Introduction

RxJava features over 100 operators to support the most common reactive dataflow patterns. Generally, there exist a combination of operators, typically `flatMap`, `defer` and `publish`, that allow composing less common patterns with standard guarantees. When you have an uncommon pattern and you can't seem to find the right operators, try asking about it on our issue list (or Stackoverflow) first.

If none of this applies to your use case, you may want to implement a custom operator. Be warned that **writing operators is hard**: when one writes an operator, the `Observable` **protocol**, **unsubscription**, **backpressure** and **concurrency** have to be taken into account and adhered to the letter.

Note that this page uses Java 8 syntax for brevity.

Considerations

Observable protocol

The `Observable` protocol states that you have to call the `Observer` methods, `onNext`, `onError` and `onCompleted` in a sequential manner. In other words, these can't be called concurrently and have to be **serialized**. The `SerializedObserver` and `SerializedSubscriber` wrappers help you with these. Note that there are cases where this serialization has to happen.

In addition, there is an expected pattern of method calls on `Observer`:

```
onNext* (onError | onCompleted)?
```

A custom operator has to honor this pattern on its push side as well. For example, if your operator turns an `onNext` into an `onError`, the upstream has to be stopped and no further methods can be called on the dowstream.

Unsubscription

The basic `Observer` method has no direct means to signal to the upstream source to stop emitting events. One either has to get the `Subscription` that the `Observable.subscribe(Observer<T>)` returns **and** be asynchronous itself.

This shortcoming was resolved by introducing the `Subscriber` class that implements the `Subscription` interface. The interface allows detecting if a `Subscriber` is no longer interested in the events.

```
interface Subscription {
   boolean isUnsubscribed();

   void unsubscribe();
}
```

In an operator, this allows active checking of the `Subscriber` state before emitting an event.

In some cases, one needs to react to the child unsubscribing immediately and not just before an emission. To support this case, the `Subscriber` class has an `add(Subscription)` method that let's the operator register `Subscription`s of its own which get unsubscribed when the downstream calls `Subscriber.unsubscribe()`.

```
InputStream in = ...

child.add(Subscriptions.create(() -> {
    try {
       in.close();
    } catch (IOException ex) {
       RxJavaHooks.onError(ex);
    }
}));
```

Backpressure

The name of this feature is often misinterpreted. It is about telling the upstream how many `onNext` events the downstream is ready to receive. For example, if the downstream requests 5, the upstream can only call `onNext` 5 times. If the upstream can't produce 5 elements but 3, it should deliver that 3 element followed by an `onError` or `onCompleted` (depending on the operator's purpose). The requests are cumulative in the sense that if the downstream requests 5 and then 2, there is going to be 7 requests outstanding.

Backpressure handling adds a great deal of complexity to most operators: one has to track how many elements the downstream requested, how many have been delivered (by usually subtracting from the request amount) and sometimes how many elements are still available (but can't be delivered without requests). In addition, the downstream can request from any thread and is not required to happen on the common thread where otherwise the `onXXX` methods are called.

The backpressure 'channel' is established between the upstream and downstream via the `Producer` interface:

```
interface Producer {
    void request(long n);
}
```

When an upstream supports backpressure, it will call the `Subscriber.setProducer(Producer)` method on its downstream `Subscriber` with the implementation of this interface. The downstream then can respond with `Long.MAX_VALUE` to start an unbounded streaming (effectively no backpressure between the immediate upstream and downstream) or any other positive value. A request amount of zero should be ignored.

Protocol-vise, there is no strict time when a producer can be set and it may never appear. Operators have to be ready to deal with this situation and assume the upstream runs in unbounded mode (as if `Long.MAX_VALUE` was requested).

Often, operators may implement `Producer` and `Subscription` in a single class to handle both requests and unsubscriptions from the downstream:

```
final class MyEmitter implements Producer, Subscription {
    final Subscriber<Integer> subscriber;

    public MyEmitter(Subscriber<Integer> subscriber) {
        this.subscriber = subscriber;
    }

    @Override
    public void request(long n) {
        if (n > 0) {
           subscriber.onCompleted();
        }
    }

    @Override
    public void unsubscribe() {
        System.out.println("Unsubscribed");
    }

    @Override
    public boolean isUnsubscribed() {
        return true;
    }
}

MyEmitter emitter = new MyEmitter(child);

child.add(emitter);
child.setProducer(emitter);
```

Unfortunately, you can't implement `Producer` on a `Subscriber` because of an API oversight: `Subscriber` has a protected final `request(long n)` method to perform **deferred requesting** (store and accumulate the local request amounts until `setProducer` is called).

Concurrency

When writing operators, we mostly have to deal with concurrency via the standard Java concurrency primitives: `AtomicXXX` classes, volatile variables, `Queue`s, mutual exclusion, Executors, etc.

RxJava tools

RxJava has a few support classes and utilities that let's one deal with concurrency inside operators.

The first one, `BackpressureUtils` deals with managing the cumulative requested and produced element counts for an operator. Its `getAndAddRequested()` method takes an `AtomicLong`, accumulates request amounts atomically and makes sure they don't overflow `Long.MAX_VALUE`. Its pair `produced()` subtracts the amount operators have produced, thus when both are in play, the given `AtomicLong` holds the current outstanding request amount for the downstream.

Operators sometimes have to switch between multiple sources. If a previous source didn't fulfill all its requested amount, the new source has to start with that unfulfilled amount. Otherwise as the downstream didn't receive the requested amount (and no terminal event either), it can't know when to request more. If this switch happens at an `Observable` boundary (think `concat`), the `ProducerArbiter` helps managing the change.

If there is only one item to emit eventually, the `SingleProducer` and `SingleDelayedProducer` help work out the backpressure handling:

```
child.setProducer(new SingleProducer<>(child, 1));

// or

SingleDelayedProducer<Integer> p = new SingleDelayedProducer<>(child);

child.add(p);
child.setProducer(p);

p.setValue(2);
```

The queue-drain approach

Usually, one has to serialize calls to the `onXXX` methods so only one thread at a time is in any of them. The first thought, namely using `synchronized` blocks, is forbidden. It may cause deadlocks and unnecessary thread blocking.

Most operators, however, can use a non-blocking approach called queue-drain. It works by posting the element to be emitted (or work to be performed) onto a **queue** then atomically increments a counter. If the value before the increment was zero, it means the current thread won the right to emit the contents of the queue. Once the queue is **drained**, the counter is decremented until zero and the thread continues with other activities.

In code:

```
final AtomicInteger counter = new AtomicInteger();
final Queue<T> queue = new ConcurrentLinkedQueue<>();
```

```
public void onNext(T t) {
    queue.offer(t);
    drain();
}

void drain() {
    if (counter.getAndIncrement() == 0) {
        do {
           t = queue.poll();
           child.onNext(t);
        } while (counter.decrementAndGet() != 0);
    }
}
```

Often, the when the downstream requests some amount, that should also trigger a similar drain() call:

```
final AtomicLong requested = new AtomicLong();

@Override
public void request(long n) {
    if (n > 0) {
        BackpressureUtils.getAndAddRequested(requested, n);
        drain();
    }
}
```

Many operators do more than just draining the queue and emitting its content: they have to coordinate with the downstream to emit as many items from the queue as the downstream requested.

For example, if one writes an operator that is unbounded-in but honors the requests of the downstream, the following `drain` pattern will do the job:

```
// downstream's consumer
final Subscriber<? super T> child;
// temporary storage for values
final Queue<T> queue;
// mutual exclusion
final AtomicInteger counter = new AtomicInteger();
// tracks the downstream request amount
final AtomicLong requested = new AtomicLong();

// no more values expected from upstream
volatile boolean done;
// the upstream error if any
```

```
Throwable error;

void drain() {
    if (counter.getAndIncrement() != 0) {
        return;
    }

    int missed = 1;
    Subscriber<? super T> child = this.child;
    Queue<T> queue = this.queue;

    for (;;) {
         long requests = requested.get();
         long emission = 0L;

         while (emission != requests) { // don't emit more than requested
             if (child.isUnsubscribed()) {
                 return;
             }

             boolean stop = done;  // order matters here!
             T t = queue.poll();
             boolean empty = t == null;

             // if no more values, emit an error or completion event
             if (stop && empty) {
                 Throwable ex = error;
                 if (ex != null) {
                     child.onError(ex);
                 } else {
                     child.onCompleted();
                 }
                 return;
             }
             // the upstream hasn't stopped yet but we don't have a value available
             if (empty) {
                 break;
             }

             child.onNext(t);
             emission++;
         }

         // if we are at a request boundary, a terminal event can be still emitted without t
         if (emission == requests) {
             if (child.isUnsubscribed()) {
```

```
                return;
            }

            boolean stop = done;  // order matters here!
            boolean empty = queue.isEmpty();

            // if no more values, emit an error or completion event
            if (stop && empty) {
                Throwable ex = error;
                if (ex != null) {
                    child.onError(ex);
                } else {
                    child.onCompleted();
                }
                return;
            }
        }

        // decrement the current request amount by the emission count
        if (emission != 0L && requests != Long.MAX_VALUE) {
            BackpressureUtils.produced(requested, emission);
        }

        // indicate that we have performed the outstanding amount of work
        missed = counter.addAndGet(-missed);
        if (missed == 0) {
            return;
        }
        // if a concurrent getAndIncrement() happened, we loop back and continue
    }
}
```

Creating source operators

One creates a source operator by implementing the `OnSubscribe` interface and then calls `Observable.create` with it:

```
OnSubscribe<T> onSubscribe = (Subscriber<? super T> child) -> {
    // logic here
};

Observable<T> observable = Observable.create(onSubscribe);
```

Note: a common mistake when writing an operator is that one simply calls `onNext` disregarding backpressure; one should use `fromCallable` instead for synchronously (blockingly) generating a single value.

The `logic here` could be arbitrary complex logic. Usually, one creates a class implementing `Subscription` and `Producer`, sets it on the `child` and works out the emission pattern:

```
OnSubscribe<T> onSubscribe = (Subscriber<? super T> child) -> {
    MySubscription mys = new MySubscription(child, otherParams);
    child.add(mys);
    child.setProducer(mys);

    mys.runBusinessLogic();
};
```

Converting a callback-API to reactive

One of the reasons custom sources are created is when one converts a classical, callback-based 'reactive' API to RxJava. In this case, one has to setup the callback on the non-RxJava source and wire up unsubscription if possible:

```
OnSubscribe<Data> onSubscribe = (Subscriber<? super Data> child) -> {
    Callback cb = event -> {
        if (event.isSuccess()) {
            child.setProducer(new SingleProducer<Data>(child, event.getData()));
        } else {
            child.onError(event.getError());
        }
    };

    Closeable c = api.query("someinput", cb);

    child.add(Subscriptions.create(() -> Closeables.closeQuietly(c)));
};
```

In this example, the `api` takes a callback and returns a `Closeable`. Our handler signals the data by setting a `SingleProducer` of it to deal with downstream backpressure. If the downstream wants to cancel a running API call, the wrap to `Subscription` will close the query.

However, in case the callback is called more than once, one has to deal with backpressure a different way. At this level, perhaps the most easiest way is to apply `onBackpressureBuffer` or `onBackpressureDrop` on the created `Observable`:

```
OnSubscribe<Data> onSubscribe = (Subscriber<? super Data> child) -> {
    Callback cb = event -> {
        if (event.isSuccess()) {
            child.onNext(event.getData());
        } else {
            child.onError(event.getError());
```

```
        }
    };

    Closeable c = api.query("someinput", cb);

    child.add(Subscriptions.create(() -> Closeables.closeQuietly(c)));
};

Observable<T> observable = Observable.create(onSubscribe).onBackpressureBuffer();
```

Creating intermediate operators

Writing an intermediate operator is more difficult because one may need to coordinate request amount between the upstream and downstream.

Intermediate operators are nothing but `Subscriber`s themselves, wrapping the downstream `Subscriber` themselves, modulating the calls to `onXXX`methods and they get subscribed to the upstream's `Observable`:

```
Func1<T, R> mapper = ...

Observable<T> source = ...

OnSubscribe<R> onSubscribe = (Subscriber<? super R> child) -> {

    source.subscribe(new MapSubscriber<T, R>(child) {
        @Override
        public void onNext(T t) {
            child.onNext(function.call(t));
        }

        // ... etc
    });

}
```

Depending on whether the safety-net of the `Observable.subscribe` method is too much of an overhead, one can call `Observable.unsafeSubscribe` but then the operator has to manage and unsubscribe its own resources manually.

This approach has a common pattern that can be factored out - at the expense of more allocation and indirection - and became the `lift` operator.

The `lift` operator takes an `Observable.Operator<R, T>` interface implementor where `R` is the output type towards the downstream and `T` is the input type from the upstream. In our example, we can rewrite the operator as follows:

```
Operator<R, T> op = child ->
    return new MapSubscriber<T, R>(child) {
        @Override
        public void onNext(T t) {
            child.onNext(function.call(t));
        }

        // ... etc
    };
}

source.lift(op)...
```

The constructor of `Subscriber(Subscriber<?>)` has some caveats: it shares the underlying resource management between `child` and `MapSubscriber`. This has the unfortunate effect that when the business logic calls `MapSubscriber.unsubscribe`, it may inadvertently unsubscribe the `child`'s resources prematurely. In addition, it sets up the `Subscriber` in a way that calls to `setProducer` are forwarded to the `child` as well.

Sometimes it is acceptable, but generally one should avoid this coupling by implementing these custom `Subscriber`s among the following pattern:

```
public final class MapSubscriber<T, R> extends Subscriber<T> {
    final Subscriber<? super R> child;

    final Function<T, R> mapper;

    public MapSubscriber(Subscriber<? super R> child, Func1<T, R> mapper) {
         // no call to super(child) !
         this.child = child;
         this.mapper = mapper;

         // prevent premature requesting
         this.request(0);
    }

    // setup the unsubscription and request links to downstream
    void init() {
         child.add(this);
         child.setProducer(n -> requestMore(n));
    }

    @Override
    public void onNext(T t) {
         try {
             child.onNext(mapper.call(t));
```

```
            } catch (Throwable ex) {
                Exceptions.throwIfFatal(ex);
                // if something crashed non-fatally, unsubscribe from upstream and signal the c
                unsubscribe();
                onError(ex);
            }
        }

        @Override
        public void onError(Throwable e) {
            child.onError(e);
        }

        @Override
        public void onCompleted() {
            child.onCompleted();
        }

        void requestMore(long n) {
            // deal with the downstream requests
            this.request(n);
        }
    }

    Operator<R, T> op = child -> {
        MapSubscriber<T, R> parent = new MapSubscriber<T, R>(child, mapper);
        parent.init();
        return parent;
    }
```

Some operators may not emit the received value to the `child` subscriber (such as filter). In this case, one has to call `request(1)` to ask for a replenishment because the downstream doesn't know about the dropped value and won't request itself:

```
// ...

        @Override
        public void onNext(T t) {
            try {
                if (predicate.call(t)) {
                    child.onNext(t);
                } else {
                    request(1);
                }
            } catch (Throwable ex) {
                Exceptions.throwIfFatal(ex);
```

```
            unsubscribe();
            onError(ex);
        }
    }

// ...
```

When an operator maps an `onNext` emission to a terminal event then before calling the terminal event it should unsubscribe the subscriber to upstream (usually called the parent). In addition, because upstream may (legally) do something like this:

```
child.onNext(blah);
//  no check for unsubscribed here
child.onCompleted();
```

we should ensure that the operator complies with the `Observable` contract and only emits one terminal event so we use a defensive done flag:

```
boolean done; // = false;

@Override
public void onError(Throwable e) {
    if (done) {
        return;
   }
   done = true;
   ...
}

@Override
public void onCompleted(Throwable e) {
    if (done) {
        return;
   }
   done = true;
   ...
}
```

An example of this pattern is seen in `OnSubscribeMap`.

Further reading

Writing operators that consume multiple source `Observable`s or produce to multiple `Subscriber`s are the most difficult one to implement.

For inspiration, see the blog posts of @akarnokd about the RxJava internals. The reader is advised to read from the very first post on and keep reading in

sequence.