Extracted from:

Reactive Programming with RxJS

Untangle Your Asynchronous JavaScript Code

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Reactive Programming with RxJS

Untangle Your Asynchronous JavaScript Code

Sergi Mansilla

edited by Rebecca Gulick

Reactive Programming with RxJS

Untangle Your Asynchronous JavaScript Code

Sergi Mansilla

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Rebecca Gulick (editor)
Potomac Indexing, LLC (index)
Candace Cunningham (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For sales, volume licensing, and support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Printed in the United States of America.
ISBN-13: 978-1-68050-129-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P2.0—August 2016

CHAPTER 5

Bending Time with Schedulers

As soon as I discovered RxJS, I started using it in my projects. For a while I thought I knew how to use it effectively, but there was a nagging question: how do I know whether the operator I'm using is synchronous or asynchronous? In other words, when exactly do operators emit notifications? This seemed a crucial part of using RxJS correctly, but it felt a bit blurry to me.

The `interval` operator, I thought, is clearly asynchronous, so it must use something like `setTimeout` internally to emit items. But what if I'm using `range`? Does it emit asynchronously as well? Does it block the event loop? What about `from`? I was using these operators everywhere, but I didn't know much about their internal concurrency model.

Then I learned about Schedulers.

Schedulers are a powerful mechanism to precisely manage concurrency in your applications. They give you fine-grained control over how an Observable emits notifications by allowing you to change their concurrency model as you go. In this chapter you'll learn how to use Schedulers and apply them in common scenarios. We'll focus on testing, where Schedulers are especially useful, and you'll learn how to make your own Schedulers.

Using Schedulers

A Scheduler is a mechanism to "schedule" an action to happen in the future. Each operator in RxJS uses one Scheduler internally, selected to provide the best performance in the most likely scenario.

Let's see how we can change the Scheduler in operators and the consequences of doing so. First let's create an array with 1,000 integers in it:

```
var arr = [];
for (var i=0; i<1000; i++) {
  arr.push(i);
}
```

Then, we create an Observable from arr and force it to emit all the notifications by subscribing to it. In the code we also measure the amount of time it takes to emit all the notifications:

```
var timeStart = Date.now();
Rx.Observable.from(arr).subscribe(
  function onNext() {},
  function onError() {},
  function onCompleted() {
    console.log('Total time: ' + (Date.now() - timeStart) + 'ms');
  });
```

```
❮ "Total time: 6ms"
```

Six milliseconds—not bad! from uses Rx.Scheduler.currentThread internally, which schedules work to run after any current work is finished. Once it starts, it processes all the notifications synchronously.

Now let's change the Scheduler to Rx.Scheduler.default.

```
var timeStart = Date.now();
Rx.Observable.from(arr, null, null, Rx.Scheduler.default).subscribe(
  function onNext() {},
  function onError() {},
  function onCompleted() {
    console.log('Total time: ' + (Date.now() - timeStart) + 'ms');
  });
```

```
❮ "Total time: 5337ms"
```

Wow, our code runs almost a thousand times slower than with the currentThread Scheduler. That's because the default Scheduler runs each notification asynchronously. We can verify this by adding a simple log statement after the subscription.

Using the currentThread Scheduler:

```
Rx.Observable.from(arr).subscribe( ... );
console.log('Hi there!');
```

```
❮ "Total time: 8ms"
"Hi there!"
```

Using the default Scheduler:

```
Rx.Observable.from(arr, null, null, Rx.Scheduler.timeout).subscribe( ... );
console.log('Hi there!');
```

```
❮ "Hi there!"
  "Total time: 5423ms"
```

Because the Observer using the default Scheduler emits its items asynchronously, our console.log statement (which is synchronous) is executed before the Observable even starts emitting any notification. Using the currentThread Scheduler, all notifications happen synchronously, so the console.log statement gets executed only when the Observable has emitted all its notifications.

So, Schedulers really can change how our Observables work. In our case here, performance really suffered from asynchronously processing a big, already-available array. But we can actually use Schedulers to improve performance. For example, we can switch the Scheduler on the fly before doing expensive operations on an Observable:

```
arr
  .groupBy(function(value) {
    return value % 2 === 0;
  })
  .map(function(value) {
➤   return value.observeOn(Rx.Scheduler.default);
  })
  .map(function(groupedObservable) {
    return expensiveOperation(groupedObservable);
  });
```

In the preceding code we group all the values in the array into two groups: even and uneven values. groupBy returns an Observable that emits an Observable for each group created. And here's the cool part: just before running an expensive operation on the items in each grouped Observable, we use observeOn to switch the Scheduler to the default one, so that the expensive operation will be executed asynchronously, not blocking the event loop.

observeOn and subscribeOn

In the previous section, we used the observeOn operator to change the Scheduler in some Observables. observeOn and subscribeOn are instance operators that return a copy of the Observable instance, but that use the Scheduler we pass as a parameter.

observeOn takes a Scheduler and returns a new Observable that uses that Scheduler. It will make every onNext call run in the new Scheduler.

subscribeOn forces the subscription and un-subscription work (not the notifications) of an Observable to run on a particular Scheduler. Like observeOn, it accepts a Scheduler as a parameter. subscribeOn is useful when, for example, we're running in the browser and doing significant work in the subscribe call but we don't want to block the UI thread with it.

Basic Rx Schedulers

Let's look a bit more in depth at the Schedulers we just used. The ones RxJS's operators use most are `immediate`, `default`, and `currentThread`.

Immediate Scheduler

The immediate Scheduler emits notifications from the Observable synchronously, so whenever an action is scheduled on the immediate Scheduler, it will be executed right away, blocking the thread. `Rx.Observable.range` is one of the operators that uses the immediate Scheduler internally:

```
console.log('Before subscription');

Rx.Observable.range(1, 5)
  .do(function(a) {
    console.log('Processing value', a);
  })
  .map(function(value) { return value * value; })
  .subscribe(function(value) { console.log('Emitted', value); });

console.log('After subscription');
```

```
❮ Before subscription
Processing value 1
Emitted 1
Processing value 2
Emitted 4
Processing value 3
Emitted 9
Processing value 4
Emitted 16
Processing value 5
Emitted 25
After subscription
```

The program output happens in the order we expect. Each `console.log` statement runs before the notification of the current item.

When to Use It

The `immediate` Scheduler is very well suited for Observables that execute predictable and not-very-expensive operations in each notification. Also, the Observable has to eventually call `onCompleted`.

Default Scheduler

The default Scheduler runs actions asynchronously. You can think of it as a rough equivalent of setTimeout with zero milliseconds delay that keeps the order in the sequence. It uses the most efficient asynchronous implementation available on the platform it runs (for example, process.nextTick in Node.js or setTimeout in the browser).

Let's take the previous example with range and make it run on the default Scheduler. For this, we'll use the observeOn operator:

```
console.log('Before subscription');
Rx.Observable.range(1, 5)
  .do(function(value) {
    console.log('Processing value', value);
  })
  .observeOn(Rx.Scheduler.default)
  .map(function(value) { return value * value; })
  .subscribe(function(value) { console.log('Emitted', value); });
console.log('After subscription');
```

```
❮ Before subscription
Processing value 1
Processing value 2
Processing value 3
Processing value 4
Processing value 5
After subscription
Emitted 1
Emitted 4
Emitted 9
Emitted 16
Emitted 25
```

There are significant differences in this output. Our synchronous console.log statement runs immediately for every value, but we make the Observable run on the default Scheduler, which yields each value asynchronously. That means our log statements in the do operator are processed before the squared values.

When to Use It

The default Scheduler never blocks the event loop, so it's ideal for operations that involve time, like asynchronous requests. It can also be used in Observables that never complete, because it doesn't block the program while waiting for new notifications (which may never happen).

Current Thread Scheduler

The currentThread Scheduler is synchronous like the immediate Scheduler, but in case we use recursive operators, it enqueues the actions to execute instead of executing them right away. A recursive operator is an operator that itself schedules another operator. A good example is repeat. The repeat operator—if given no parameters—keeps repeating the previous Observable sequence in the chain indefinitely.

You'll get in trouble if you call repeat on an operator that uses the immediate Scheduler (such as return). Let's try this by repeating the value 10 and then use take to take only the first value of the repetition. Ideally, the code would print 10 once and then exit:

```
// Be careful: the code below will freeze your environment!
Rx.Observable.return(10).repeat().take(1)
  .subscribe(function(value) {
    console.log(value);
  });
```

```
❮ Error: Too much recursion
```

This code causes an infinite loop. Upon subscription, return calls onNext(10) and then onCompleted, which makes repeat subscribe again to return. Since return is running on the immediate Scheduler, this process repeats itself, causing an infinite loop and never getting to take.

But if instead we schedule return on the currentThread Scheduler by passing it as the second parameter, we get this:

```
var scheduler = Rx.Scheduler.currentThread;
Rx.Observable.return(10, scheduler).repeat().take(1)
  .subscribe(function(value) {
    console.log(value);
  });
```

```
❮ 10
```

Now, when repeat resubscribes to return, the new onNext call will be queued because the previous onCompleted is still happening. repeat then returns a disposable object to take, which calls onCompleted and cancels the repetition by disposing repeat, and ultimately the call from subscribe returns.

As a rule of thumb, currentThread should be used to iterate on large sequences and when using recursive operators such as repeat.

When to Use It

The currentThread Scheduler is useful for operations that involve recursive operators like repeat, and in general for iterations that contain nested operators.