RxJava Wiki · GitHub

  Some factory methods

Factory method	Lazy creation
.create	☑
.just	☐
.fromIterable	☐
.fromArray	☐
.fromCallable	☑
.fromRunnable	☑
.fromAction	☑
.interval	☐
.range	☐
.defer	☑

  Lazy evaluation: the Cold Observables - their factory methods, distinguish themselves, at which point the underlying sequence will be created.
  - For the eager one, the sequence will be created immediately, upon the factory call.
  - For the lazy one, the creation of the sequence to be published is postponed - until subscription occurs.

  "eager-by-creation" -> "lazy-by-creation"
  To turn "eager" Observable, into "lazy" one, we can use **.defer factory method**

```
public Observable<User> createFromUsersListDeferred() {
    return Observable.defer(this::fetchUsersFromNetwork);
}
```

  Drawback is that with each subscription, the new - dedicated Observable will be created, computing
  the each time the underlying sequence for the new publishing stream - for the new subscriber.
  Also, if the sequence is produced as result of some network API call, it may happen that the
  sequence will change (due to DB being changed), which means that not all subscribers will receive the
  updated sequence - observe the same stream.

- **Hot Observables**: independently from Observers, emits the items immediately after they've arrived (UI events: button clicks, etc.).
  Upon the new subscription, the previous emission will not be reemitted.
  Obviously, they don't produce the data - they are driven outside, an act more as an event dispatchers.

Cold->Hot

For certain scenarios, to prevent backpressure (as result of reemitting a huge sequence of data, having producer emits the items with much higher ratio, than consumer can observe them) - to enforce the **multicasting** of the items without reemission the "history", having single shared stream among the Observers.

ConnectableObservable can be used to turn the cold into hot observable

```
public static <T> CompositeDisposable test_cold2hot(Observable<T> cold, List<Consumer<? super T>> consumers) {
   CompositeDisposable subscriptions = new CompositeDisposable();
   ConnectableObservable<T> hot = cold.publish();// turn cold into hot observable
   for (Consumer<? super T> consumer : consumers) {
       subscriptions.add(hot.subscribe(consumer));
   }
   hot.connect(); // trigger point - starts to emit items
   return subscriptions;
}
```

Operators

Alphabetical List of Observable Operators · ReactiveX/RxJava Wiki · GitHub
https://proandroiddev.com/exploring-rxjava-in-android-operators-for-combining-observables-25734080f4be

There is common misconception on Rx operator, that they are inherently asynchronous, they aren't - they are by **default synchronous**.
The asynchronous in Rx processing chain could be
- the arbitrary emissions in time, driven by some external events (like User-actions)
- providing for the processing chain separate thread context from the application main thread - to keep the application responsive.

Transformation operators

https://github.com/ReactiveX/RxJava/blob/3.x/docs/Transforming-Observables.md

The transforming operators (monads), transformed emitted item of type **T** into

- Another type **U**
  f(T)->U

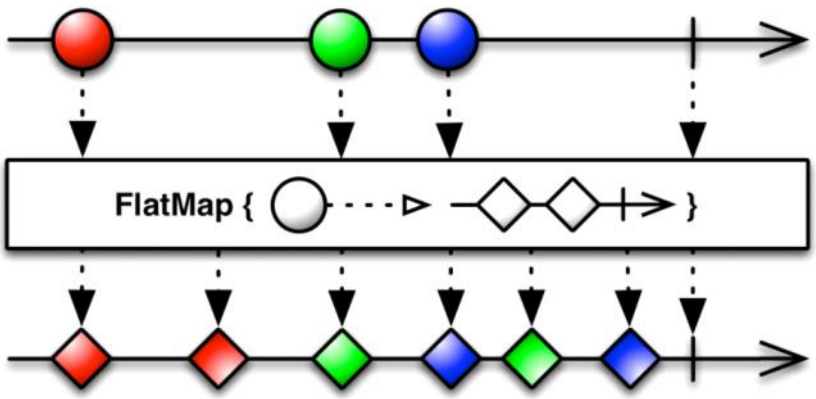
```
Observable<List<Person>> persons = getPersons();
persons.map(listOfPersons->{
    List<String> names = new ArrayList<>();
    for(person: listOfPersons) {
        names.add(person.getName());
    }
    return names;// List<String>
}).subscribe(this::updateNames);
```

- Another Publisher: **Observable<U>**

  f(T)->Observable<U>

```
Observable<List<Person>> persons = getPersons();
persons.flatMap( (Function<List<Person>, Observable<List<String>>>) listOfPersons -> {
    return Observable.fromIterable(listOfPersons) // Observable<Person>
            .map(Person::getName)
            .doOnNext(name->Log(TAG, "Name=" + name))
            .toList() // Single<List<String>>
            .toObservable(); // Observable<List<String>>
})
.subscribe(this::updateNames); // This will crash, since touch the UI control: see the threading section!
```

.flatMap



Marble diagram (source: https://reactivex.io/documentation/operators/flatmap.html)

It transforms each emitted item T into Observable<U>, and flatten emission of all these single Observables into one- resulting Observable.
This way, we don't need to subscribe/unsubscribe to each of this individual Observable, but rather only on the resulting one.
The emission of the individual sources is **interleaved** - there is no guaranties that the original order will be preserved.

Observer (Subscriber)

https://reactivex.io/documentation/operators/subscribe.html

Any Subscriber needs to implement the fallowing callbacks interface

```
public interface Observer<T> {
    void onSubscribe(@NotNull Disposable d);
    void onNext(@NotNull T data);
    void onError(@NotNull Throwable error);
    void onComplete();
};
```

There are different overloaded versions of the **.subscribe() operator**, that enable us to specify
the subscriber, as fully, or partially implementation of the given interface

```
Observable<T> publisher = ...;
Disposable subscription =
    publisher.op1(t->u)
             .op2(u)
             .subscribe(// overloaded version of Observable<T>.subscribe
                this::updateCallback,
                this::errorCallback,
                this::completionCallback
        );
```

For example, to specifying the callable that will be invoked upon receiving the emitted item, leaving the error handling and completion notification uncovered

```
.subscribe(Consumer<? super T> callback)

Observable<T> publisher = ...;
Disposable subscription =
    publisher.op1(t->u)
             .op2(u)
             .subscribe(this::updateCallback);
```

Disposable has a meaning of the reference to the subscription- there is also *CompositeDisposable*, for referencing the collection of related subscriptions.
Usually, the enclosing class will hold the reference on the subscription(MainActivity.onCreate), and unsubscribe - detach the subscribers from receiving the updates on the publisher state from some finalization method (MainActivity.onDestroy)

```
if (!subscription.isDisposed()) {
    subscription.dispose();
}
```

Schedulers - threading

https://medium.com/android-news/rxjava-schedulers-what-when-and-how-to-use-it-6cfc27293add
https://www.baeldung.com/rxjava-schedulers

Talking on decoupling subscriber from publisher, there is even possibility to specify the different thread context in which the Publisher will emit the items (events), from the one in which the Subscriber will observe them.
We distinguish

- ***subscribeOn*** - is the way for specifying thread context in which the Publisher will emit the items, and if not other way specified, the context in which the Subscriber will observe them.
  This also affects the processing chain of operations, independent from the operator position, unless thread context switch occurs (@see *observeOn*).
  It should be call only once - any subsequent call of this operator will be ignored - have no effect.

- ***observeOn*** - is the way to specify the thread context in which the Subscriber will observe- receive the emitted items.
  It can be called multiple times, causing each time the thread context switch, affecting only the *below* chained operations.

  This is especially important for the interactive applications, in case that we need to access the UI controls (widgets), since this is allowed only within the **UI - main thread.**

```
Observable<List<Person>> persons = getPersons();
persons.flatMap( (Function<List<Person>, Observable<List<String>>>) listOfPersons -> {
    return Observable.fromIterable(listOfPersons) // Observable<Person>
            .map(Person::getName)
            .doOnNext(name->Log(TAG, "Name=" + name))
            .toList() // Single<List<String>>
            .toObservable(); // Observable<List<String>>
})
.observeOn(AndroidSchedulers.mainThread())// This will prevent the crash - it will observed in Android main thread
.subscribe(this::updateNames); // listed the names in UI control
```

Schedulers class provides different kind of factory methods - to obtain different kind of schedulers

io()
It's related to the unbounded thread-pool (virtual threads)
This thread pool is intended to be used for asynchronously performing blocking IO.
You would typically use Schedulers.io() for network/filesystem related tasks that doesn't drain too much CPU power.

single()
This will return each time - on each subscription, the very same background thread.
That means, calling it with - for example - observeOn

```
Observable1.observeOn(Schedulers.single()).subscribe(…);
Observable2.observeOn(Schedulers.single()).subscribe(…);
```

will result that all subscribers, even those that are subscribed to a different publishers - will be sequentially executed within the same thread context.

newThread()
As the name implies - it will create a new thread, on each subscription: without reusing existing thread-pool of threads.
Each new thread can be monitored as *"RxNewThreadScheduler-<indices>"*

computation()
This scheduler is related to bounded thread-pool (hardware threads): thread pool that will have as much spawned threads as we have cores in the system.
It's for some computational jobs that require a lot of CPU power: non-blocking, non-frequent, non-time consuming algorithmic-like jobs.

immediate()
This scheduler is uncommon - will result the executing the subscribed callable within the clients thread.
This can be useful, where you have a generic API which requires the Scheduler to be specified: but you actually don't want to specify a new one - but rather force synchronous execution.

trampoline()
Similar as immediate() - it doesn't introduce another concurrency by itself, but rather relies on the hosting - clients thread, on which publisher will emit the items/events.
This factory method will create scheduler that will queue all tasks (operators) in the reactive chain, so that they will be executed sequentially, one after another - even if these tasks are meant to be asynchronous, on the client thread.

from()
For using custom-specific Scheduler: backed by the custom-specific Executor.
We can rely on the ready-to use one, provided by the Java library (*Java.util.concurrent.Executors*)

```
Schedulers.from(Executors.newSingleThreadExecutor())
```

To specify the Executor backed with limited number of threads in the thread pool

```
Scheduler.from(Executors.newFixedThreadPool(n))
```

Subjects

ReactiveX - Subject

Subjects are Rx entities that have **dual nature** - can act as an Observable, and at the same time as Observer as well.
This comes from the implementation details

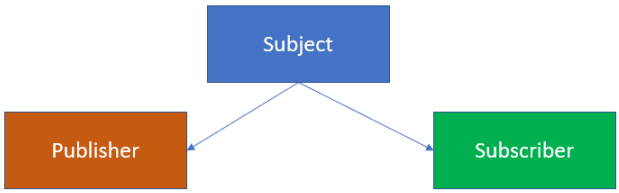
```
class Subject<T> extends Observable<T>
                 implements Observer<T>
{...}
```

They are used to **bridge the observable with observer(s)**, in case that it's inconvenient to subscribe Observer(s) directly to the Observable

- for multicasting the resulting sequence to the all subscribed Observers, without repeating some expensive operations in processing chain

- ***PublishSubject*** **(stateless)**
  Act as an *event dispatcher*: just emits the item (event), independent on the active subscription (*Hot Observable*).
  It doesn't store the most latest emitted state internally, so it can't reemit it upon new subscription.

- ***BehaviorSubject*** **(stateful)**
  Opposite to *PublishSubject*, the most latest emitted state is preserved - internally cached.
  With new subscription, the most latest state (or default one - if specified, if no value is emitted yet) will be reemitted.



Subject as Subscriber:
Observable<T>observable = ...
observable.subscribe(Subject<T> subject)

Subject as Publisher:
subject.observeOn(Schedulers.io())
.subscribe(Consumer<? super T> subscriber)

```
BehaviorSubject<UserType> usersType = BehaviorSubject.create();

Observable<UserType> fetchAdminsObservable = RxView.clicks(fetchAdminsButton).map(e->UserType.ADMIN);
Observable<UserType> fetchOthersObservable = RxView.clicks(fetchOthersButton).map(e->UserType.NORMAL);
Observable.merge(fetchAdminsObservable, fetchOthersObservable)
          .subscribe(usersType); // Subscriber
```

```
subscription = usersType // Publisher
        .switchMap(type->TestRxJava.fetchUsersWithFilter(user -> user.userType == type))
        .observeOn(AndroidSchedulers.mainThread())// Switch to the UI main thread
        .subscribe(names -> { // Listener: update the UI control, with the newly fetched users
                usersAdapter.clear();
                usersAdapter.addAll(names);
        });
```

- ***ReplaySubject***
  As the name implies, it caches the entire published sequence and reemit it upon each subscription

- ***AsyncSubject***
  Emits only the last item, upon the completion

Importing the Rx to Android Studio

Into application **app/build.gradle**, add the fallowing lines, in order to link to the Rx related libraries

```
dependencies {

        def rxJavaVersion='3.0.11-RC3'
        def rxAndroidVersion='3.0.0'
        def rxBindingVersion='4.0.0'

        // RxJava
        implementation "io.reactivex.rxjava3:rxjava:${rxJavaVersion}"

        // RxAndroid
        implementation "io.reactivex.rxjava3:rxandroid:${rxAndroidVersion}"

        // RxBinding - for reactive UI controls
        implementation "com.jakewharton.rxbinding4:rxbinding:${rxBindingVersion}"

        ...

  }
```

The Android provides its own way of having GUI controls which can act "reactively" - as event publishers, with addition that they are lifecycle-aware: emit only to the active Observers (Activities, Fragments, Services).
This way, Observer doesn't need to explicitly unsubscribe from updates - when subscriber is gone, preventing calling subscriber with invalid reference.
LiveData Overview | Android Developers