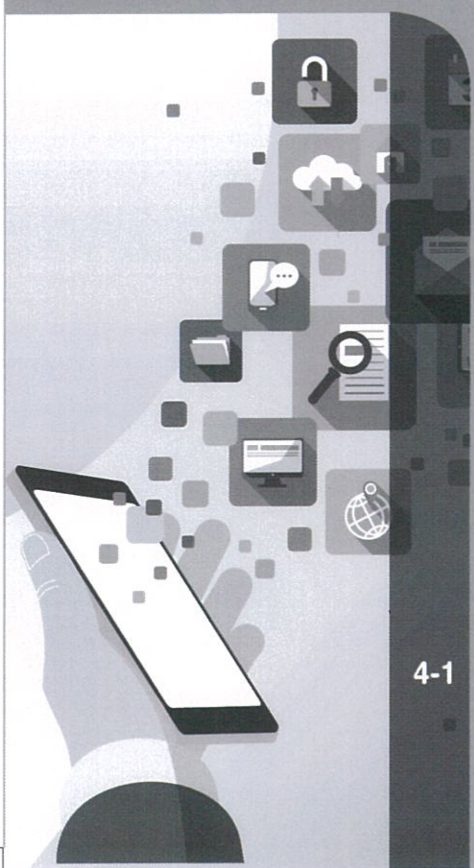
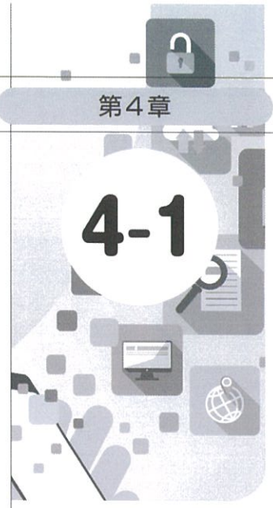
第4章

●●●●●●●●●●

リアクティブ プログラミング入門

oooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooo
oooooooooo





4-1 リアクティブプログラミング概要

○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
◆→リード、プロフィールは初校戻しにて入れるのでアタリです←◆

OOOOOOOO株式会社
OO OO *OOOO Ooooo* 000@000.jp TwitterID：@0000

リアクティブプログラミング(Reactive Programming)という言葉をご存知でしょうか？一時期話題を呼んだこともあって、お聞きになった方もいらっしゃるでしょう。しかし、リアクティブプログラミングといったときに思い浮かべるものは人によってさまざまです。

本章では、モバイルアプリ開発という観点から、プログラミング手法としてのリアクティブプログラミングについて説明します。リアクティブプログラミングの考え方の適用範囲は広いのですが、とくにモバイルアプリ開発とは相性がよく、頭の痛い多くの問題を解決する可能性を秘めています。

考え方は取っつきづらいところもありますが、とても強力な手法なので、ぜひ身に付けていただければと思います。

リアクティブプログラミングとは

リアクティブプログラミングという概念の初出はよくわかっていないようです。Gérard Berryの1989年の論文[注1]にリアクティブプログラムという単語が登場しますが、そこではリアルタイムプログラムの文脈でリアクティブプログラムについて述べられています。

> インタラクティブプログラムは、ユーザーもしくは他のプログラムと、自身のスピードで対話を行う。ユーザーの観点からはタイムシェアリングシステムはインタラクティブである。リアクティブプログラムもまた外部環境との継続的な対話を維持するが、そのスピードはプログラムではなく環境によって決定される。

ここから、リアクティブプログラムが環境に対する応答によって定義されるということはわかりますが、今いわれているリアクティブプログラミングとはちょっと結び付きません。

定義としては、英語版のWikipedia[注2]の冒頭に述べられている次の文がよりわかりやすいのではないかと思います。

> In computing, reactive programming is a programming paradigm oriented arounddata flows and the propagation of change.(拙訳：計算機においては、リアクティブプログラミングはデータフローと変更の伝播を中心としたプログラミングパラダイムである。)

リアクティブプログラミングはオブジェクト指向などと同様の1つのパラダイムととらえるのがよいでしょう。言語やライブラリ、フレームワークに縛られるものではなく、指針のようなものであり、フワッとしたものなのです。

では、「データフローと変更の伝播を中心とする」というのはどういうことでしょう。

注1) Berry, G. *Real Time Programming : Special Purpose or General Purpose Languages*. RR-1065, INRIA. 1989.
URL http://www-sop.inria.fr/members/Gerard.Berry/Papers/Berry-IFIP-89.pdf

注2) URL https://en.wikipedia.org/wiki/Reactive_programming

リアクティブプログラミングの特徴

リアクティブプログラミングは従来の**命令型プログラミング**(Imperative Programming)と対比して理解するとわかりやすいでしょう。通常のプログラミングでは、リスト1のように変数に対して実行する命令を順に書いていきます。

一方、リアクティブプログラミングでは図1のように変数間の関係を記述します。そして、変数間の関係を記述したあとに変数Aと変数Bの変更を伝播させることで変数Cの値を更新します(図2)。

このように、リアクティブプログラミングでは①**データフロー**と②**変更の伝播**の2つの段階があります。多くの場合、プログラマーが書くのは①だけで、②の変更の伝播についてはフレームワークやライブラリが面倒を見てくれます。

ここで重要なことは、先に満たすべき関係を記述しておけば、あとは変更があるたびに伝播させることで正しい結果が得られるということです。命令型プログラミングでは、その伝播に相当する操作を書き下す必要がありましたが、リアクティブプログラミングではその必要はありません。

モバイルアプリではネットワーク通信、ユーザーの入力、位置情報の変更といった多様な入力がなされます。従来の命令型プログラミングで、

◆リスト1 命令型プログラミング

```
A = 2
B = 3
C = (A + 1) * (B - 1)
```

それらを組み合わせて望む処理を行うには、多くの状態変数の導入が必要でした。一方、リアクティブプログラミングを行うことで、状態変数の数を減らし、よりバグの混入しにくいプログラムを作成できるのです。

リアクティブエクステンション(Reactive Extension、Rx)

リアクティブエクステンション(Reactive Extension)あるいはReactiveX、Rxと呼ばれるライブラリは、前節で説明したリアクティブプログラミングを実現するためのライブラリです。

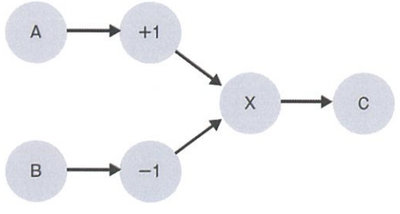
公式サイトによると、次のように説明されています[注3]。

> ReactiveXは、観測可能なシーケンス(Observable sequences)を使うことで非同期でイベントベースのプログラムを合成(compose)するライブラリです。

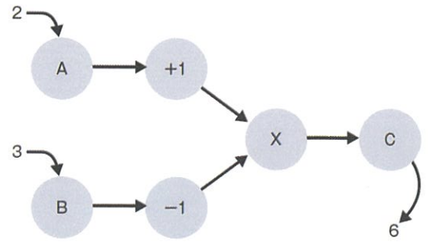
出てくる概念について、1つ1つ確認しながら、Rxの詳細について説明していきましょう。重要な用語は次の3つです。

注3) URL http://reactivex.io/intro.html

◆図1 変数間の関係



◆図2 変更を伝播



- 観測可能なシーケンス（Observable sequences）
- 非同期でイベントベースのプログラム
- 合成（compose）

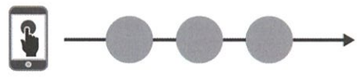
2つめの**非同期でイベントベース**なプログラムというのは、Rxが対象としているプログラムを示します。たとえば、ネットワーク通信を行い、そのレスポンスを待つのは非同期なプログラムになりますし、画面のタップに応じて何かを行うのはイベントベースなプログラムになります。

私たちが開発しているモバイルアプリは、まさにこのようなプログラムで構成されていますので、Rxがターゲットとしている領域であることがわかります。逆に非同期でなかったりイベントベースでないなら、無理にRxを使う必要はないともいえます。

次に観測可能なシーケンス（Observable sequences）ですが、Rxはオブザーバーパターンと呼ばれる、変更が発生する対象にオブザーバーを登録するデザインパターンをベースにしています。つまり、タップイベントであれば、タップが発生するボタンなどのUI部品にそのタップイベントを観測するオブザーバーを登録するのです。

さらに重要なことは、そのオブザーバーはただ一度のイベントを観測するのではなく、解除されるまでの間に発生するイベントすべてをシーケンス（sequence）として扱います。

◆図3　UIイベントの並びをシーケンスとして扱う



◆図4　非同期処理をシーケンスとして扱う

◆リスト2　Observableの購読

```
def tapObservable = ... // タップイベントを表すObservableの作成
def myOnNext = ... // イベントを観測したときに行う処理
tapObservable.subscribe(myOnNext) // 観測開始
```

図3はこれを模式的に表したものです。丸はタップイベントを、矢印は時間の向きを表し、左から右に流れています。古いイベントほど右側にあります。ユーザーが画面をタップするたびにイベントが発生し、この図では3回のタップイベントがほぼ等間隔に発生したことがわかります。

このように、イベントをシーケンスとして扱うことがRxの最も大きな特徴といってよいでしょう。これによって、後述するようなさまざまな複雑な処理を、私たちにとって馴染みがあるリスト処理として扱うことが可能になるのです。

では、一度しか発生しないネットワーク通信のような非同期処理はどのようになるでしょうか。こちらもイベント要素が1つのシーケンスとして扱うことができます（図4）。こちらでは、イベントを表す丸の後ろに縦棒が見えますね。これは、イベントのシーケンスが終了したことを表します。

このようなイベントを排出（emit）するもととなるものを、Rxでは**Observable**と呼びます。そして、そのイベントを観測する主体を**Observer**といいます。アプリ開発でRxを使う際には、何らかのイベントや非同期処理を表すObservableをまず作成し、それを消費するタイミングでObserverが観測を始める（**購読する**（subscribe）といいます）という流れになります。

リスト2にはGroovyのような疑似コードで流れを書いてみました。よく見ると、先ほどまで説明していたObserverがこのコードには明示的に登場しません。一般に、Observerを作成して**subscribe**メソッドの引数に渡すこともできますが、ラムダ式や無名関数が使える言語では、Observerの対応するメソッドの実装をそのまま**subscribe**の引数に渡すことが多いです。

リスト2では**onNext**のみを渡しましたが、ほかにも次のようなメソッドの実装を渡すことができます。

赤字

- **onNext**

Observableからイベントが排出されるたびに呼ばれるメソッドです。メソッドの引数には排出されたイベントが渡されます。

- **onError**

Observableが何らかのエラーを起こしたときに呼ばれるメソッドです。これ以降に**onNext**や**onCompleted**が呼ばれることはありません。引数にはエラーの原因が渡されます。

- **onCompleted**

最後の**onNext**が呼ばれたあとに呼ばれるメソッドです。

リスト3 赤字

◆リスト3　ネットワーク通信のObservable

```
def networkObservable = ... // ネットワーク通信を表すObservable
networkObservable.subscribe(
    { it -> show(it.message) }, // イベントを受け取ったときに呼ばれる処理
    { err -> showErr(err.message) }, // エラーが起きたときに呼ばれる処理
    { show("Completed!") }) // 完了したときに呼ばれる処理
```

◆リスト4　ネットワーク通信のObservable

```
def networkObservable = ... // ネットワーク通信を表すObservable
... // 他の処理。この時点ではネットワーク通信はまだ発生していない。
networkObservable.subscribe({ it -> show(it.message) }) // 通信結果を表示
```

◆リスト5　●●●キャプション入る●●●●●●

```
// 1秒ごとに5つのイベントが排出される
def myObservable = Observable.interval(1, TimeUnit.SECONDS).take(5)
myObservable.subscribe({ it -> log("First observer: $it") })
... // 1秒以上かかる何かの処理
myObservable.subscribe({ it -> log("Second observer: $it") })
```

これらを使った例は次のようになります。

これらの**onNext**、**onError**、**onCompleted**は、Observableの契約[注4]に従います。Observableは**onNext**を0もしくは1回以上呼び、**onError**もしくは**onCompleted**のどちらかを呼ぶかもしれません。その場合は、これ以上のメソッド呼び出しはありません。

1つ大事なことは、Observableは基本的に購読されるまで、イベントを排出し始めないということです。

初学者が陥りがちな罠として（筆者もハマりました！）、Observableを作ったらイベントの排出が始まるのではないか、と勘違いしてしまうことです。実際には購読が行われるまでイベントが排出されないことがほとんどです。

たとえば、特定のネットワーク通信を表すObservableを作っただけでは、通信は発生しません。その代わり購読されたタイミングで初めて通信が発生し、レスポンスが返ってくると、その内容がイベントとしてSubscriberに伝わります（リスト4）。

これと並んで、初学者が陥りがちな罠は、「同じObservableを購読したときに排出されるイベントのシーケンスは共通である」という勘違いです。実際には、observerが受け取るイベントシーケンスは独立です。どういうことでしょうか？

リスト5では、同じObservableインスタンスを2つのObserverが購読しています。このObservableは1秒ごとに数字を**1, 2, 3, 4, 5**と排出します。このとき、2つのObserverのログ出力はどうなるでしょうか？ 2つめのObserverの購読は遅れているので、最初に受け取るイベントは2になるでしょうか？

実はどちらも出力は**1, 2, 3, 4, 5**になります。なぜなら、Observableは、observerごとに別々のイベントシーケンスを排出するからです。もし通信を行うObservableであれば、購読するごとに通信が発生するので気をつけましょう。

最後に合成（compose）について説明しましょう。Rxでは、イベントをシーケンスとして扱うだけでなく、シーケンスを変形して新しいシーケンスを生成したり、複数のシーケンスから1つのシーケンスを生成することができます。このとき、一連のイベントをシーケンスとして扱っていたことが意味を持ちます。というのも、私たちがコレクションに対して行うのと似たような操作を使うことができるからです。

注4）リンク http://reactivex.io/documentation/contract.html

注3が2個ある？—DTP

トル

実際に、Rxを従来のリスト操作におけるIterableと比較してプル型ではなくプッシュ型という説明がされることもあります[注5]。確かに、リスト操作との共通点は多いのですが、あまりそこにこだわらず、オペレータと共通点があるという程度にとらえておくのがよいと筆者は思います。

オペレータの例を挙げましょう。リスト6を見てください。

この例ではシーケンス1, 2, 3, 4を生成し、最初のfilterメソッドを適用することで偶数のみのシーケンスを生成します。さらにmapメソッドを適用し、各要素を二乗しています。したがって、最終的に出力されるのは4, 16となります。

ご覧のように、filter、mapなどのコレクション操作で馴染みのある演算を行うことができます。このようにイベントのシーケンスを変形していく演算を、Rxでは**オペレータ**(Operator)と呼び、もう1つの特徴となっています。

このオペレータを使うメリットの1つは、一時変数や状態変数を持つ必要がなくなることです。

たとえば、タップイベントを観測してダブルタップを検出したいとしましょう。Rxを使わない場合の実装は、タップイベントを観測して、直前のタップとの時間差を計算するものです。一定時間以内だったらダブルタップと判断できます。

一方、Rxでは次のように考えます。シングルタップのイベントシーケンスを生成し、それをもとにダブルタップのイベントシーケンスを作成するのです。そのようなイベントシーケンス(Observable)ができれば、あとはそれを購読するだけです。では、どうすればダブルタップObservableが作れるでしょうか。実はRxのオペレータには、一定期間内かつ連続した2つのイベントを別のイベントに変換するものがあります(参考までに、bufferというオペレータがそれです)。したがって、そのオペレータを適用すれば、ダブルタップのイベントシーケンスを作成できます(図5)。

このとき大事なことは、状態変数(Rxを使わない例では直前のタップ時刻)を必要としないということです。もちろんRx内部ではそのような変数を持っているでしょうが、私たちはそのような細かい実装に頭を悩ませずにオペレータを適切に適用するだけで望みのObservableが作れます。

ここまで出てきた重要な用語をまとめると次のようになります。

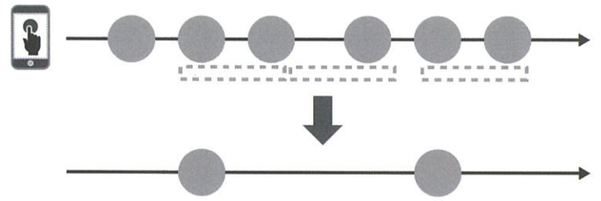
- **Observable**
  イベントを排出するもの。UI部品のタップイベントだったり、ネットワーク通信だったりします。
- **購読(subscribe)**
  Observableの観測を開始すること。
- **Observer**
  Observableを購読する主体。文脈によってはSubscriberやReactorと呼ばれることもあります。排出されるイベントや完了/エラー通知に応じて対応するObserverのメソッドが呼ばれます。ラムダ式が使える言語の場合は、subscribeメソッドの引数に渡されるラムダ式で表されることもあります。
- **Operator**
  Observableから別のObservableを作成する関数。多くの場合は関数を引数に取る高階関数。

Rxの難しさ

さて、Rxについて一般的によくいわれることは「学習コストが高く、チーム全員が習熟するのが大変」ということです。では、何が学習コストを高くしているのでしょうか。筆者はRxを学習するうえでのステップは5つあると考えています。

1. 基本的な概念の理解
2. オペレータの使いこなし(実現したいことをオペレータで実現する)
3. Subscription、Cold/Hot Observable、Subject、Backpressure、Schedulerの理解
4. Rxの内部実装の理解
5. 自分でObservableやOperatorを作成する

◆図5　シングルタップをダブルタップに変換するオペレータ



ステップ1.については、実はそれほど難しくありません。前節で述べたことを理解できれば問題はないでしょう。繰り返しになりますが、次の2点にさえ気をつければ怖いことはありません。

- Observableは作っただけでは基本的に何も起こらない。購読して初めてイベントを排出する。
- Observableは基本的に共有されない。つまり購読するたびに新しくそのobserver向けにイベントが排出される。

次に、ステップ2.についてですが、こればかりは何度も実際に使ってみるしかないでしょう。さいわい、Rxにはさまざまな言語実装があります。本書ではAndroid向けのRxJava、iOS向けのRxSwiftについてのちほど詳しく述べますが、それ以外にJavaScript実装のRxJSもあります。そして筆者のお勧めはRxJSでオペレータの使い方に慣れることです。

JavaやSwiftのようなコンパイル言語に比較して、JavaScriptなら1回の試行錯誤にかかる時間を非常に短くできます。また、UIに相当する部品もHTMLで簡単に用意できます。何か実現したいことが出てきたら、まずはRxJSで手元で確認してみるのがよいでしょう。オペレータは言語間でほぼ同一ですので、実現方法に満足がいったらそれをJavaなりSwiftなりに移植すればよいのです。

なお、筆者はRxJSをJSFiddleで試していますので、よかったら参考にしてください[注6]。BABELを使うことでES2015のラムダ式を使えるのが嬉しいところです。

ステップ3.についてですが、これは今まで説明してこなかったRxの概念です。正直にいえば、これらの概念についてはオペレータをある程度使いこなせるようになってから学ぶのがよいでしょう。多くの場合は、これらの概念を使わずに実現できますし、Subjectのように知ってしまうと乱用しがちなものもあります。

とはいえ、Rxを深く使ううえでは避けて通れない概念でもあります。次節以降で説明しますので、参考にしてください。

最後のステップ4.と5.ですが、これについてはRx利用者のすべてが習熟する必要はないと考えています。これには2つの理由があり、1つは、言語ごとに実装が異なる可能性があり汎用性がない可能性があるからです。もう1つは、筆者が今までRxについて受けた質問の大半は、オペレータの挙動がよくわからない、もしくはどうオペレータを使ったらよいかわからない、というものだったからです。ともすると、私たち開発者は自身のObservableやオペレータを作りたくなります。しかし、それらを実現するには、Rxの実装の詳細を知る必要があり、しかも既存の用意されたオペレータの組み合わせでこと足りることが大半なのです。

繰り返しになりますが、まずは用意されたオペレータとその組み合わせ方に習熟しましょう。一見すると、それらは自分のやりたいことを実現す

注5) URL https://jsfiddle.net/hydrakecat/f8tv1phn/

◆リスト6　Rxによるシーケンスの合成

```
Observable.just(1, 2, 3, 4)
          .filter({ x -> x % 2 == 0 })
          .map({ x -> x * x })
          .subscribe({ x -> print(x) }) // [4, 16]が表示される
```

注6) URL http://davesexton.com/blog/post/To-Use-Subject-Or-Not-To-Use-Subject.aspx

るのに十分でないように見えるかもしれません。しかし、Rxのオペレータは非常によく考えられており、組み合わせることで予想以上の柔軟な使い方ができるのです。

とはいえ、とくにデバッグなどではRxの実装の詳細を知っていたほうがよいこともあります。本書ではそれらについて深く述べませんが、オープンソースの利点を活かしてソースを読んでみるのもよいでしょう。

その他の重要なRxの概念

前節で説明しなかったRxの重要な概念をいくつか補足します。以降では、次の4つの概念を説明します。

1 Subscription
2 Hot/Cold Observable
3 Subject
4 Scheduler

最初に、**Subscription**という概念について説明します。RxJavaやRxSwiftでは**Observable**の**subscribe**メソッドの戻り値になり、観測している状態を表します。もし観測が不要になったらRxJavaの場合は**unsubscribe**メソッドを、RxSwiftの場合は**dispose**メソッドを呼ぶことで、Observableにイベントの排出が不要になったことを知らせます(リスト7)。

このメソッドが重要なのは、リソースの解放に深く関わるからです。モバイルアプリではメモリの制約が強く、不要なリソースを解放せずに保持してメモリリークにつながると、アプリのクラッシュを引き起こすことになります。

実際の開発では、必ず不要になったタイミングで**unsubscribe**を呼ぶようにしましょう。具体的な注意点については、次章以降のRxSwift、RxJavaのところで説明します。

次に、**Hot Observable**について説明します。今まで述べたように、Observableは一般に購読するまでイベントを排出しません。また、購読するObserverごとに独立したイベントを排出します。このようなObservableは**Cold Observable**と呼ばれます。

一方、Hot Observableは、作成されたタイミングでイベントを排出し始めます。そして、複数のObserverが同じObservableを観測した場合は、同じイベントがすべてのObserverに対して排出されます。したがって、Observableが作成されたタイミングとそれぞれのObserverが購読するタイミングによって受け取るイベントは異なることになります。

◆リスト7　Subscriptionの例

```
def subscription = myObservable.subscribe({ it -> println(it) })
subscription.unsubscribe() // これ以降はonNextが呼ばれないことが保証される
```

◆リスト8　Connectable Observableの例

```
def connectable = Observable.just(1, 2, 3).publish()
connectable.subscribe({ it -> println("observer 1: $it") })
connectable.subscribe({ it -> println("observer 2: $it") })
// ここでは、まだイベントが1つも排出されていない
connectable.connect()
// このときには、すでにすべての数字が排出されているので、
// observer 3は1つもイベントを観測しない
connectable.subscribe({ it -> println("observer 3: $it") })

// Output:
//   observer 1: 1
//   observer 2: 1
//   observer 1: 2
//   observer 2: 2
//   observer 1: 3
//   observer 2: 3
```

◆リスト9　Subjectの例

```
def subject = PublishSubject.create()
subject.subscribe({ it -> println("observer 1: $it") })
subject.onNext(1)
subject.subscribe({ it -> println("observer 2: $it") })
subject.onNext(2)

// Output:
//   observer 1: 1
//   observer 1: 2
//   observer 2: 2
```

◆リスト10　Schedulerの例

```
// デフォルトではsubscribeメソッドが呼ばれたスレッド(ここではメインスレッド)で実行される
Observable.range(1, 5)
  .subscribeOn(Schedulers.computation()) // このObservableの処理はcomputationスケジューラで開始される
  .map({ i -> i * i })
  .observeOn(MainScheduler.instance) // 以降の処理はメインスレッドで行う
  .subscribe({ x -> show(x) })
```

また言語によっては、**Connectable Observable**と呼ばれるものがあります。これは、**connect()**というメソッドを呼ばれて初めてイベントの排出を始めるHot Observableで、通常はこのConnectable Observableを利用することになるでしょう。本書で扱うRxJava、RxSwiftの両方ともサポートしており、Cold Observableに対して**publish()**というメソッドを呼ぶことによりConnectable Observableに変換することが可能です(リスト8)。

Hot Observableは便利ですが、使い方には注意が必要です。というのも、購読が実行されるタイミングによって予期しない挙動をすることがあるからです。よく必要になる場面としては、1つしかコールバックメソッドを取らないビューに対して複数の挙動が必要になる場合が挙げられます。

3つめに説明するのは**Subject**です。これは一言でいえば「Observableであり、かつobserverであるもの」です。より詳しくいえば「任意のObservableを購読し、排出されたイベントを排出するObservable」です。

例を見てみましょう。リスト9では、**PublishSubject**というSubjectの一種を使っています。これは、**onNext**が呼ばれると、即座に今購読しているobserverに対して受け取ったイベントをそのまま排出します。

したがって、最初のobserverはイベントをすべて受け取りますが、2つめのobserverは2番めのイベントしか受け取りません。

見てわかるようにSubjectはHot Observableです。したがって、**subscribe**と**onNext**のタイミングによって結果が変わることに注意すべきでしょう。Subjectはあまり使うべきでないという意見は多く耳にします[注7]。筆者も基本的には避けるべきという意見に賛成です。まずは通常のオペレータや既存のライブラリで解決できないか検討し、どうしても難しそうなら使うようにするのがよいでしょう。

最後に**Scheduler**について説明します。モバイルアプリ開発で非同期処理を行う際には、UIインタラクションを阻害しないように、メインスレッド(UIスレッド)ではなく別スレッドで行う必要があります。Rxのオペレータのいくつかは**Scheduler**を引数に取り、適切な**Scheduler**を指定することで、操作を別スレッドで行うことができます。たとえば、リスト10の例ではObservable内の処理および**map**オペレータの操作を別スレッドで行い、最後にUIで表示するときにメインスレッドを使用しています。

こうすることで、時間のかかる非同期処理は別スレッドで行い、UI操作を行う処理はメインスレッドで行う、といったことが簡単に切り替えられます。

Rxの4つの概念について、簡単にですが解説しました。これらの概念は文章を読んだだけでは理解できないものも多く、実際に使ってみることが理解への近道です。次章以降のRxSwiftおよびRxJavaの解説では、どういうときにこれらの概念を利用できるかを詳しく説明しますので、ぜひそちらも参考にしてください。

注7) URL http://davesexton.com/blog/post/To-Use-Subject-Or-Not-To-Use-Subject.aspx