# Re:VIEW テンプレート

**TechBooster 編　著**

**2017-10-22 版　　TechBooster 発行**

# はじめに

この本を手にとって頂き、ありがとうございます。本書では、「比較して学ぶ」をテーマにして CallBack、Delegate、KVO パターン等と比較しながら、RxSwift の基礎知識、アプリのパーツとしてどう書くかについて解説していきます。

RxSwift は 2016 年頃に一気に普及し、2018 年現在ではいわゆる「イケてる」アプリのほとんどは RxSwift（もしくは ReactiveSwift）を採用しています。

ほとんどが RxSwift を採用しているということは、もちろん RxSwift について知っていなければ機能を開発していくことができません。しかし、その概念を習得するのはなかなかコストが高く、iOS アプリエンジニアになってから日の浅い人にとっては、とっっっても高い壁になっているのではないかと感じます。

正直にいうと、筆者も RxSwift を触り始めてからまだ日が浅く、習得する時はとっっっても苦労しました。

RxSwift について学ぶ時、Google 検索で調べて技術ブログや Qiita の記事を漁ったり、公式ドキュメントを見たりしていました。学んでいく上で、技術ブログの記事や Qiita の記事はとても勉強になりましたし、今でも時折参照します。

当時から「RxSwift を日本語で解説してて体系的に学べる本、無いかなーあったら楽だなー」とずっと思っていて、かなり試行錯誤しながらコードを書いていました。

本書は、そんな過去の自分が最強に欲しかった本になるように書きました。

この本を読んで、RxSwift の概念がようやくわかった！ RxSwift の理解がもっと深まった！ となってくれたら嬉しいです。

## 対象読者

本書は次の読者を対象として作成しています。

- プログラミング歴 1 年以上（種類問わず）
- Swift による iOS アプリの開発経験が少しだけある（3 ヶ月〜1 年未満）
- RxSwift ライブラリを使った開発をしたことがまったくない・ほんの少しだけある

## 必須知識

- Swift の基本的な言語仕様
  - if, for, switch, enum, class, struct
  - map や filter などの高階関数の扱い
- Xcode の基本的な操作
- よく使われる UIKit の仕様
  - UILabel UITextView UITableView UICollectionView

## 推奨知識

- 設計パターン
  - MVVM アーキテクチャ
- デザインパターン
  - delegate パターン
  - KVO パターン
  - Observer パターン

## 想定環境

- OSX High Sierra
- Xcode 9.4
- Swift 4.1
- cocoapods 1.5.3

## お問い合わせ先

- Twitter
  - @k0uhashi

## 免責事項

本書は有志によって作成されているもので、米 Apple 社とは一切関係がありません。

# 目次

# 第 1 章

# iOS アプリ開発と Swift

iOS のアプリ開発において、いまではほぼ Swift 一択の状況ではないでしょうか？Swift が登場したころから Storyboard の機能も充実し、UI と処理の分けてさらに書きやすくなりました。Objective-C を使っていたころよりもアプリ開発が楽になり、iOS アプリ開発初心者でもかなりとっつきやすくなったのがわかります。

ですが、かなりとっつきやすくなったと言ってもまだ問題はいくつかあります。たとえば、「非同期処理が実装しにくい、読みにくい」「通信処理の成功・失敗の制御」「Delegate や addTarget, IBAction 等、UI と処理が離れている」などあります。これを解決する 1 つの方法としてあるのが、RxSwift（リアクティブプログラミング）の導入です。

では具体的にどう解決できるのか簡単なサンプルを例に出しながら解説します。

# 第 2 章

# RxSwift 入門

## 2.1 覚えておきたい用語と１行概要

- Reactive Extensions
  - GoF のデザインパターンの１つ、‘オブザーバパターン‘ を表したインターフェース
- RxSwift
  - ReactiveExtensions を Swift で扱えるように拡張されたライブラリ
- RxCocoa
  - UIKit で Rx を使えるようにさまざまな UI クラスを extension 定義しているライブラリで、RxSwift とにこいちでよく導入されます。
- オブザーバパターン
  - プログラム内のオブジェクトのイベント（事象）を他のオブジェクトへ通知する処理で使われるデザインパターンの一種

## 2.2 RxSwift って何？

RxSwift とは「ReactiveExtensions」を Swift で扱えるように拡張されたライブラリのことを指します。

github 上でオープンソースライブラリと公開されていてさまざまな人が日々コントリビュートしています。

Reactive Extensions については後述しますが非同期操作とイベント/データストリーム（時系列処理）の実装を容易にできるライブラリのことを指します。

## 2.3 Reactive Extensions って何？

### 2.3.1 思想

Reactive Extensions とは、「Reactive Programming」を実現するための「デザイン」とその実装ができる「ライブラリ」のことを指します。名前のとおり、Reactive Programming をするために既存のプラットフォームの機能を拡張します。

### 2.3.2 歴史

元々は Microsoft が研究して開発した.NET 用のライブラリで、２００９年に「Reactive Extensions」という名前で公開しました。現在はオープンソース化され「ReactiveX」という名前に変更されています。

この「ReactiveExtensions」の考え方がとても有用だったため JavaScript や Java、Swift など、垣根を越えてさまざまな言語に移植されていて、その中の１つが本書で紹介する「RxSwift」です。

本書では RxSwift と関連するライブラリ群についてのみ解説しますが世の中には ‘RxJava‘, ‘RxJS‘, ‘RxScala‘ などさまざまなライブラリがあります。

どのライブラリも概念はおおまかな考え方は一緒です。概念だけでも１度覚えておくと他の言語でもすぐに扱えるようになるためこの機会にぜひ覚えてみましょう！

## 2.4 RxSwift の特徴

RxSwift の特徴として、「値の変化が検知しやすい」「非同期処理を簡潔に書ける」等が挙げられます。

これは主に UI の検知（タップや文字入力の検知）や通信処理等で使われ、RxSwift を用いると delegate や callback を用いたコードよりもスッキリと見やすいコードを書けるようになります。

その他のメリットとしては次のものが挙げられます。

- 時間経過に関する処理をシンプルに書ける
- ViewController の呼び出し側で処理が書ける
- コード全体が一貫する
- まとまった流れが見やすい
- 差分がわかりやすい
- スレッドを変えやすい

また、デメリットとして主に「学習コストが高い」「デバッグしにくい」が挙げられ

ます。

プロジェクトメンバーが全員 RxSwift を書けるわけではないのにもかかわらず、とりあえず RxSwift を使えば開発速度が早くなるんでしょ？ という考え方で安易に導入すると逆に開発速度が落ちる可能性があります。

その他のデメリットとしては次のものが挙げられます。

- 一度 error が発生すると止まってしまう
  - UI とバインドするような時は止まってしまうと困るので、error が流れないものを使う
- 簡単な処理で使うと長くなりがち

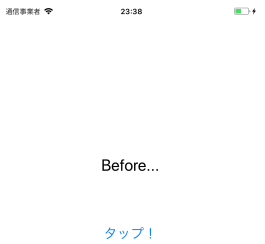
プロジェクトによって RxSwift の有用性が変わるので、そのプロジェクトの特性と RxSwift のメリット・デメリットを照らし合わせた上で検討しましょう。

## 2.5 RxSwift は何が解決できる？

RxSwift では本当に色々なことができますが、１番わかりやすくて簡単なのは「Delegate や IBAction だと動作するところと処理が離れている」の解決だと思います。

実際にコードを書いて見てみましょう。

UIButton と UILabel が画面に配置されていて、ボタンをタップすると文字列が変更されるという仕様のアプリを題材として作ります。




▲図 2.1　イメージ１


▲図 2.2　イメージ 2

まずは従来の IBAction を使った方法で作ってみましょう。

▼リスト 2.1　IBAction を用いたコード

```
1: class SimpleTapViewController: UIViewController {
2:
3:     @IBOutlet weak var messageLabel: UILabel!
4:
5:     @IBAction func buttonTap(_ sender: Any) {
6:         messageLabel.text = "Changed!!"
7:     }
8: }
```

通常の書き方だと、１つのボタンに対して１つの関数を定義します。

この場合だと UI と処理が 1 対 1 で非常に強い結合度になりますね。

仕様が非常にシンプルなため、コードもシンプルに書けてはいますが、ボタンを１つ増やすたびに対応する関数が１つずつ増えていき、コード量が次第に大きくなってしまいます。

次に、RxSwift を用いて書いてみます。

▼リスト 2.2　RxSwift を用いたコード

```
1: import RxSwift
2: import RxCocoa
3:
4: class SimpleTapViewController: UIViewController {
5:
```

```
 6:     @IBOutlet weak var tapButton: UIButton!
 7:     @IBOutlet weak var messageLabel: UILabel!
 8:
 9:     private let disposeBag = DisposeBag()
10:
11:     override func viewDidLoad() {
12:         super.viewDidLoad()
13:         tapButton.rx.tap
14:             .subscribe(onNext: { [weak self] in
15:                 self?.messageLabel.text = "Changed!!"
16:             })
17:             .disposed(by: disposeBag)
18:     }
19: }
```

まったく同じ処理を RxSwift で書きました。

tapButton のタップイベントを購読し、イベントが発生したら UILabel のテキストを変更しています。

コードを見比べてみると、１つのボタンと１つの関数が強く結合していたのが、１つのボタンと１つのプロパティの結合で済むようになっていて、UI と処理の制約を少し緩くできました。

シンプルな処理なのでコード量は RxSwift を用いた場合のほうが長いですが、この先ボタンを増やすことを考えると、１つ増やすたびに対応するプロパティが１行増えるだけなので、コードがとてもシンプルになります。

また、画面上の UI を変更してもソースコードへの影響は少なくなるので変更が楽になります。

addTarget を利用する場合のコードも見てみましょう

UILabel, UITextField を画面に２つずつ配置し、入力したテキストをバリデーションして「あと N 文字」と UILabel に反映するよくある仕組みのアプリを作ってみます


▲図 2.3　画面のイメージ

▼リスト 2.3　addTarget を用いたコード

```
 1: class ExampleViewController: UIViewController {
 2:
 3:     @IBOutlet weak var nameField: UITextField!
 4:     @IBOutlet weak var nameLabel: UILabel!
 5:
 6:     @IBOutlet weak var addressField: UITextField!
 7:     @IBOutlet weak var addressLabel: UILabel!
 8:
 9:     let maxNameFieldSize = 10
10:     let maxAddressFieldSize = 50
11:
12:     let limitText: (Int) -> String = {
13:         return "あと\($0) 文字"
14:     }
15:
16:     override func viewDidLoad() {
17:         super.viewDidLoad()
18:         nameField.addTarget(self, action: #selector(nameFieldEditingChanged(sender:)), fo
19:         addressField.addTarget(self, action: #selector(addressFieldEditingChanged(sender
20:     }
21:
22:     @objc func nameFieldEditingChanged(sender: UITextField) {
23:         guard let changedText = sender.text else { return }
24:         let limitCount = maxNameFieldSize - changedText.count
25:         nameLabel.text = limitText(limitCount)
26:     }
27:
28:     @objc func addressFieldEditingChanged(sender: UITextField) {
29:         guard let changedText = sender.text else { return }
```

```
30:         let limitCount = maxAddressFieldSize - changedText.count
31:         addressLabel.text = limitText(limitCount)
32:     }
33: }
```

UI と処理のコードが離れているので、パッとじゃ処理のイメージがしにくいですね。

対象の View がもっと増えるとどの関数がどの UI の処理なのかわかりにくくなってしまいます。

次に RxSwift を用いて書いてみます

▼リスト 2.4　RxSwift version

```
 1: import RxSwift
 2: import RxCocoa
 3:
 4: class RxExampleViewController: UIViewController {
 5:
 6:     // フィールド宣言は全く同じなので省略
 7:
 8:     private let disposeBag = DisposeBag()
 9:
10:     override func viewDidLoad() {
11:         super.viewDidLoad()
12:
13:         nameField.rx.text
14:             .map { [weak self] text -> String? in
15:                 guard let text = text else { return nil }
16:                 guard let maxNameFieldSize = self?.maxNameFieldSize else { return nil }
17:                 let limitCount = maxNameFieldSize - text.count
18:                 return self?.limitText(limitCount)
19:             }
20:             .filterNil() // import RxOptional が必要
21:             .observeOn(MainScheduler.instance)
22:             .bind(to: nameLabel.rx.text)
23:             .disposed(by: disposeBag)
24:
25:         addressField.rx.text
26:             .map { [weak self] text -> String? in
27:                 guard let text = text else { return nil }
28:                 guard let maxAddressFieldSize = self?.maxAddressFieldSize else { return
29:                 let limitCount = maxAddressFieldSize - text.count
30:                 return self?.limitText(limitCount)
31:             }
32:             .filterNil() // import RxOptional が必要
33:             .observeOn(MainScheduler.instance)
34:             .bind(to: addressLabel.rx.text)
35:             .disposed(by: disposeBag)
36:     }
37: }
```

さきほどの addTarget のパターンとまったく同じ動作をします。

全ての処理が viewDidLoad() 上で書けるようになり、UI と処理がバラバラになら

ないのですごく見やすいですね。

慣れていない方はまだ少し読みにくいかもしれませんが、Rx の書き方に慣れるとすごく読みやすくなります。

## 2.6 導入事例

- LINE（プロダクト不明）
  - 出典元：iOSDC のノベルティ
- NIKKEI 日経電子版
  - 出典元：iOSDC のノベルティ

# 第 3 章

# RxSwift の導入

## 3.1　導入要件

- RxSwift リポジトリより引用（２０１８年８月３１日現在）
- Xcode 9.0
- Swift 4
- Swift 3.x （rxswift-3.0 ブランチを指定)
- Swift 2.3 （rxswift-2.0 ブランチを指定)

## 3.2　導入方法

RxSwift の導入方法は CocoaPods や Carthage、SwiftPackageManager 等いくつかありますが、ここでは１番簡単でよく使われる（著者の観測範囲）CocoaPods での導入方法を紹介します。

CocoaPods とは、iOS/Mac 向けのアプリを開発する際のライブラリ管理をしてくれるツールのことで、これを使うと外部ライブラリが簡単に導入できます

CocoaPods を導入するには Ruby が端末にインストールされてる必要があります。(Mac ではデフォルトで入っているのであまり気にしなくてもよいですが)

次のコマンドで CocoaPods を導入できます

```
gem install cocoapods
gem install -v 1.5.3 cocoapods = バージョンを本書と同じにしたい場合はコッチ
```

これで CocoaPods を端末に導入することができました。

次に、CocoaPods を用いて、プロジェクトに外部ライブラリを導入してみます。

大まかな流れは次のとおりです。

1. Podfile というファイルを作成
2. Podfile に導入したいライブラリを定義
3. ターミナルで pod install と入力

では、実際にやってみましょう。

```
# プロジェクトのルートディレクトリで実行
vi Podfile
```

▼リスト 3.1　Podfile

```
1: # Podfile
2: use_frameworks!
3:
4: target 'YOUR_TARGET_NAME' do
5:     pod 'RxSwift',    '~> 4.0'
6:     pod 'RxCocoa',    '~> 4.0'
7: end
```

‘YOUR_TARGET_NAME‘ は各自のプロジェクト名に置き換えてください

```
# プロジェクトのルートディレクトリで実行
pod install
```

‘Pod installation complete¡ というメッセージが出力されたら導入成功です！

もし導入できていなさそうな出力であれば、書き方や typo をもう一度確認してみてください。

Tips: Podfile は Ruby とまったく同じ構文で定義されています

# 第 4 章

# 基本的な書き方

ここからは、RxSwift の書き方を説明していきます。

RxSwift の書き方はさまざまあり、本書では全てを紹介できませんがよく使われるところを抜粋して紹介します。

## 4.1 メソッドチェーンのように直感的に書ける

RxSwift/RxCocoa は、メソッドチェーンのように直感的に書くことができます。

メソッドチェーンとは、その名前のとおりメソッドを実行してその結果に対してさらにメソッドを実行するような書き方を指します。jQuery を扱ってた人はなんとなく分かるかと思います。

具体的には次のように書くことができます

▼リスト 4.1　hogeButton のイベント購読

```
1: hogeButton.rx.tap
2:   .subscribe(onNext: { [weak self] in
3:     // 処理
4:   })
5:   .disposed(by: disposeBag)
```

hogeButton のタップイベントを購読し、タップされたときに subscribe メソッド引数である onNext のクロージャ内の処理を実行します。

最後にクラスが解放されたとき自動的に購読を破棄してくれるように `disposed(by:)`メソッドを使っています。

まとめると、次の流れで定義しています。

1. イベントの購読
2. イベントが流れてきたときの処理
3. クラスが破棄と同時に購読を破棄

## 4.2 Hello World

RxSwift での Hello World 的な書き方を書いてみます。

公式な HelloWorld な書き方ではありませんが、なんとなく概念は掴めるかと思います

▼リスト 4.2　subject-example

```
 1: import RxSwift
 2: import RxCocoa
 3:
 4: let helloWorldSubject = PublishSubject<String>()
 5:
 6: helloWorldSubject
 7:   .subscribe(onNext: { [weak self] value in
 8:     print("value = \(value)")
 9:   })
10:   .disposed(by: disposeBag)
11:
12: helloWorldSubject.onNext("Hello World!")
13: helloWorldSubject.onNext("Hello World!!")
14: helloWorldSubject.onNext("Hello World!!!")
```

結果は次のとおり

▼

```
value = Hello World!
value = Hello World!!
value = Hello World!!!
```

処理の流れのイメージは次のとおりです。

1. ‘helloWorldSubject‘ という Subject を定義
2. Subject を購読
3. 値が流れてきたら print 文で値を出力させる
4. 定義したクラスが破棄されたら購読も自動的に破棄させる
5. 値を流す

Subject を使った書き方はよく使われます。たとえば、ViewController/ViewModel 間のデータの受け渡しや親子 ViewController 間でのデータ受け渡しで使われます。

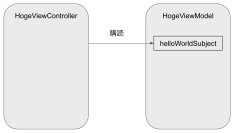
上記のコードは同じクラス内に書いていて、Subject の強みがあまり出ません。

実際に ViewController/ViewModel に分けて書いてみましょう。

▼リスト 4.3　ViewController/ViewModel に分けて書く

```
 1: class HogeViewController: UIViewController {
 2:
 3:   private let disposeBag = DisposeBag()
 4:
 5:   var viewModel: HogeViewModel!
 6:
 7:   override func viewDidLoad() {
 8:     super.viewDidLoad()
 9:
10:     viewModel = HogeViewModel()
11:
12:     viewModel.helloWorldSubject
13:       .subscribe(onNext: { [weak self] value in
14:         print("value = \(value)")
15:       })
16:       .disposed(by: disposeBag)
17:
18:     viewModel.updateItem()
19:   }
20:
21: }
22:
23: class HogeViewModel {
24:   let helloWorldSubject = PublishSubject<String>()
25:
26:   func updateItem() {
27:     helloWorldSubject.onNext("Hello World!")
28:     helloWorldSubject.onNext("Hello World!!")
29:     helloWorldSubject.onNext("Hello World!!!")
30:   }
31: }
```

出力結果自体はさきほどと同じです。
このコードを図で表してみましょう。




▲図 4.1　Subject イメージ図

ViewController が ViewModel の Subject を購読したことで、変更があった場合にイベント・値を受け取れます。

ViewModel はデータの変更を ViewController に伝える必要がなくなるので、ViewController を知らなくても良くなり、「UI とロジックの分離」ができました。テストも書きやすくなりますね。

なんとなく処理の流れは掴めたでしょうか？ ここで記載した HelloWorld のコー

ドは RxSwift の数ある機能の中ではほんの一部にしかすぎないので、やりながら覚えていきましょう！

## 4.3 よく使われるクラス・メソッドについて

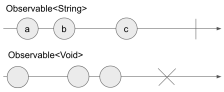
さて、ここまで本書を読み進めてきていくつか気になるワードやメソッド、クラス名が出てきたのではないでしょうか？

ここからようやくそれらのクラスとそれらを支える概念についてもう少し深く触れていきたいと思います。

### 4.3.1 Observable

Observable は翻訳すると観測可能という意味で文字どおり観測可能なものを表現するクラスで、イベントを検知するためのものです。

まずは、次の図を見てください。




▲図 4.2　Observable

RxSwift（ReactiveExtension）について少し調べた方は大体みたことあるような図ではないでしょうか？

これがまさに Observable を表しています。

Observable に流れるイベントには次の種類あります。

- onNext
  - デフォルトのイベント
  - 値を格納でき、何度でも呼ばれる
- onError
  - エラーイベント
  - １度だけ呼ばれ、その時点で終了、購読を破棄
- onCompleted
  - 完了イベント
  - １度だけ呼ばれ、その時点で終了、購読を破棄

これをコードでどう扱うかというと、次のように扱います

▼リスト 4.4　hogeSubject の購読の仕方

```
1: hogeSubject
2:   .subscribe(onNext: {
3:     print("next")
4:   }, onError: {
5:     print("error")
6:   }, onCompleted: {
7:     print("completed")
8:   })
```

コードを見てなんとなく予想できるかと思いますが、onNext イベントが流れてきたときは onNext のクロージャが実行され、onError イベントが流れてきたときは onError のクロージャーが実行されます。

また、UIKit を subscribe する場合は onError や onCompleted イベントが流れてこないので、onNext のクロージャ以外は省略できます。

Tips: Observable と Observer

さまざまな資料に目を通していると、Observable と Observer という表現が出てきますがどちらも違う意味です。イベント発生元が Observable でイベント処理が Observer です。ややこしいですね。

コードで見てみましょう。

▼リスト 4.5　Observable と Observer

```
1: hogeObservable // Observable（イベント発生元）
2:   .map { $0 * 2 } // Observable（イベント発生元）
3:   .subscribe(onNext: {
4:     // Observer(イベント処理)
5:   })
6:   .disposed(by: disposeBag)
```

コードで見てみるとわかりやすいですね。名前は似てますが違う意味だというのを頭の隅に入れておくと理解がより進むかと思います。

### 4.3.2 Dispose

ここまでコードを見てくると、なにやら subscribe したあとに必ず `disposed(by:)` メソッドが呼ばれているのが分かるかと思います。さてこれは何でしょう？

一言で説明すると、メモリリークを回避するための仕組みです。

Dispose は購読を解除（破棄）するためのもので、dispose() メソッドを呼ぶことで購読を破棄できます。

ただし、Observable セクションで記述したとおり、onError や onCompleted イベントが流れてくると購読が自動で破棄してくれるので dispose() メソッドを呼ぶ必要

はありません。

しかし、onError や onCompleted イベントが全ての場合で必ず流れてくるわけではありません。

とはいえ、購読を破棄するタイミング、難しいですよね？ その画面がしばらく呼ばれなくなった時？ インスタンスが破棄されたとき？ いちいちそんなこと考えてられないし deinit につらつらと書くのも面倒ですね

そこで活躍するのが DisposeBag という仕組みです。

DisposeBag は Disposable インスタンスを貯めておいて、自身が解放（deinit）されたときに管理している購読を全て自動で解放（unsubscribe）してくれる便利なものです。楽ですね。

コードで見てみましょう

▼リスト 4.6　Dispose のサンプルコード

```
 1: class HogeViewController {
 2:   @IBOutlet weak var hogeButton: UIButton!
 3:   @IBOutlet weak var fooButton: UIButton!
 4:   private let disposeBag = DisposeBag()
 5:
 6:   override viewDidLoad() {
 7:     super.viewDidLoad()
 8:     hogeButton.rx.tap
 9:       .subscribe(onNext: { // .. })
10:       .disposed(by: disposeBag)
11:
12:     fooButton.rx.tap
13:       .subscribe(onNext: { // .. })
14:       .disposed(by: disposeBag)
15:   }
16: }
```

上記の場合だと、HogeViewController が解放（deinit）されるときに保持している hogeButton の subscribe と fooButton の subscribe の Disposable を dispose してくれます。

とりあえず購読したら必ず `disposed(by: disposeBag)` しておけば大体間違いないです。

Tips: シングルトンインスタンス内で DisposeBag を扱うときは注意！

DisposeBag はとても便利な仕組みですが、シングルトンインスタンス内で DisposeBag を扱う時は注意が必要です。DisposeBag の仕組みはそのクラスが解放されたとき、管理してる Disposable を dispose するとさきほど記述しましたね。

つまり DisposeBag のライフサイクルは保持しているクラスのライフサイクルと同一のものになります。

しかし、シングルトンインスタンスのライフサイクルはアプリのライフサイクルと同一のため、いつまでたっても disposable されず、メモリリークになる可能性があり

ます。

回避策がまったくないわけではありませんが、ここでは詳細を省きます。シングルトンインスタンスで扱う場合には注意が必要！ ということだけ覚えておいてください。

### 4.3.3 Subject, Relay

Subject、Relay は簡単にいうと、イベントの検知に加えてイベントの発生もできるすごいやつです。

ここで少し Observable について振り返ってみましょう。Observable とは、イベントを検知するためのクラスでしたね。Subject、Relay はそれに加えてイベントを発生させることもできるクラスです。

Subject、Relay は色々ありますが、代表としてよく使われる次の 4 種類を紹介します。

- PublishSubject
- BehaviorSubject
- PublishRelay
- BehaviorRelay

それぞれの違いをざっくりとですが、次のテーブル図にまとめました。

▼表 4.1　Subject と Relay の主な種類

| 流れるイベント | 初期値 |
|---|---|
| PublishSubject | onNext, onError, onComplete 持たない |
| BehaviorSubject | onNext, onError, onComplete 持つ |
| PublishRelay | onNext 持たない |
| BehaviorRelay | onNext 持つ |

### 4.3.4 初期値について

初期値をもつ・もたないの違いは、Subject、Relay を Observer した瞬間にイベントが流れるか流れないかの違いです。やりたいことに合わせて使い分けましょう。

### 4.3.5 それぞれの使い分け

とはいえ、最初はどう使い分けるかが難しいと思います。特に Subject と Relay はどう使えばよいか困る人も多いと思います。

簡単な使い分けを紹介すると、「通信処理や DB 処理等」エラーが発生したとき

に分岐させたい場合は Subject、UI に値を Bind するようなものは Relay を使いましょう。

（RxSwift のデメリットでも触れましたが、UI に Bind している Observable で onError や onComplete が発生しまうと、購読が止まってしまう為、onNext のみが流れる Relay を使うのが適切です。）

### 4.3.6 Tips: internal（public）な Subject,Relay

Subject, Relay はすごく便利でいろいろなことができます。便利なのはよいことですが、いろいろ出来すぎてしまうと逆にコードが複雑になってしまうことがあります。

どういうことかというと、クラスの外からもイベントを発生させることができるためどのクラスがどこでイベント発生させているか段々わからなくなり、デバッグをするのがかなりしんどくなります。

その解決策として、Subject、Relay は private として定義して、外部用の Observable を用意するのが一般的に用いられています。

次のコードのように定義します。

▼リスト 4.7　BehaviorRelay のサンプル

```
1: private let items: BehaviorRelay<[Item]>(value: [])
2:
3: var itemsObservable: Observable<[Item]> {
4:   return items
5: }
```

### 4.3.7 Tips: Subject と Relay

Subject と Relay、それぞれ特徴が違うと書きましたが、コードを見てみると、実は Relay は Subject の薄いラッパーとして定義されています。

それぞれ onNext イベントは流せますが、Relay の場合は呼び出すメソッドが Subject と異なるので注意しましょう

▼リスト 4.8　Subject と Relay の値の流し方

```
1: let hogeSubject = PublishSubject<String>()
2: let hogeRelay = PublishRelay<String>()
3:
4: hogeSubject.onNext("ほげ")
5: hogeRelay.accept("ほげ")
```

呼び出すメソッドが違うからなにか特別なことしてるのかな？ と思うかもしれま

せんが、特別なことはしていません。PublishRelay のコードを見てみましょう。

▼リスト 4.9　PublishRelay の実装コード（一部省略）

```
public final class PublishRelay<Element>: ObservableType {
  public typealias E = Element

  private let _subject: PublishSubject<Element>

  // Accepts ‘event‘ and emits it to subscribers
  public func accept(_ event: Element) {
    _subject.onNext(event)
  }

  // ...
```

コードを見てみると、内部的には onNext を呼んでいるので、特別なことはしていないというのがわかります。

### 4.3.8　bind

Observable/Observer に対して bind メソッドを使うと指定した Observer にイベントストリームを接続することができます。

「bind」と聞くと双方向データバインディングを想像しますが、RxSwift では単方向データバインディングです。双方向データバインディングが不可能というわけではありませんが、筆者の観測範囲では使っている人は少ないです。

bind メソッドが独自でなにか難しいことをやっているわけではなく、振る舞いは subscribe と同じです。

実際にコードを比較してみましょう。

▼リスト 4.10　Bind を用いたコードのサンプル

```
import RxSwift
import RxCocoa

// ...

@IBOutlet weak var nameTextField: UITextView!
@IBOutlet weak var nameLabel: UILabel!
private let disposeBag = DisposeBag()

// ① bind を利用
nameTextField.rx.text
  .bind(to: nameLabel.rx.text)
  .disposed(by: disposeBag)

// ② subscribe を利用
nameTextField.rx.text
  .subscribe(onNext: { [weak self] text in
```

```
18:     nameLabel.text = text
19:   })
20:   .disposed(by: disposeBag)
```

上記のコードでは① bind を利用した場合と② subscribe を利用した場合それぞれ定義しましたが、まったく同じ動作をします。振る舞いが同じという意味が伝わったでしょうか？

### 4.3.9 Operator

ここまでコードでは、入力された値をそのまま bind するコードが多く登場しました。ですが実際のプロダクションコードではそのまま bind する場合はほぼ無く、何か加工して bind する場合が多いかと思います。

たとえば、入力されたテキストの文字数を数えて「あと N 文字」とラベルのテキストに反映する仕組みは代表的なパターンの１つだと思います。

そこで活躍するのが Operator という概念です。

Operator というは Observable のイベント値を途中で変換したり、絞り込んだりすることや、２つの Observable のイベントを合成したり結合したりすることを指します。

Operator は本当に沢山、色々なことができてそれだけで１冊の本が書けるレベルです。なので、ここではよく使われる Operator を簡単に紹介します。

- 変換
  - map
    - 通常の高階関数と同じ動き
  - flatMap
    - 通常の高階関数と同じ動き
  - reduce
    - 通常の高階関数と同じ動き
  - scan
    - reduce に似ていて途中結果もイベント発行ができる
  - debounce
    - 指定時間イベントが発生しなかったら最後に流されたイベントを流す
- 絞り込み
  - filter
    - 通常の高階関数と同じ動き
  - take
    - 指定時間の間だけイベントを通知して onCompleted する

  - skip
    - 名前のとおり、指定時間の間はイベントを無視する
  - distinct
    - 重複イベントを除外する
- 組み合わせ
  - zip
    - 複数の Observable を組み合わせる（異なる型でも可能）
  - merge
    - 複数の Observable を組み合わせる（異なる型では不可能）
  - combineLatest
    - 複数の Observable の最新値を組み合わせる（異なる型でも可能）
  - sample
    - 引数に渡した Observable のイベントが発生されたら元の Observable の最新イベントを通知
  - concat
    - 複数の Observable のイベントを順番に組み合わせる（異なる方では不可能）

ここで一覧で紹介されてもおぼえきれねーよ！ と思うかもしれませんがそのとおりです。すぐ覚えなくてもよいので、こんなことできるのか〜とフワっと覚えていただければ良いです。

また、RxSwift を書き始めたばっかりの人はどれがどんな動きをするか全然わからないとは思いますが、やっていきながら段々と覚えていきましょう！

さらにスコープを狭めて、割と簡単に使いやすくてよく使うものをサンプルコードを加えてさらにピックアップしてみました。

### map

▼リスト 4.11　Operator - map のサンプル

```
1: // hogeTextField のテキスト文字数を数えて fooTextLabel のテキストへ反映
2: hogeTextField.rx.text
3:   .map { return "あと\($0.count) 文字 }
4:   .bind(to: fooTextLabel.rx.text)
5:   .disposed(by: disposeBag)
6: }
```

### filter

▼リスト 4.12　Operator - filter サンプル

```
1: // 整数が流れる Observable から偶数のイベントのみに絞り込んで evenObservable に
流す
2: numberObservable
3:   .filter { $0 % 2 == 0 }
4:   .bind(to: evenObservable)
5:   .disposed(by: disposeBag)
```

また、かならず Observable のイベントを使う必要はありません、次のようにクラス変数やメソッド内変数を取り入れて bind することもできます

▼リスト 4.13　Operator - filter サンプル 2

```
1: // ボタンをタップしたときに nameLabel にユーザの名前を表示する
2: let user = User(name: "kOuhashi")
3:
4: showUserNameButton.rx.tap
5:   .map { [weak self] in
6:     return self?.user.name
7:   }
8:   .filterNil() // import RxOptional が必要
9:   .bind(nameLabel.rx.text)
10:  .disposed(by: disposeBag)
```

**zip**

複数の API にリクエストして同時に反映したい場合に使用します

▼リスト 4.14　Operator - zip サンプル

```
1: Observable.zip(api1Observable, api2Observable)
2:   .subscribe(onNext { (api1, api2) in
3:     // ↑タプルとして受け取ることができます
4:     //...
5:   })
6:   .disposed(by: disposeBag)
```

## 4.4　Hot な Observable と Cold な Observable

これまで Observable とそれらを支える仕組みについて記載してきましたが、Observable といっても Hot な Observable と Cold な Observable と 2 つの種類があります。

本書では Hot な Observable を主に扱いますが、Cold な Observable もあるということを頭の中に入れておきましょう。

Hot な Observable の特徴は次のとおりです。

- subscribe されなくても動作する
- 複数の箇所で subscribe しても全ての Observable で同じイベントが同時に流れる

Cold な Observable の特徴は次のとおりです。

- subscribe したときに動作する
- 複数の箇所で subscribe したとき、それぞれの Observable でそれぞれのイベントが流れる

Cold な Observable は主に非同期通信処理で使われます。

試しに subscribe 時に 1 つの要素を返す Observable を作成する関数を定義してみましょう

▼リスト 4.15　Cond な Observable

```
 1: func myJust<E>(_ element: E) -> Observable<E> {
 2:   return Observable.create { observer in
 3:     observer.on(.next(element))
 4:     observer.on(.completed)
 5:     return Disposables.create()
 6:   }
 7: }
 8:
 9: _ = myJust(100)
10:   .subscribe(onNext: { value in
11:     print(value)
12:   })
```

余談ですが、このような 1 回通知して onCompleted する Observable のことは「just」と呼ばれてます

Observable の create 関数はコードを見て分かるとおり、クロージャを使用して subscribe メソッドを簡単に実装できる便利な関数です。

subscribe メソッドと同様に observer を引数にとり、disposable を返却します。

### 4.4.1　振り返り Tips: myJust は disposed（by:）しなくてもよい

あれ、そういえば disposed 関数呼んでないな？ と思う方のために少し振り返ってみましょう。Observable の特徴として onError、onCompleted イベントは 1 度しか流れず、その時点で購読を破棄するというのがありましたね。myJust 関数内では onNext イベントが送られたあと、onCompleted イベントを送っていますね。なので `disposed(by:)` しなくてもよいということになります。

# 第 5 章

# 簡単なアプリを作ってみよう！

ここまでは RxSwift/RxCocoa の概念や基本的な使い方について紹介してきましたが、ここからは実際にアプリを作りながら説明していきます。

まずは簡単なアプリから作ってみましょう。

とはいえ、いきなり RxSwift でコードを書いても理解に時間がかかるかと思います。（自分はそうでした）

なので、１つのテーマごとに callback や KVO、delegate で書かれたコードを最初に書いて、これをどう RxSwift に置き換えるか？ という観点でアプリを作っていきます。（本書のテーマである「比較して学ぶ」というのはこのことを指しています）

では、作っていきましょう！ テーマはこちら！

## 5.1 カウンターアプリ

この節ではカウンターアプリをテーマに callback、delegate、RxSwift でどうかくかを書きます。

まずはアプリの機能要件を決めます！

### 5.1.1 機能要件

- カウントの値が見れる
- カウントアップができる
- カウントダウンができる
- リセットができる

### 5.1.2 アーキテクチャ

- MVVM

### 5.1.3　画面のイメージ

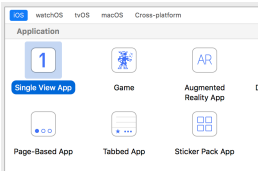

▲図 5.1　作成するアプリのイメージ

### 5.1.4　プロジェクトの作成

まずはプロジェクトを作成します。ここは特別なことをやっていないのでサクサクといきます。

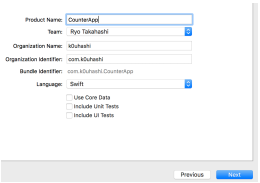

▲図 5.2　プロジェクトの作成

Xcode を新規で起動して、 Create a new Xcode project を選択します。




▲図 5.3　テンプレートの選択

テンプレートを選択します。 Single View App を選択




▲図 5.4　プロジェクトの設定

プロジェクトの設定をします。 ここは各自好きなように設定してください。
Next ボタンを押してプロジェクトの作成ができたら、一度 Xcode を終了します

### 5.1.5　環境設定

terminal.app を起動し、作成したプロジェクトのルートまで移動します

```
ryo-takahashi@~/CounterApp>
```

まず最初に RxSwift/RxCocoa ライブラリの導入を行います。プロジェクト内で ‘pod init‘ を実行します。
成功すると、ディレクトリ内に Podfile というファイルが生成されているのでこれを編集します。

```
vi Podfile
```

ファイルを開いたら、次のように編集してください。

▼リスト 5.1　Podfile の編集

```
 1: # Uncomment the next line to define a global platform for your project
 2: # platform :ios, '9.0'
 3:
 4: target 'CounterApp' do
 5:   = Comment the next line if you're not using Swift and don't want to use dynamic framew
 6:   use_frameworks!
 7:
 8:   = Pods for CounterApp
 9:   pod 'RxSwift' = ★この行を追加
10:   pod 'RxCocoa' = ★この行を追加
11:
12: end
```

編集して保存したら、導入のためインストール用コマンドを入力します。

```
pod install
```

次のような結果が出たら成功です。

```
Analyzing dependencies
Downloading dependencies
Installing RxCocoa (4.2.0)
Installing RxSwift (4.2.0)
Generating Pods project
Integrating client project

[!] Please close any current Xcode sessions and use `CounterApp.xcworkspace` fo
Sending stats
Pod installation complete! There are 2 dependencies from the Podfile and 2 tota

[!] Automatically assigning platform `ios` with version `11.4` on target `Count
```

環境設定はこれで完了です。

次回以降プロジェクトを開く時は、必ず "YOUR_PROJECT_NAME.xcworkspace" から開くようにしましょう

（*.xcworkspace から開かないと導入したライブラリが使えません）

### 5.1.6 開発を加速させる設定

★ このセクションは今後何度も使うので付箋やマーカーを引いておきましょう！

この節では、節タイトルのとおり開発を加速させる簡単な設定を行います。本書のテーマとは少しずれるので早足で進めます。

具体的には、Storyboard を廃止して ViewController + Xib を使って開発する手法に切り替えます。

### Storyboard の廃止

Storyboard は画面遷移の設定が簡単にできたり、パッと見るだけで画面がどう遷移していくかわかりやすくてよいのですが、

反面としてアプリが大きくなってくると画面遷移が複雑で逆に見辛くなったり、小さな ViewController（アラートやダイアログを出すものなど）の生成が面倒だったり、チーム人数が複数になると*.storyboard が conflict しまくるなど色々問題があるので、Storyboard を使うのをやめます。

Storyboard を廃止するために、次のことを行います

- Main.storyboard の削除
- Info.plist の設定
- AppDelegate の整理
- ViewController.xib の作成

#### ■Main.storyboard の削除

- CounterApp.xcworkspace を開く
- /CouterApp/Main.storyboard を Delete
  - Move to Trash を選択

**■Info.plist**　Info.plist にはデフォルトで Main.storyboard を使ってアプリを起動するような設定が書かれているので、それを削除します

- Info.plist を開く
- Main storyboard file base name の項目を削除する

**■AppDelegate の整理**　Main.storyboard を削除したことによって、一番最初に起動する ViewController の設定が失われ、アプリの起動が失敗するようになってしまったので、AppDelegate に一番最初に起動する ViewController を設定します。

▼リスト 5.2　AppDelegate.swift を開く

```
1: //AppDelegate.swift
2: import UIKit
3:
4: @UIApplicationMain
5: class AppDelegate: UIResponder, UIApplicationDelegate {
6:
7:     var window: UIWindow?
8:
9:     func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOp
```

```
10:         self.window = UIWindow(frame: UIScreen.main.bounds)
11:         let navigationController = UINavigationController(rootViewController: ViewControl
12:         self.window?.rootViewController = navigationController
13:         self.window?.makeKeyAndVisible()
14:         return true
15:     }
16:
17: }
```

**■ViewController.xib の作成**　Main.storyboard を削除してことによって一番最初に起動する ViewController の画面のデータがなくなってしまったので作成します。

- New File > View > Save As: ViewController.xib > Create
- ViewController.xib を開く
- Placeholders > File's Owner を選択
- Class に ViewController を指定

▲図 5.5　ViewController.xib の設定 1

Outlets の view と ViewController の View をつなげる

▲図 5.6　ViewController.xib の設定 2

これでアプリの起動ができるようになりました。Build & Run で確認してみましょう。
次のような画面が出たら成功です。

▲図 5.7　起動したアプリの画面

これで環境設定は終了です。今後画面を追加していくときは同様の手順で作成していきます。

1. ViewController.swift の作成
2. ViewController.xib の作成
3. ViewController.xib の設定
4. Class の指定
5. View の Outlet の設定

Tips: 画面遷移

ViewController.swift + ViewController.xib 構成にしたことによって、ViewController の生成が楽になりました。

また、そのおかげで画面遷移が少ない行で実装できるようになりました。次のコードで画面遷移を実装できます。

▼リスト 5.3　画面遷移の実装

```
1: let viewController = ViewController()
2: navigationController?.pushViewController(viewController, animated: true)
```

### 5.1.7　CallBack で作るカウンターアプリ

ようやくここから本題に入ります、まずは ViewController.swift を整理しましょう

- ViewController.swift を開く
- 次のように編集
  - didReceiveMemoryWarning メソッドは特に使わないので削除します。

▼リスト 5.4　ViewController の整理

```
 1: import UIKit
 2:
 3: class ViewController: UIViewController {
 4:
 5:     override func viewDidLoad() {
 6:         super.viewDidLoad()
 7:     }
 8: }
```

次に、画面を作成します。
UIButton 3つと UILabel を 1つ配置しましょう

▲図 5.8　部品の設置

UI 部品の配置が終わったら、早速 ViewController と繋げましょう。
UILabel は IBOutlet、UIButton は IBAction として繋げます

▼リスト 5.5　IBAction の作成

```
 1: import UIKit
 2:
 3: class ViewController: UIViewController {
 4:
 5:     @IBOutlet weak var countLabel: UILabel!
 6:
 7:     override func viewDidLoad() {
 8:         super.viewDidLoad()
 9:     }
10:
11:     @IBAction func countUp(_ sender: Any) {
12:     }
13:
14:     @IBAction func countDown(_ sender: Any) {
15:     }
16:
```

```
17:     @IBAction func countReset(_ sender: Any) {
18:     }
19: }
```

次に、ViewModel を作ります。ViewModel には次の役割をもたせます

- カウントデータの保持
- カウントアップ、カウントダウン、カウントリセットの処理

▼リスト 5.6　ViewModel の作成

```
 1: class ViewModel {
 2:     private(set) var count = 0
 3:
 4:     func incrementCount(callback: (Int) -> ()) {
 5:         count += 1
 6:         callback(count)
 7:     }
 8:
 9:     func decrementCount(callback: (Int) -> ()) {
10:         count -= 1
11:         callback(count)
12:     }
13:
14:     func resetCount(callback: (Int) -> ()) {
15:         count = 0
16:         callback(count)
17:     }
18: }
```

ViewModel を作ったので、ViewController で ViewModel を使うように修正と、IBAction の修正を行っていきます。

▼リスト 5.7　ViewController の修正

```
 1: class ViewController: UIViewController {
 2:
 3:     @IBOutlet weak var countLabel: UILabel!
 4:
 5:     private var viewModel: ViewModel!
 6:
 7:     override func viewDidLoad() {
 8:         super.viewDidLoad()
 9:         viewModel = ViewModel()
10:     }
11:
12:     @IBAction func countUp(_ sender: Any) {
13:         viewModel.incrementCount(callback: { [weak self] count in
14:             self?.updateCountLabel(count)
15:         })
16:     }
17:
```

```
18:     @IBAction func countDown(_ sender: Any) {
19:         viewModel.decrementCount(callback: { [weak self] count in
20:             self?.updateCountLabel(count)
21:         })
22:     }
23:
24:     @IBAction func countReset(_ sender: Any) {
25:         viewModel.resetCount(callback: { [weak self] count in
26:             self?.updateCountLabel(count)
27:         })
28:     }
29:
30:     private func updateCountLabel(_ count: Int) {
31:         countLabel.text = String(count)
32:     }
33: }
```

これで、機能要件を満たすことができました。
実際に Build & Run して確認してみましょう。
callback で書く場合のメリット・デメリットをまとめてみます。

- メリット
  - 記述が簡単
- デメリット
  - ボタンを増やすたびにボタンを押下時の処理メソッドが増えていく
    - ラベルの場合も同様
    - 画面が大きくなっていくにつれてメソッドが多くなり、コードが読みづらくなってくる
  - ViewController と ViewModel に分けたものの、完全に UI と処理の切り分けができているわけではない

### 5.1.8 Delegate で作るカウンターアプリ

次に、delegate を使って実装してみましょう。
delegate を使う場合、設計は MVP パターンのほうが向いてるので、MVP パターンに沿って実装していきます。
まずは Delegate を作ります

▼リスト 5.8　Delegate の作成

```
1: protocol CounterDelegate {
2:     func updateCount(count: Int)
3: }
```

次に、Presenter を作ります

▼リスト 5.9　Presenter の作成

```
 1: class CounterPresenter {
 2:     private var count = 0 {
 3:         didSet {
 4:             delegate?.updateCount(count: count)
 5:         }
 6:     }
 7:
 8:     private var delegate: CounterDelegate?
 9:
10:     func attachView(_ delegate: CounterDelegate) {
11:         self.delegate = delegate
12:     }
13:
14:     func detachView() {
15:         self.delegate = nil
16:     }
17:
18:     func incrementCount() {
19:         count += 1
20:     }
21:
22:     func decrementCount() {
23:         count -= 1
24:     }
25:
26:     func resetCount() {
27:         count = 0
28:     }
29: }
```

最後に、ViewController をさきほど作成した Presenter を使うように修正するのと、Delegate を extension するように修正しましょう

▼リスト 5.10　ViewController の修正

```
 1: class ViewController: UIViewController {
 2:
 3:     @IBOutlet weak var countLabel: UILabel!
 4:
 5:     private let presenter = CounterPresenter()
 6:
 7:     override func viewDidLoad() {
 8:         super.viewDidLoad()
 9:         presenter.attachView(self)
10:     }
11:
12:     @IBAction func countUp(_ sender: Any) {
13:         presenter.incrementCount()
14:     }
```

```
15:
16:     @IBAction func countDown(_ sender: Any) {
17:         presenter.decrementCount()
18:     }
19:
20:     @IBAction func countReset(_ sender: Any) {
21:         presenter.resetCount()
22:     }
23: }
24:
25: extension ViewController: CounterDelegate {
26:     func updateCount(count: Int) {
27:         countLabel.text = String(count)
28:     }
29: }
```

Build & Run してみましょう。callback の場合と同じ動きをします。
Delegate を使った書き方のメリット・デメリットをまとめます。

- メリット
  - 処理を委譲できる
  - incrementCount(), decrementCount(), resetCount() がデータの処理に集中できる
  - callback(count) しなくてもよい
- デメリット
  - ボタンを増やすたびにメソッドが増えていく

データを処理する関数が完全に処理に集中できるようになったのはよいことですが、まだボタンとメソッドの個数が１：１になっている問題が残っていて、このままアプリが大きくなっていくにつれてメソッドが多くなり、どのボタンの処理がどのメソッドの処理なのかパッと見た感じではわからなくなってしまいます。
この問題は RxSwift/RxCocoa を使うことで解決できます。
実際に RxSwift を使って作ってみましょう。

### 5.1.9 RxSwift で作るカウンターアプリ

さきほどの Presenter と CounterProtocol はもう使わないので削除しても大丈夫です。
まずは ViewModel を作るための Protocol と Input 用の構造体を作ります

▼リスト 5.11　Protocol と Struct の作成

```
 1: // ViewModel と同じクラスファイルに定義したほうが良いかも（好みやチームの規約によ
る）
 2:
 3: // ボタンの入力シーケンス
 4: struct RxViewModelInput {
 5:     let countUpButton: Observable<Void>
 6:     let countDownButton: Observable<Void>
 7:     let countResetButton: Observable<Void>
 8: }
 9:
10: // ViewController に監視させる対象を定義
11: protocol RxViewModelOutput {
12:     var counterText: Driver<String> { get }
13: }
14:
15: // ViewModel に継承させる protocol を定義
16: protocol RxViewModelType {
17:     var outputs: RxViewModelOutput? { get }
18:     init(input: RxViewModelInput)
19: }
```

次に ViewModel を作ります。CallBack パターンでも作りましたが、紛らわしくならないように新しい名前で作り直します

▼リスト 5.12　swift

```
 1: import RxSwift
 2: import RxCocoa
 3:
 4: class RxViewModel: RxViewModelType {
 5:     var outputs: RxViewModelOutput?
 6:
 7:     private let countRelay = BehaviorRelay<Int>(value: 0)
 8:     private let initialCount = 0
 9:     private let disposeBag = DisposeBag()
10:
11:     required init(input: RxViewModelInput) {
12:         self.outputs = self
13:         resetCount()
14:
15:         input.countUpButton
16:             .subscribe(onNext: { [weak self] in
17:                 self?.incrementCount()
18:             })
19:             .disposed(by: disposeBag)
20:
21:         input.countDownButton
22:             .subscribe(onNext: { [weak self] in
23:                 self?.decrementCount()
24:             })
25:             .disposed(by: disposeBag)
26:
27:         input.countResetButton
28:             .subscribe(onNext: { [weak self] in
29:                 self?.resetCount()
30:             })
```

```
31:             .disposed(by: disposeBag)
32:
33:     }
34:
35:
36:     private func incrementCount() {
37:         let count = countRelay.value + 1
38:         countRelay.accept(count)
39:     }
40:
41:     private func decrementCount() {
42:         let count = countRelay.value - 1
43:         countRelay.accept(count)
44:     }
45:
46:     private func resetCount() {
47:         countRelay.accept(initialCount)
48:     }
49:
50: }
51:
52: extension RxViewModel: RxViewModelOutput {
53:     var counterText: SharedSequence<DriverSharingStrategy, String> {
54:         let counterText = countRelay
55:             .map {
56:                 "Rx パターン:\($0)"
57:             }
58:             .asDriver(onErrorJustReturn: "")
59:         return counterText
60:     }
61: }
```

ViewController も修正しましょう。全ての IBAction と接続を消して IBOutlet を定義して接続しましょう。

▼リスト 5.13　ViewController の修正

```
 1: import RxSwift
 2: import RxCocoa
 3:
 4: class RxViewController: UIViewController {
 5:
 6:     @IBOutlet weak var countLabel: UILabel!
 7:     @IBOutlet weak var countUpButton: UIButton!
 8:     @IBOutlet weak var countDownButton: UIButton!
 9:     @IBOutlet weak var countResetButton: UIButton!
10:
11:     private let disposeBag = DisposeBag()
12:
13:     var viewModel: RxViewModel!
14:
15:     override func viewDidLoad() {
16:         super.viewDidLoad()
17:         setupViewModel()
18:     }
19:
```

```
20:     private func setupViewModel() {
21:         let input = RxViewModelInput(countUpButton: countUpButton.rx.tap.asObservable(),
22:         viewModel = RxViewModel(input: input)
23:
24:         viewModel.outputs?.counterText
25:             .drive(countLabel.rx.text)
26:             .disposed(by: disposeBag)
27:     }
28: }
```

Build & Run で実行してみましょう。まったく同じ動作をしていたら成功です。

注意！！：ここで IBAction の接続解除・IBOutlet の接続が正しくできていない場合、起動時にクラッシュするので、要注意！

もしクラッシュしてしまう場合、ViewController.xib の IBAction・IBOutlet の設定を見直しましょう！

setupViewModel 関数として切り出して定義して viewDidLoad() 内で呼び出しています。

この書き方についてまとめてみます。

- メリット
  - ViewController
    - スッキリした
    - Input/Output だけ気にすれば良くなった
  - ViewModel
    - 処理を集中できた
    - increment, decrement, reset がデータの処理に集中できた
    - ViewController のことを意識しなくてもよい
      - 例: delegate?.updateCount(count: count) のようなデータの更新を ViewController に伝えなくてもよい
  - テストがかきやすくなった
- 悪い
  - コード量が他パターンより多い
  - 書き方に慣れるまで時間がかかる

RxSwift を使った場合の一番大きなよい点はやはり「ViewModel は ViewController のことを考えなくてもよくなる」ところです。

ViewController が ViewModel の値を監視して変更があったら UI を自動で変更するため、ViewModel 側から値が変わったよ！ と通知する必要がなくなるのです。

次に、テストが書きやすくなりました。今までは ViewController と ViewModel (Presenter) が密になっていてテストが書きづらい状況でしたが、今回は分離ができたのでとても書きやすくなりました。

やり方としては ViewModel をインスタンス化するときに Input を注入し、Output を期待したとおりになっているかのテストを書く感じになります。

ですが、この方法は慣れるまで時間がかかるかと思います。まずは UIButton.rx.tap だけ使う、PublishSubject 系だけを使う・・・など小さく始めるのもありかと思います。

個人開発のアプリであれば全リプレースにチャレンジしてみても面白いかもしれませんが、業務で使うアプリでチームメンバーのほとんどが RxSwift に慣れていない場合、キャッチアップで手一杯になって逆に開発効率が落ちることもありえるのでちゃんとチームメンバーと相談しましょう！

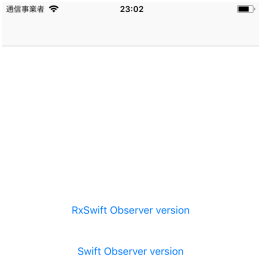
## 5.2 WKWebView を使ったアプリ

この章では KVO の実装パターンを RxSwift に置き換える方法について学びます。

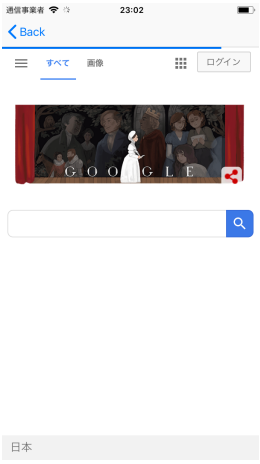
### 5.2.1 この章のストーリー

- WKWebView+KVO を使った WebView アプリを作成
- WKWebView+RxSwift に書き換える

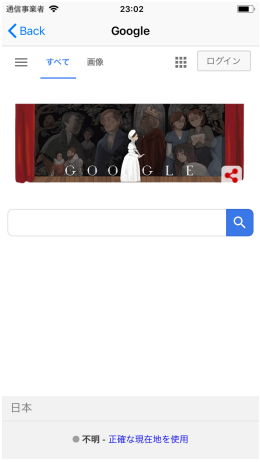
### 5.2.2 イメージ




▲図 5.9 アプリのイメージ１




▲図 5.10　アプリのイメージ 2




▲図 5.11　アプリのイメージ 3

WebView と ProgressView を配置して、Web ページの読み込みに合わせてゲージ、インジケータ、Navigation タイトルを変更する機能を作ります。

サクっといきましょう。KVO で実装した場合、次のコードになります。

▼リスト 5.14　KVO で実装する

```
 1: import UIKit
 2: import WebKit
 3:
 4: class WKWebViewController: UIViewController {
 5:   @IBOutlet weak var webView: WKWebView!
 6:   @IBOutlet weak var progressView: UIProgressView!
 7:
 8:   override func viewDidLoad() {
 9:     super.viewDidLoad()
10:     setupWebView()
11:   }
12:
13:   private func setupWebView() {
14:     // webView.isLoading の値の変化を監視
15:     webView.addObserver(self, forKeyPath: "loading", options: .new, context: nil)
16:     // webView.estimatedProgress の値の変化を監視
17:     webView.addObserver(self, forKeyPath: "estimatedProgress", options: .new, context: ni
18:
19:     let url = URL(string: "https://www.google.com/")
20:     let urlRequest = URLRequest(url: url!)
21:     webView.load(urlRequest)
22:     progressView.setProgress(0.1, animated: true)
23:   }
24:
25:   deinit {
26:     // 監視を解除
27:     webView?.removeObserver(self, forKeyPath: "loading")
28:     webView?.removeObserver(self, forKeyPath: "estimatedProgress")
29:   }
30:
31:   override func observeValue(forKeyPath keyPath: String?, of object: Any?, change: [NSKey
32:     if keyPath == "loading" {
33:       UIApplication.shared.isNetworkActivityIndicatorVisible = webView.isLoading
34:       if !webView.isLoading {
35:         // ロード完了時に ProgressView の進捗を 0.0(非表示) にする
36:         progressView.setProgress(0.0, animated: false)
37:         // ロード完了時に NavigationTitle に読み込んだページのタイトルをセット
38:         navigationItem.title = webView.title
39:       }
40:     }
41:     if keyPath == "estimatedProgress" {
42:       // ProgressView の進捗状態を更新
43:       progressView.setProgress(Float(webView.estimatedProgress), animated: true)
44:     }
45:   }
46: }
```

KVO（Key-Value Observing:キー値監視）とは、特定のオブジェクトのプロパティ値の変化を監視する仕組みです。Objective − C のメカニズムを使っていて、NSValue クラスに大きく依存しています。また、構造体（struct）は NSObject を継承できないため KVO の仕組みは使えません。

KVO を Swift で使うためにはオブジェクトをクラスで定義し、プロパティに @objc 属性と dynamic をつけます。

WKWebView には title, url, estimatedProgress など KVO に対応したプロパティ

があるので今回はそれを使っています。

では実際コード内で何をしているかというと、viewDidLoad() 時に WebView の値を監視させて、値が変更されたときに UI を更新させています。

addObserver の引数にプロパティ名を渡すとその値が変化された時に `observeValue(forKeyPath keyPath: String?, of object: Any?, change: [NSKeyValueChangeKey : Any]?, context: UnsafeMutableRawPointer?)` が呼ばれるようになります。

observeValue の keyPath には addObserver で設定した forKeyPath の値が流れてくるので、その値で条件分岐して UI を更新します。

ただ、全ての通知を observeValue で受け取って条件分岐するため、段々と observeValue メソッドが肥大化していく問題があります。

また、KVO は Objective-C のメカニズムであるため、型の安全性が考慮されているわけではありません。

KVO を使った場合の注意点として、addObserver した場合、deinit 時に removeObserver を呼ばないとアプリが強制終了する可能性があります。忘れずに removeObserver を呼びましょう。

とはいえ、removeObserver を呼ぼうと注意していても人間は忘れます、それにクラスが大きくなってくるとなおさら removeObserver を呼ぶのを忘れます。

こういった問題は RxSwift がまるまるっと解決してくれます！！ RxSwift に書き換えてみましょう。

と、その前に RxOptional という RxSwift の拡張ライブラリを導入します。理由は後述しますが、簡単にいうと Optional な値を流すストリームに対してさまざまなことができるようにするライブラリです。

Podfile にライブラリを追加しましょう

▼リスト 5.15　Podfile の修正

```
1: pod 'RxSwift'
2: pod 'RxCocoa'
3: pod 'RxOptional' = New!
```

では、導入したライブラリも使いつつ、KVO で書かれた実装を RxSwift を使うようにリプレースしていきます。

▼リスト 5.16　RxSwift でリプレース

```
import UIKit
import WebKit
import RxSwift
import RxCocoa
import RxOptional
```

```
class WKWebViewController: UIViewController {
    @IBOutlet weak var webView: WKWebView!
    @IBOutlet weak var progressView: UIProgressView!

    private let disposeBag = DisposeBag()

    override func viewDidLoad() {
        super.viewDidLoad()
        setupWebView()
    }

    private func setupWebView() {

        // プログレスバーの表示制御、ゲージ制御、アクティビティインジケータ表示制御で
使うため、一旦オブザーバを定義
        let loadingObservable = webView.rx.observe(Bool.self, "loading")
            .filterNil()
            .share()

        // プログレスバーの表示・非表示
        loadingObservable
            .map { return !$0 }
            .observeOn(MainScheduler.instance)
            .bind(to: progressView.rx.isHidden)
            .disposed(by: disposeBag)

        // iPhone の上部の時計のところのバーの（名称不明）アクティビティインジケータ
表示制御
        loadingObservable
            .bind(to: UIApplication.shared.rx.isNetworkActivityIndicatorVisible)
            .disposed(by: disposeBag)

        // NavigationController のタイトル表示
        loadingObservable
            .map { [weak self] _ in return self?.webView.title }
            .observeOn(MainScheduler.instance)
            .bind(to: navigationItem.rx.title)
            .disposed(by: disposeBag)

        // プログレスバーのゲージ制御
        webView.rx.observe(Double.self, "estimatedProgress")
            .filterNil()
            .map { return Float($0) }
            .observeOn(MainScheduler.instance)
            .bind(to: progressView.rx.progress)
            .disposed(by: disposeBag)

        let url = URL(string: "https://www.google.com/")
        let urlRequest = URLRequest(url: url!)
        webView.load(urlRequest)
    }
}
```

どうでしょうか？ ネストも浅くなり、かなり読みやすくなってのではないでしょうか。

色々説明するところはありますが、この章ではじめてでてきたメソッドについて説

明していきます。

### import RxOptional

導入した RxOptional ライブラリを swift ファイル内で使用するために宣言

### rx.observe

rx.observe は KVO を取り巻く単純なラッパーです。単純であるため、パフォーマンスが優れていますが、用途は限られています。

self. から始まるパスと、子オブジェクトだけ監視できます。

たとえば、self.view.frame を監視したい場合、第二引数に"view.frame"を指定します。

ただし、プロパティに対して強参照するため、循環参照を引き起こし最悪アプリがクラッシュする可能性があります。弱参照したい場合は、rx.observeWeakly を使いましょう。

KVO は Objective-C の仕組みで動いていると書きましたが、RxCocoa では実は構造体である CGRect、CGSize、CGPoint に対して KVO を行う仕組みが実装されています。

他の構造体を購読したいときは NSValue から手動で抽出する仕組みを実装することでできます。

RxCocoa の KVORepresentable+CoreGraphics.swift に KVORepresentable プロトコルを使って抽出する実装コードが書かれているので、独自で作りたい場合はここを参照しましょう。

### filterNil()

RxOptional で定義されている Operator

名前でなんとなくイメージできるかもしれませんが、nil の場合は値を流さず、nil じゃない場合は unwrap して値を流す Operator です。

わかりやすく、コードで比較してみましょう。次のコードはどちらもまったく同じ動作をします。

▼リスト 5.17　filterNil() の比較

```
1: // RxSwift
2: Observable<String?>
3:   .of("One",nil,"Three")
4:   .filter { $0 != nil }
5:   .map { $0! }
6:   .subscribe { print($0) }
7:
8: // RxOptional
```

```
 9: Observable<String?>
10:     .of("One", nil, "Three")
11:     .filterNil()
12:     .subscribe { print($0) }
```

### share()

文章で説明するより、まずは次のコードを見てください。

▼リスト 5.18　share() がない場合

```
 1:     let text = textField.rx.text
 2:         .map { text -> String in
 3:             print("call")
 4:             return "☆☆\(text) ☆☆"
 5:         }
 6:
 7:     text
 8:         .bind(to: label1.rx.text)
 9:         .disposed(by: disposeBag)
10:
11:     text
12:         .bind(to: label2.rx.text)
13:         .disposed(by: disposeBag)
14:
15:     text
16:         .bind(to: label3.rx.text)
17:         .disposed(by: disposeBag)
```

上記のコードは UITextField である textField へのテキスト入力を監視し、値を複数の Label へ bind、リアルタイムで入力したテキストをラベルへ反映する仕組みを実装するコードです。

ここで textField へ「123」と入力した場合、`print("call")`は何回呼ばれるか予想してみましょう。

パッと見た感じだと、3 回入力するので 3 回出力するのでは？ と思いがちですが実際は違います。実行して試してみましょう！

▼

```
call
call
call
call
call
call
call
call
call
```

call は９回呼ばれます。なるほど？

値を入力するたびに map 関数が３回呼ばれてますね。これはいけない。

今回のように値を変換したり print 出力するだけならそれほどパフォーマンスに影響はありませんが、データベースアクセスするものや、通信処理が発生するものではこの動作は好ましくありません。

では、なぜこの現象が起こるのか？

その前に、textField.rx.text が何なのかを紐解いて見ましょう。

textField.rx.text は RxCocoa で extension 定義されているプロパティで、Observable<String?>ではなく、ControlProperty<String?>として定義されています。（が、実態は Observable です）

ControlProperty は主に UI 要素のプロパティで使われていて、メインスレッドで値が購読されることを保証しています。

また、これは Cold な Observable です。

Cold な Observable の仕様として、subscribe した時点で計算リソースが割当られ、複数回 subscribe するとその都度ストリームが生成されるという仕組みがあります。

今回の場合、３回 `subscribe(bind)` したので、３個のストリームが生成されます。

するとどうなるかというと、値が変更されたときに Operator が３回実行されてしまうようになります。

このままではまずいので、どうにかして何回購読しても Operator を１回実行で済むように実装したいです。

では、どうすればよいのかというと、Hot な Observable に変換してあげるとよいです。

やりかたはいくつかあるのですが、今回は `share()` という Operator を使います。実際のコードは次のとおりです。

▼リスト 5.19　share() を使う

```
// これを
let text = textField.rx.text
    .map { text -> String in
        print("call")
        return "☆☆\(text) ☆☆"
    }
// こうしましょう
let text = textField.rx.text
    .map { text -> String in
        print("call")
        return "☆☆\(text) ☆☆"
    }
    .share() // ☆追加
```

Build & Run を実行してもう一度「１２３」とテキストに入力してみましょう。

出力結果が次のようになっていたら成功です。

▼

```
call
call
call
```

### observeOn

ストリームの実行スレッドを決めるオペレータで、このオペレータよりあとに書かれているストリームに対して適用されます

引数には「ImmediateSchedulerType」プロトコルに準拠したクラスを指定します。

### MainScheduler.instance

MainScheduler のシングルトンインスタンスを指定しています。

observeOn の引数に MainScheduler のシングルトンインスタンスを渡してあげると、その先の Operator はメインスレッドで処理してくれるようになります。

説明が長くなりましたが、本題に戻りましょう。

KVO で書いた処理を RxSwift に置き換えてみた結果、かなり読みやすくなりましたね。

特に、removeObserver を気にしなくてもよくなるのはだいぶ安全になりますね。

というよりは、RxSwift の場合は removeObserver の役割が`.disposed(by:)`に変わったイメージのほうがわかりやすいかもしれません。

`disposed(by:)` を結局呼ばないといけないのなら、そんなに変わらなくね？ と思うかもしれませんが、RxSwift では Warning が出るので removeObserver だったころより忘れる確率は低くなります。

しかし、この方法では Key 値がベタ書きになっていることと、値の型を指定してあげないといけないという問題も残っています。

もっと使いやすくするように自分で extension を定義するのもアリですが、

実はもっと便利に WKWebView を扱える「RxWebKit」という RxSwift 拡張ライブラリがあるので、それを使ってみましょう。

Podfile を編集します

▼リスト 5.20　Podfile の編集

```
1:  pod 'RxSwift'
2:  pod 'RxCocoa'
3:  pod 'RxOptional'
4:  pod 'RxWebKit'
```

ライブラリをインストールします。

```
pod install
```

さきほど書いた RxSwift パターンのコードを次のコードに書き換えてみましょう！

▼リスト 5.21　RxWebKit を用いる

```
 1: import UIKit
 2: import WebKit
 3: import RxSwift
 4: import RxCocoa
 5: import RxOptional
 6: import RxWebKit
 7:
 8: class RxWebkitViewController: UIViewController {
 9:     @IBOutlet weak var webView: WKWebView!
10:     @IBOutlet weak var progressView: UIProgressView!
11:
12:     private let disposeBag = DisposeBag()
13:
14:     override func viewDidLoad() {
15:         super.viewDidLoad()
16:         setupWebView()
17:     }
18:
19:     private func setupWebView() {
20:
21:         // プログレスバーの表示制御、ゲージ制御、アクティビティインジケータ表示制
御で使うため、一旦オブザーバを定義
22:         let loadingObservable = webView.rx.loading
23:             .share()
24:
25:         // プログレスバーの表示・非表示
26:         loadingObservable
27:             .map { return !$0 }
28:             .observeOn(MainScheduler.instance)
29:             .bind(to: progressView.rx.isHidden)
30:             .disposed(by: disposeBag)
31:
32:         // iPhone の上部の時計のところのバーの（名称不明）アクティビティインジ
ケータ表示制御
33:         loadingObservable
34:             .bind(to: UIApplication.shared.rx.isNetworkActivityIndicatorVisible)
35:             .disposed(by: disposeBag)
36:
37:         // NavigationController のタイトル表示
38:         webView.rx.title
39:             .filterNil()
40:             .observeOn(MainScheduler.instance)
41:             .bind(to: navigationItem.rx.title)
42:             .disposed(by: disposeBag)
43:
44:         // プログレスバーのゲージ制御
45:         webView.rx.estimatedProgress
46:             .map { return Float($0) }
47:             .observeOn(MainScheduler.instance)
48:             .bind(to: progressView.rx.progress)
49:             .disposed(by: disposeBag)
```

```
50:
51:
52:         let url = URL(string: "https://www.google.com/")
53:         let urlRequest = URLRequest(url: url!)
54:         webView.load(urlRequest)
55:     }
56: }
```

Build & Run で実行してみましょう。まったく同じ動作であれば成功です。

RxWebKit を使ったことでさらに可動性がよくなりました。

RxWebKit はその名前のとおり、WebKit を RxSwift で使いやすくしてくれるように拡張定義しているラッパーライブラリです。

これを使うことで、「Key のべた書き」と「値の型指定」問題がなくなりました。感謝です。

RxWebKit には他にも `canGoBack()`、`canGoForward()`に対して subscribe することもできるので、色々な用途に使えそうですね。

# 第 6 章

# Github Search サンプルアプリ

# 第 7 章

# 次のステップ

ここまでで RxSwift とその周辺についての解説は終わりです。お疲れ様でした。

ここでは、次のステップについて紹介します。この本を読んでなるほど・・・で終わってはいけません！

次は、実際に動かしているアプリで導入してみましょう！！

読者の方々によって状況はさまざまかと思いますが、まずは個人で作っているアプリに導入するのが一番早いです。

まだ作っているアプリがない場合は、好きなテーマを決めて、RxSwift を使ったアプリを作り始めましょう！

仕事で作っているアプリに導入するのでももちろんよいですが、必ずプロジェクトメンバーと相談の上、導入しましょう。（あたりまえですが）

また、気になったクラスやメソッドについて調べて技術ブログや Qiita 等にアウトプットするのもよいですし、

なんならこの書籍に続いて RxSwift 本を書いてほしいです！（読みたい）

まとめると、趣味や仕事のアプリで導入して実際に書いてインプット、

わかってきたこと・気になることをアウトプットすることを継続していくのが一番よい方法かと思っているので、やっていきましょう！！

## 7.1 コミュニティ

RxSwift を今後学んでいく上で、開発者コミュニティは非常に重要な存在です。日本国内では RxSwift 専用のコミュニティは筆者の観測内では見つけられませんでしたが、RxSwift Community（Github Project）の Slack ワークスペースがあるので、興味がある人は覗いてみてください。URL は RxSwift Repository 内の README に記載されています。

他にも、RxSwift 専用ではありませんが、Swift の国内コミュニティであればいくつかあります。

オンラインであれば、Discord 上で作られている swift-developers-japan というコミュニティや、オフラインであれば、年に 1 回行われる大規模カンファレンスの「iOSDC」や「try! Swift」などですね。

また、筆者も iOS アプリ開発に関係する勉強会を立ちあげていて（名称：iOS アプリ開発がんばるぞ！！）、主にリモートもくもく会という形でたまに開催しています。

リモートもくもく会、（というかリモート勉強会）よいソリューションだと思うんですがいまいち活発じゃないんですよね〜〜・・・インフラ勉強会はめちゃめちゃ活発なので、この動きが全体に広まって欲しい気持ちはあります。

定期的に開催しているので、見かけたら気軽に参加してください！ iOS アプリ開発に関わるものだったらなにやっても OK です！

- swift*developers-japan
  - https://medium.com/swift-column/discord-ios-20d586e373c0
- iOS アプリ開発がんばるぞ！！ の会
  - https://ios-app-yaru.connpass.com/
- iOSDC
  - https://iosdc.jp/2018/
- try! Swift
  - https://www.tryswift.co/

# 第 8 章

# さまざまな RxSwift 系ライブラリ

## 8.1 RxOptional

## 8.2 RxWebkit

## 8.3 RxDataSources

# 第 9 章

# 補足 Tips

## 9.1 参考 URL・ドキュメント・文献

- Apple Developer Documentation
  - https://developer.apple.com/documentation/
- Swift.org - Documentation
  - https://swift.org/documentation/
- ReactiveX
  - http://reactivex.io/
- RxSwift Repository
  - https://github.com/ReactiveX/RxSwift
- RxSwift Community
  - https://github.com/RxSwiftCommunity
- CocoaPods
  - https://cocoapods.org/
- Homebrew
  - https://brew.sh/index_ja
- Qiita
  - https://qiita.com/

# 著者紹介

**髙橋　凌（Takahashi Ryo）**

情報系専門学校を 2017 年に卒業、同年入社した受託開発会社を経て、2018 年に株式会社トクバイに入社。以来、破壊的イノベーションで小売業界を変革するため iOS アプリエンジニアとして従事。とにかくサービスを作ることが好きで、学生時代から色々なアプリ・Web サイトを作り、モバイルアプリでは合計 49 本のアプリをリリース。最近はサービス開発以外にも、技術同人誌の執筆や勉強会の主催を行うなど、これまでにやったことのなかった領域に手を伸ばし、視野を広げようと活動している。

## Re:VIEW テンプレート

2017 年 10 月 22 日　技術書典 3 版 v1.0.0
著　者　TechBooster 編
編　集　mhidaka
発行所　TechBooster