

比較して学ぶ RxSwift4 入門

k0uhashi 著

2018-10-08 版 iOS アプリ開発がんばるぞ！！の会 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 callback、delegate、KVO と RxSwift での実装パターンを比較しながら RxSwift について解説していきます。

解説は RxSwift をまったく触ったことのない人向けに思想・歴史から基礎知識、よく使われる文法、実際にアプリの部品としてどう書くかまでできるだけわかりやすく書きました。

RxSwift は 2016 年頃に iOS アプリ開発者界隈へ一気に普及し、2018 年現在ではいわゆる「イケてる」アプリのほとんどが RxSwift（もしくは ReactiveSwift）を採用しています。

... 主語がデカいですが、筆者の観測範囲の中ではそれくらいあたりまえのように使われています。

ほとんどが RxSwift を採用しているということは、扱いを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ためには学習コストが高く、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、高い壁になっているのではないかと感じます。

筆者もまだ日が浅かった頃は、Google 検索で出てきた技術ブログ、Qiita の解説記事や RxSwift のリポジトリ内のドキュメントなど各メディアに分散されている知見を参照しながら実装していて、

「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」と思いながら試行錯誤してコードを書いていた。

本書はそんな過去の自分と、これから RxSwift について学びたい方に向けて体系的に学べるコンテンツを提供したいという思いから生まれました。

この本を読んで、RxSwift の概念がわかった！ 理解がもっと深まった！ 完全に理解した！ RxSwift ちょっとデキル！ 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。

-
- 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
 - よく使われる高階関数の扱い
 - * map や filter など
- Xcode の基本的な操作
- よく使われる UIKit の大まかな仕様
 - UILabel、UITextField、UITableView、UICollectionView

推奨知識

- 設計パターン
 - MVP アーキテクチャ
 - 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 10
- Swift 4.2
- cocoapods 1.5.3

お問い合わせ先

- Twitter
 - <https://twitter.com/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。
また、掲載されている内容は情報の提供のみを目的にしている、本書を用いた開発・運用は必ず読者の責任と判断によって行ってください。
著者は本書の内容による開発、運用の結果によっての結果について、いかなる責任も負いません。

目次

はじめに	i
対象読者	i
必須知識	ii
推奨知識	ii
想定環境	ii
お問い合わせ先	iii
免責事項	iii
第 1 章 iOS アプリ開発と Swift	1
第 2 章 RxSwift 入門	2
2.1 覚えておきたい用語と 1 行概要	2
2.2 RxSwift って何？	2
2.3 Reactive Extensions って何？	3
2.3.1 概念	3
2.3.2 歴史	3
2.4 リアクティブプログラミングとは？	3
2.4.1 リアクティブプログラミングと Excel	4
2.5 RxSwift の特徴	5
2.6 RxSwift は何が解決できる？	6
第 3 章 RxSwift の導入	12
3.1 導入要件	12

3.2	導入方法	12
第 4 章	RxSwift の基本的な書き方	15
4.1	メソッドチェーンのように直感的に書ける	15
4.2	Hello World	16
4.3	よく使われるクラス・メソッドについて	18
4.3.1	Observable	19
4.3.2	Dispose	21
4.3.3	Subject、Relay	23
4.3.4	初期値について	23
4.3.5	それぞれの使い分け	23
4.3.6	bind	25
4.3.7	Operator	26
4.4	Hot な Observable と Cold な Observable	30
第 5 章	簡単なアプリを作ってみよう！	32
5.1	カウンターアプリを作ってみよう！	32
5.1.1	機能要件	32
5.1.2	画面のイメージ	33
5.1.3	プロジェクトの作成	33
5.1.4	環境設定	35
5.1.5	開発を加速させる設定	37
5.1.6	Callback パターンで作るカウンターアプリ	41
5.1.7	Delegate で作るカウンターアプリ	45
5.1.8	RxSwift で作るカウンターアプリ	47
5.1.9	まとめ	52
5.2	WebView アプリを作ってみよう！	52
5.2.1	この章のストーリー	52
5.2.2	イメージ	53
第 6 章	さまざまな RxSwift 系ライブラリ	65
6.1	RxDataSources	65

6.1.1	作ってみよう！	65
6.1.2	イメージ	66
6.1.3	その他セクションを追加してみよう！	73
6.2	RxKeyboard	77
6.3	RxOptional	78
第 7 章	次のステップ	80
7.1	開発中のアプリに導入	80
7.2	コミュニティへの参加	80
第 8 章	補足	82
8.1	参考 URL・ドキュメント・文献	82
著者紹介		83
余白		84

第1章

iOS アプリ開発と Swift

新規の iOS 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 の登場によって Objective-C より強い静的型付け・型推論の恩恵を借りて安全なアプリケーションを作ることができるようになったり、Storyboard の機能の充実により UI の構築が楽になり、初心者でも簡単に iOS アプリを開発できるようになりました。

しかし、簡単に開発できるようになったと言っても問題はまだいくつかあります。たとえば、「非同期処理が実装しにくい、callback 地獄で読みにくい」「通信処理の成功・失敗の制御が統一しにくい」「delegate や addTarget、IBAction などを使った実装方法が UI の定義と処理が離れていて読みづらい」などがあります。これらを解決する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ていきます。

第2章

RxSwift 入門

2.1 覚えておきたい用語と 1 行概要

- Reactive Extensions
 - 「オブザーバパターン」「イテレータパターン」「関数型プログラミング」の概念を実装したインターフェース
- 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
- RxSwift
 - Reactive Extensions の概念を Swift で扱えるようにした拡張ライブラリ
- RxCocoa
 - Reactive Extensions の概念を UIKit で扱えるようにした拡張ライブラリ 主に RxSwift と一緒に導入される

2.2 RxSwift って何？

RxSwift とは Microsoft が公開した .NET Framework 向けのライブラリである「Reactive Extensions」の概念を Swift でも扱えるようにした拡張ライブラリで、GitHub 上でオープンソースライブラリとして公開されています。

同じく Reactive Extensions の概念を取り入れた「ReactiveSwift」というライブラリも存在しますが、本書では、ReactiveSwift については触れず、RxSwift にのみ焦点を当てて解説していきます。

Reactive Extensions については後述しますが、RxSwift を導入することによって非同期操作とイベント/データストリーム（時系列処理）の実装が用意できるようになります。

2.3 Reactive Extensions って何？

2.3.1 概念

Reactive Extensions とは、「オブザーバパターン」「イテレータパターン」「関数型プログラミング」の概念を実装している.NET Framework 向けの拡張ライブラリです。これを導入することによって、リアクティブプログラミングが実現できます。

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」という名前で公開されました。現在は製品化され「ReactiveX」という名前に変更されています。

この「Reactive Extensions」の概念が有用だったため、色々な言語へと移植されています。たとえば、Java であれば RxJava、JavaScript であれば RxJS と、静的型付け・動的型付けなど関係なしに、さまざまな言語に垣根を超えて移植されています。

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、RxSwift とさきほど挙げたライブラリ群の概念の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の Rx 系ライブラリ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 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は「時間とともに変化する値」と「振る舞い」の関係を宣言的に記述するプログラミングの手法です。「ボタンをタップするとアラートを表示」のようなインタラクティブなシステムや通信処理、アニメーションのようにダイナミックに状態が変化するようなシステムに対して宣言的に動作を記述することができるため、フロントエンド側のシステムでよく使われます。

リアクティブプログラミングの説明の前に、少し命令型のプログラミングの書き方について振り返ってみます。次のコードを見てみましょう。

リスト 2.1: 擬似コード

```
1: a = 2
2: b = 3
3: c = a * b
4: a = 3
5: print(c)
```

何の前提も無く、プログラマーにこの疑似コードで出力される値を聞くと、大体は「6」と答えます。

命令型プログラミングとしてのこの結果は正しいですが、リアクティブプログラミングの観点からみた結果としては正しくありません。

冒頭の部分で少し触れましたが、リアクティブプログラミングは「値」と「振る舞い」の「関係」を宣言的に記述するプログラミングの手法です。

リアクティブプログラミングの観点では、「c にはその時点での $a * b$ の演算の結果を代入する」のではなく、「c は $a * b$ の関係をもつ」という意味で解釈されます。

つまり、c に $a * b$ の関係を定義した後は、a の値が変更されるたびに b の値がバックグラウンドで再計算されるようになります。

結果、例題の疑似コードをリアクティブプログラミングの観点からみた場合は、「9」が出力されるということになります。

2.4.1 リアクティブプログラミングと Excel

リアクティブプログラミングは、Excel を題材として説明されることがよくあります。Excel の計算式を想像してみてください。

↶↷🖨📋

100%

¥ % .0 .00 123

fx

=A1*B1

	A	B	C
1	2	3	6
2			

図 2.1 Excel

C1 セルには A1 セル値と B1 セル値を掛け算した結果を出力させています。試しに A1 セルを変更してみます。

↶↷🖨📋

100%

¥ % .0 .00 123

fx

3

	A	B	C
1	3	3	9
2			
3			

図 2.2 Excel

A1 の値の変更に合わせて、C1 が自動で再計算されました。値と振る舞いの関係を定義する、概念的にはこれがリアクティブプログラミングです。

2.5 RxSwift の特徴

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

これは UI の変更の検知（タップや文字入力）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
- 処理スレッドを変えやすい

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のほとんどが RxSwift の扱いにあまり長けていない状況の中で、とりあえずこれを導入すれ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。

- 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。

2.6 RxSwift は何が解決できる？

一番わかりやすくないのは「アプリのライフサイクルと UI の delegate や IBAction などの処理を定義している部分が離れている」の解決です。

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 2.3 画面のイメージ

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 2.2: 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。
仕様がシンプルのため、コードもシンプルに見やすく書けています。

ここから、もうひとつボタンを増やすことを想像してみましょう。

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、更に画面のパーツが増えるたびにメソッドが増えていき、読みづら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 2.3: 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。

シンプルな仕様なのでコード量は 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、比較するとこちらのほうが可動性が高くなります。

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。

第2章 RxSwift 入門

addTarget を利用する場合のコードも見てください。

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。

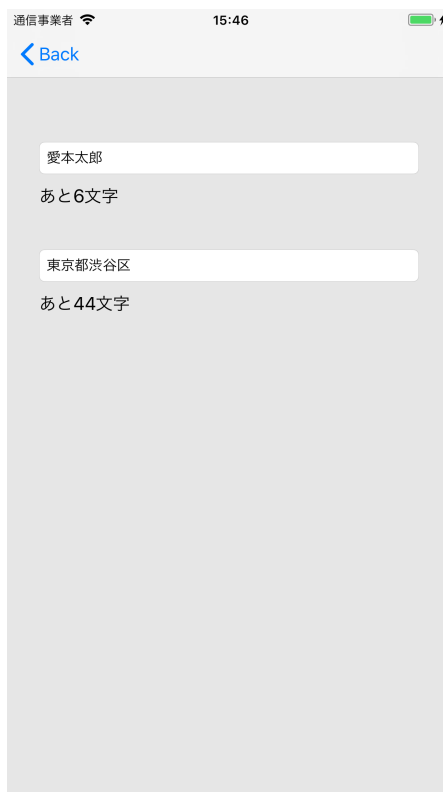


図 2.4 画面のイメージ

リスト 2.4: addTarget を用いたコード

```
class ExampleViewController: UIViewController {

    @IBOutlet weak var nameField: UITextField!
    @IBOutlet weak var nameLabel: UILabel!

    @IBOutlet weak var addressField: UITextField!
    @IBOutlet weak var addressLabel: UILabel!

    let maxNameFieldSize = 10
    let maxAddressFieldSize = 50

    let limitText: (Int) -> String = {
        return "あと\($0) 文字"
    }
}
```



```
override func viewDidLoad() {
    super.viewDidLoad()
    nameField.addTarget(self, action:
        #selector(nameFieldEditingChanged(sender:)),
        for: .editingChanged)
    addressField.addTarget(self, action:
        #selector(addressFieldEditingChanged(sender:)),
        for: .editingChanged)
}

@objc func nameFieldEditingChanged(sender: UITextField) {
    guard let changedText = sender.text else { return }
    let limitCount = maxNameFieldSize - changedText.count
    nameLabel.text = limitText(limitCount)
}

@objc func addressFieldEditingChanged(sender: UITextField) {
    guard let changedText = sender.text else { return }
    let limitCount = maxAddressFieldSize - changedText.count
    addressLabel.text = limitText(limitCount)
}
}
```

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 が更に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 2.5: RxSwift version

```
import RxSwift
import RxCocoa
import RxOptional // RxOptional という RxSwift 拡張ライブラリのインストールが必要

class RxExampleViewController: UIViewController {

    // フィールド宣言は全く同じなので省略

    private let disposeBag = DisposeBag()

    override func viewDidLoad() {
        super.viewDidLoad()
```

```
nameField.rx.text
    .map { [weak self] text -> String? in
        guard let text = text else { return nil }
        guard let maxNameFieldSize = self?.maxNameFieldSize
            else { return nil }
        let limitCount = maxNameFieldSize - text.count
        return self?.limitText(limitCount)
    }
    .filterNil() // import RxOptional が必要
    .observeOn(MainScheduler.instance)
    .bind(to: nameLabel.rx.text)
    .disposed(by: disposeBag)

addressField.rx.text
    .map { [weak self] text -> String? in
        guard let text = text else { return nil }
        guard let maxAddressFieldSize
            = self?.maxAddressFieldSize else { return nil }
        let limitCount = maxAddressFieldSize - text.count
        return self?.limitText(limitCount)
    }
    .filterNil() // import RxOptional が必要
    .observeOn(MainScheduler.instance)
    .bind(to: addressLabel.rx.text)
    .disposed(by: disposeBag)
}
```

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読みやすいですね。

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

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 4.0
- 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)

※ Xcode 9.0、Swift 4.0 と書いていますが Xcode 10、Swift 4.2 でも動作します。

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。

これを導入するには Ruby が端末にインストールされている必要があります。(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。

第3章 RxSwift の導入

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. Xcode でプロジェクトを作る
2. ターミナルでプロジェクトを作ったディレクトリへ移動
3. Podfile というファイルを作成
4. Podfile に導入したいライブラリを定義
5. ライブラリのインストール

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
pod init
vi Podfile
```

リスト 3.1: Podfile

```
1: # Podfile
2: use_frameworks!
3:
4: target 'YOUR_TARGET_NAME' do
5:     pod 'RxSwift',      '~> 4.3.1'
6:     pod 'RxCocoa',      '~> 4.3.1'
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、書き方が間違えていないか、typo していないかをもう一度確認してみてください。

Tips: Podfile

Podfile は Ruby のまったく同じ構文で定義されています。

そのため、シンタックスハイライトに Ruby を指定してあげることで、正しく表示してくれます。

Podfile では導入するライブラリのバージョンを指定することができ、本書でもバージョンを固定しています。

しかし、基本的にはバージョンを固定せずライブラリを定期的にバージョンアップさせることが推奨されています。

プロダクション環境で使う場合には、Podfile は次のように定義しましょう。

```
pod 'RxSwift'  
pod 'RxCocoa'
```

第4章

RxSwift の基本的な書き方

本章では、RxSwift の基本的な書き方や仕組みについて解説していきます。RxSwift を支える全ての仕組みを解説することは本書のテーマから逸れてしまうので、良く使われるところを抜粋して解説します。

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コードを書くことができます。メソッドチェーンとは、その名前のおりメソッドを実行し、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 を扱った人はなんとなく分かるのではないのでしょうか？

たとえば、次のように書けます。

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 メソッドの引数である 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、次のように処理を定義しています。

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. クラスが破棄されると同時に購読を破棄

4.2 Hello World

RxSwift での HelloWorld 的なものを書いてみます。

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

```
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 ViewController/ViewModel 間のデータの受け渡しや、遷移元/遷移先の ViewController 間でのデータの受け渡しでよく使われます。

また、前述したコードは同じクラス内に書いていて、Subject の強みが生かせていません。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

```
class HogeViewController: UIViewController {

    private let disposeBag = DisposeBag()

    var viewModel: HogeViewModel!

    override func viewDidLoad() {
        super.viewDidLoad()

        viewModel = HogeViewModel()

        viewModel.helloWorldSubject
            .subscribe(onNext: { [weak self] value in
                print("value = \(value)")
            })
            .disposed(by: disposeBag)

        viewModel.updateItem()
    }

}

class HogeViewModel {
    let helloWorldSubject = PublishSubject<String>()

    func updateItem() {
        helloWorldSubject.onNext("Hello World!")
        helloWorldSubject.onNext("Hello World!!")
        helloWorldSubject.onNext("Hello World!!!")
    }
}
```


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。

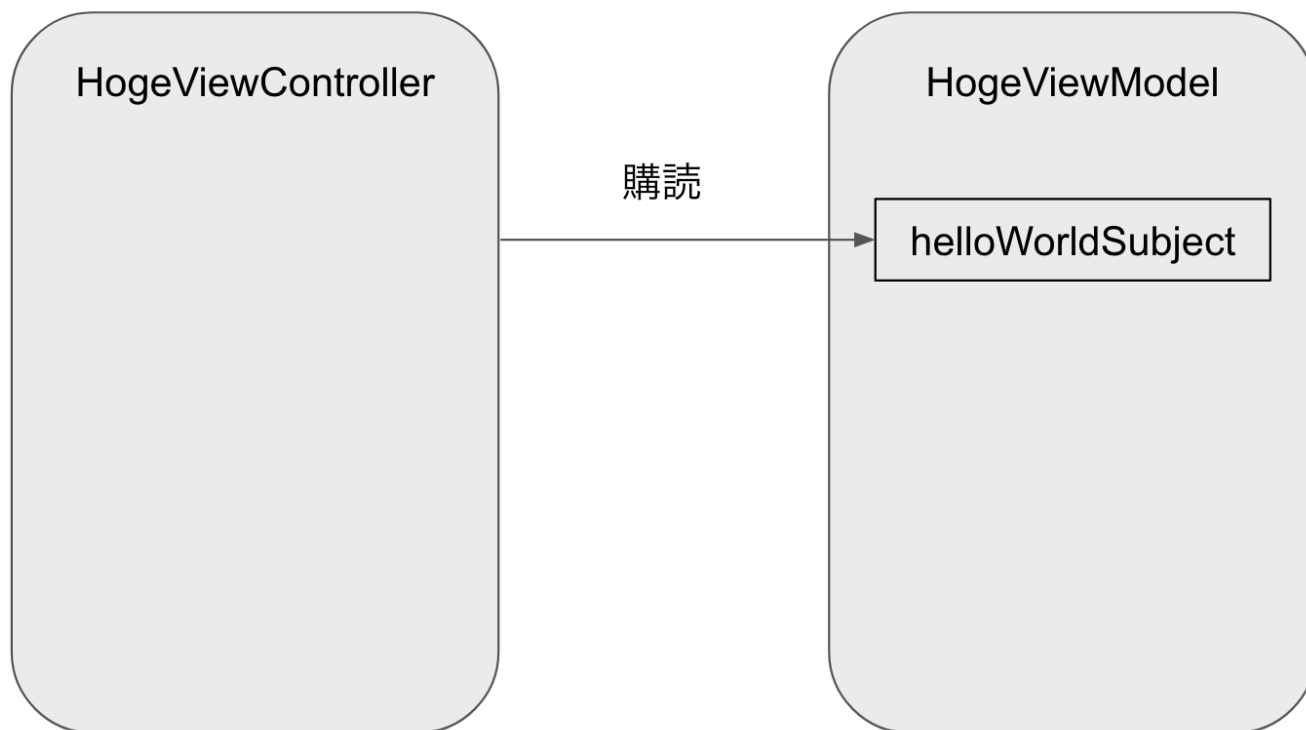


図 4.1 Subject イメージ図

ViewController が ViewModel の Subject を購読し、イベントを受け取っています。
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 のコードで使われている仕組みは RxSwift の機能の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た中で、いくつか気になるワードやクラス、メソッドが出てきたのではないのでしょうか？
ここからようやくそれらのクラスと支える概念についてもう少し深く触れていきます。

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。

Observable が通知す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を流す
 - イベント内に値を格納でき、何度でも呼びせる
- 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

コードでは、次のように扱います。

リスト 4.4: hogeObservable の購読の仕方

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

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 のクロージャが実行されます。

また、onError や onCompleted は省略することができ、それらのイベントが流れてこないことが保証されている場合は省略しましょう。

省略する場合は次のように書きます。

リスト 4.5: onNext 以外を省略する

```
1: hogeObservable
2:   .subscribe(onNext: {
3:     print("next")
4:   })
```

また、Observable は次の図を使って説明されることがよくあります。

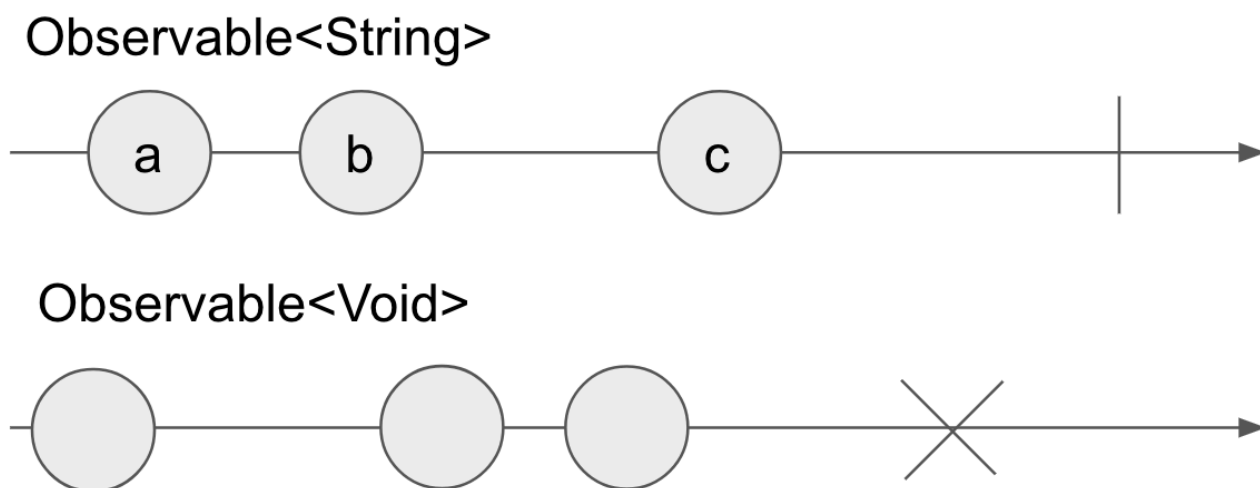


図 4.2 Observable

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

この図はマーブルダイアグラムといい、横線が時間軸で左から右に時間が流れるようなイメージです。

マーブルダイアグラムを使った解説は初級者向けではないのでここでは割愛しますが、頭の隅に置いていてください。

Tips: Observable と Observer

さまざまな資料に目を通してしていると、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。

コードで見てください。

リスト 4.6: 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)
```

コードで見るとわかりやすいです。名前は似てますが違う意味だということを頭の隅に入れておくと理解がより進みます。

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、これはイイ感じに購読を破棄して、メモリリークを回避するための仕組みです。

Observable を subscribe (bind 等) すると、Dispose クラスのインスタンスが帰ってきます。

Dispose は購読を解除（破棄）するためのクラスで、dispose() メソッドを呼ぶことで明示的に購読を破棄できます。

今回はその Dispose クラスの disposed(by:) メソッドを使っています。
クラス内に DisposeBag クラスのインスタンスを保持しておいて、引数のそのインスタンスを渡します。

引数として渡すと DisposeBag は Dispose インスタンスを貯め、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) してくれるようになります。特に購読の破棄を意識することなく、Observable を扱えるようになっているのはこの仕組みのおかげです。

コードで見てください。

リスト 4.7: Dispose のサンプルコード

```
class HogeViewController:UIViewController {
    @IBOutlet weak var hogeButton: UIButton!
    @IBOutlet weak var fooButton: UIButton!
    private let disposeBag = DisposeBag()

    override func viewDidLoad() {
        super.viewDidLoad()
        hogeButton.rx.tap
            .subscribe(onNext: {
                // ..
            })
            .disposed(by: disposeBag)

        fooButton.rx.tap
            .subscribe(onNext: {
                // ..
            })
            .disposed(by: disposeBag)
    }
}
```

リスト 4.7 では、HogeViewController が解放 (deinit) されるときに保持している hogeButton と fooButton の Disposable を dispose してくれます。

とりあえず購読したら `disposed(by: disposeBag)` しておけば大体間違いありません。

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。

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 dispose されず、最悪の場合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便利なクラスです。

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。

Subject、Relay はそれに加えて自身でイベントを発生させることもできるクラスです。

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 subscribe (bind 等) した瞬間にイベントが流れるか流れないかの違いで、やりたいことに合わせて使い分けます。

4.3.5 それぞれの使い分け

大まかな使い分けは次の通りです。

- Subject
 - 「通信処理や DB 処理等」エラーが発生したときにその内容によって処理を分岐させたい
- Relay
 - UI に値を Bind する

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、そのさきのタップイベントや入力イベントを拾えなくなってしまうので、onNext のみが流れることが保証されている Relay を使うのが適切です。

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。

便利なのは開発の幅が広がるのでよいことですが、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、internal (public) な Subject、Relay を定義してしまうと、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、アプリが肥大化していくうちにどこでイベント発生させているかわかりずらくなり、デバッグをするのが大変になってきます。

その解決策として、Subject、Relay は private として定義して、外部へ公開するための 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 4.8: BehaviorRelay のサンプル

```
1: let items = BehaviorRelay<[String]>(value: [])
2:
3: var itemsObservable: Observable<[String]> {
4:     return items.asObservable()
5: }
```

Tips: Relay は、Subject の薄いラッパー

Subject と Relay、それぞれ特徴が違うと書きましたが、Relay の実装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ることがわかります。

それぞれ onNext イベントは流せますが、Relay の場合は onNext イベントを流すメソッドが

Subject と異なるので注意しましょう

リスト 4.9: Subject と Relay の onNext イベントの流し方

```
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 4.10: PublishRelay の実装コード（一部省略）

```
1: public final class PublishRelay<Element>: ObservableType {
2:     public typealias E = Element
3:
4:     private let _subject: PublishSubject<Element>
5:
6:     // Accepts `event` and emits it to subscribers
7:     public func accept(_ event: Element) {
8:         _subject.onNext(event)
9:     }
10:
11:     // ...
```

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。

4.3.6 bind

Observable/Observer に対して bind メソッドを使うと指定したもの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 の bind は単方向データバインディングです。

bind メソッドは、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 とだいたい同じです。

実際にコードを比較してみましょう。

リスト 4.11: Bind を用いたコードのサンプル

```
1: import RxSwift
2: import RxCocoa
3:
4: // ...
5:
6: @IBOutlet weak var nameTextField: UITextView!
7: @IBOutlet weak var nameLabel: UILabel!
8: private let disposeBag = DisposeBag()
9:
10: // ① bind を利用
11: nameTextField.rx.text
12:   .bind(to: nameLabel.rx.text)
13:   .disposed(by: disposeBag)
14:
15: // ② subscribe を利用
16: nameTextField.rx.text
17:   .subscribe(onNext: { [weak self] text in
18:     self?.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。2つのコードは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？

4.3.7 Operator

ここまでのコードでは、Observable から流れてきた値をそのまま subscribe（もしくは bind）するコードがほとんどでした。

ですが実際のプロダクションコードではそのまま subscribe（bind）することはほぼ無く、何か途中で値を加工して subscribe（bind）する場合があります。

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

する仕組みはよくあるパターンの1つです。

そこで活躍するのが Operator という概念です。

Operator は Observable に対してイベントの値の変換・絞り込み等、加工を施して新たに Observable を生成する仕組みです。

他にも、2つの Observable のイベントを合成・結合もできます。

Operator は色々なことができて全ての Operator の概要、使い方の説明だけで1冊の本が書けるレベルです。

本書とは少し趣旨がずれてしまうため、ここではよく使われる Operator にフォーカスを絞って、紹介します。

よく使われる Operator は次のとおりです。

- 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 のイベントが発生したら元の 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）**

RxSwift を書き始めたば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思うので、さらにスコープを狭めて、簡単でよく使うものをサンプルコードを加えてピックアップしました。

map

リスト 4.12: 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 { text -> String? in
4:       guard let text = text else { return nil }
5:       return "あと\(text.count) 文字"
6:   }
7:   .bind(to: fooTextLabel.rx.text)
8:   .disposed(by: disposeBag)
```

filter

リスト 4.13: 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: numberSubject
3:   .filter { $0 % 2 == 0 }
4:   .bind(to: evenSubject)
5:   .disposed(by: disposeBag)
```

また、**かならず Observable に流れるイベントの値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 4.14: 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: "k0uhashi")
3:
4: showUserNameButton.rx.tap
5:   .map { [weak self] in
6:     return self?.user.name
7:   }
8:   .bind(to: nameLabel.rx.text)
9:   .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 4.15: 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。

Observable には実は2種類の性質があり、Hot な Observable と Cold な Observable というのがあります。

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 したとき、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
- 単体では意味がない
- 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 4.16: Cond な Observable のサンプル

```
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 メソッドを実装できる簡単で便利な関数です。

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

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

さきほど作った myJust 関数は、disposed(by:) もしくは dispose() を呼ばなくても大丈夫です。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。なので明示的に購読を破棄する必要はありません。

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。
本章では実際にアプリを作りながら解説していきます。

まずは簡単なアプリから作ってみましょう。

いきなり RxSwift を使ってコードを書いても理解に時間がかかるかと思うので、1つのテーマごとに callback や delegate、KVO パターンを使って実装し、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！

5.1 カウンターアプリを作ってみよう！

この節ではカウンターアプリをテーマに callback、delegate、RxSwift、それぞれのパターンで実装したコードを比較し、どう書くかを学びます。

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 画面のイメージ

通信事業者 11:33



Delegateパターン: 1

カウントアップ

カウントダウン

リセット

図 5.1 作成するアプリのイメージ

5.1.3 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。






-  **Get started with a playground**
Explore new ideas quickly and easily.
 -  **Create a new Xcode project**
Create an app for iPhone, iPad, Mac, Apple Watch or Apple TV.
 -  **Clone an existing project**
Start working on something from an SCM repository.
- ☒ Show this window when Xcode launches

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。

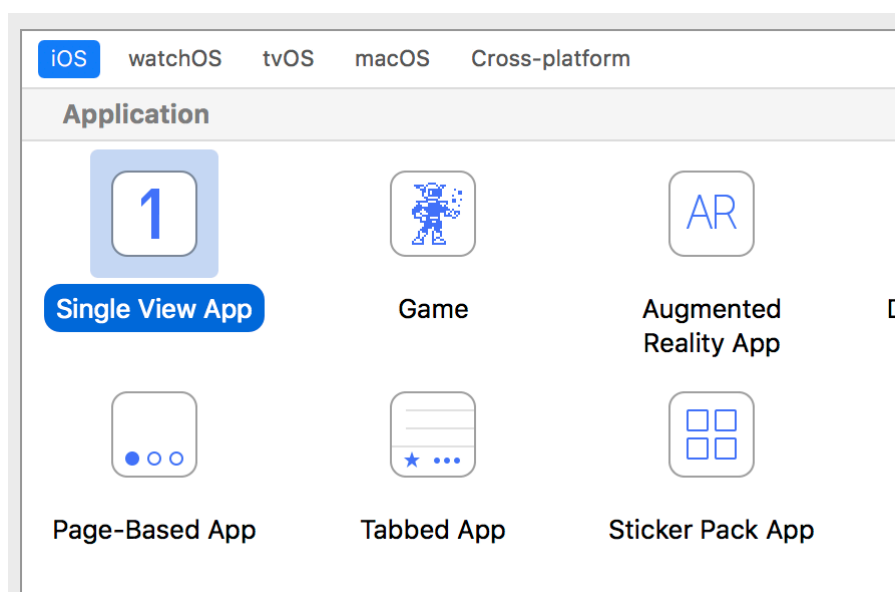
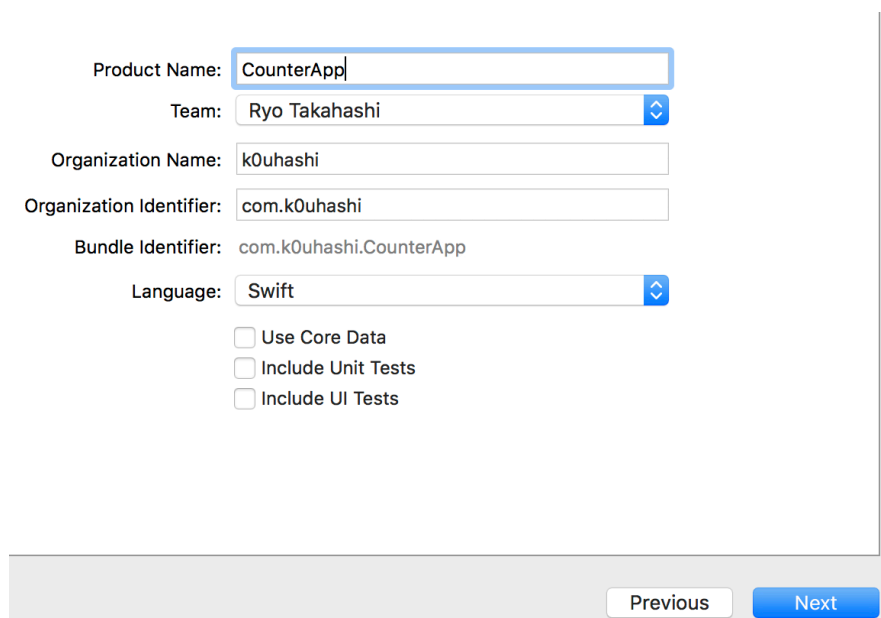


図 5.3 テンプレートの選択

テンプレートを選択します。Single View App を選択



Product Name: CounterApp

Team: Ryo Takahashi

Organization Name: k0uhashi

Organization Identifier: com.k0uhashi

Bundle Identifier: com.k0uhashi.CounterApp

Language: Swift

☐ Use Core Data

☐ Include Unit Tests

☐ Include UI Tests

Previous Next

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なら、一度 Xcode を終了します

5.1.4 環境設定

terminal.app を起動し、作成したプロジェクトのディレクトリまで移動します。

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

ライブラリの導入を行います。プロジェクト内で Cocoapods の初期化を行いましょう。

```
pod init
```

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。

第5章 簡単なアプリを作ってみよう！

```
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: # platform :ios, '9.0'
2:
3: target 'CounterApp' do
4:   use_frameworks!
5:
6:   pod 'RxSwift',      '~> 4.3.1' # ★この行を追加
7:   pod 'RxCocoa',      '~> 4.3.1' # ★この行を追加
8:
9: 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 (4.3.1)
Installing RxSwift (4.3.1)
Generating Pods project
Integrating client project

[!] Please close any current Xcode sessions and use
`CounterApp.xcworkspace` for this project from now on.Sending stats
Pod installation complete! There are 2 dependencies
from the Podfile and 2 total pods installed.

[!] Automatically assigning platform `ios` with version `11.4` on
target `CounterApp` because no platform was specified.
Please specify a platform for this target in your Podfile.
```

See ``https://guides.cocoapods.org/syntax/podfile.html=platform``.

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「CounterApp.xcworkspace」から開くようにしましょう

(Xcode 上、もしくは Finder 上で CounterApp.xcworkspace を指定しないと導入したライブラリが使いません)

5.1.5 開発を加速させる設定

■この節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、開発を加速させる設定を行います。

具体的には、Storyboard を廃止して ViewController.swift + xib を使って開発する手法に切り替えるための設定を行います。

Storyboard の廃止

Storyboard は画面遷移の設定が簡単にできたり、一見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、

アプリが大きくなってくると画面遷移が複雑になって見辛くなったり、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 しまくるなど色々な問題があるので、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
- /CounterApp/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、その項目を削除します。

- 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 の設定が失われるため、アプリを正しく起動できません。これを解消するために、AppDelegate.swift 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,
10:                     didFinishLaunchingWithOptions launchOptions:
11:                         [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
12:         self.window = UIWindow(frame: UIScreen.main.bounds)
13:         let navigationController =
14:             UINavigationController(rootViewController: ViewController())
15:         self.window?.rootViewController = navigationController
16:         self.window?.makeKeyAndVisible()
17:         return true
18:     }
19:
20: }
```

■**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新しく作成します。

- 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

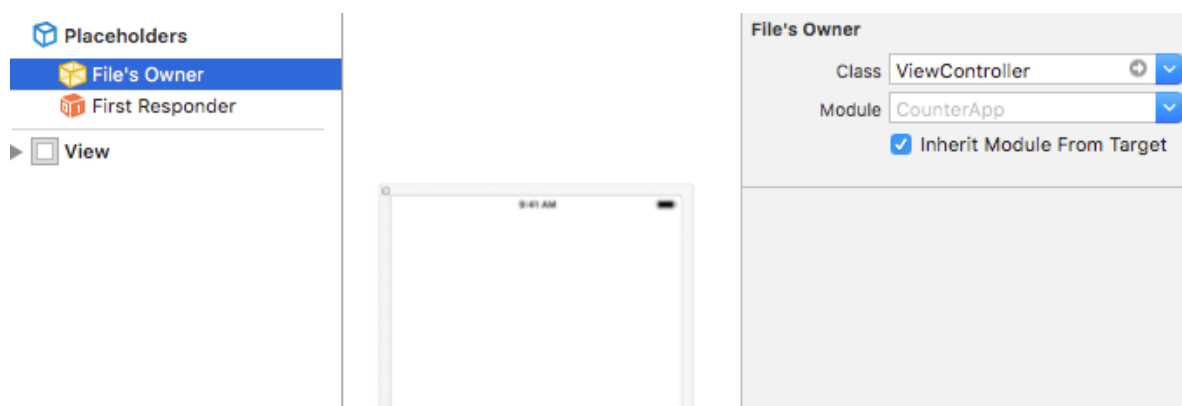


図 5.5 ViewController.xib の設定 1

Outlets の view と ViewController の View を接続します。

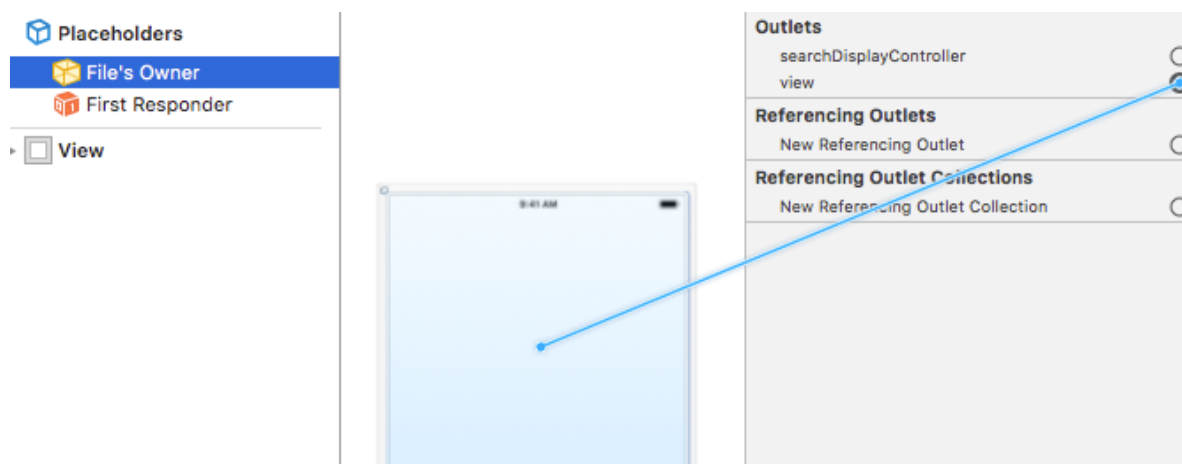


図 5.6 ViewController.xib の設定 2

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

起動に失敗する場合、ViewController.xib が正しく設定されているかもういちど確認してみましょう。

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。

■新しい画面を追加するときの手順まとめ

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 + Xib 構成にしたことによって、ViewController の生成が楽になり、画面遷移の実装がが少ない行で済むようになりました。

画面遷移は次のコードで実装できます。

リスト 5.3: 画面遷移の実装

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

5.1.6 CallBack パターンで作るカウンターアプリ

さて、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: }
```

スッキリしました。使わないメソッドやコメントは積極的に削除していきましょう。

次に、画面を作成します。

`UIButton` 3つと `UILabel` を1つ配置します。



図 5.8 部品の設置

UI 部品の配置が終わったら、ViewController.swift と UI を繋げます。
UILabel は IBOutlet、UIButton は IBAction として繋げていきます。

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ています。

- 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 を使うように修正します。

リスト 5.7: ViewController の修正

```
class ViewController: UIViewController {  
  
    @IBOutlet weak var countLabel: UILabel!  
  
    private var viewModel: ViewModel!  
  
    override func viewDidLoad() {
```

```
super.viewDidLoad()
viewModel = ViewModel()
}

@IBAction func countUp(_ sender: Any) {
    viewModel.incrementCount(callback: { [weak self] count in
        self?.updateCountLabel(count)
    })
}

@IBAction func countDown(_ sender: Any) {
    viewModel.decrementCount(callback: { [weak self] count in
        self?.updateCountLabel(count)
    })
}

@IBAction func countReset(_ sender: Any) {
    viewModel.resetCount(callback: { [weak self] count in
        self?.updateCountLabel(count)
    })
}

private func updateCountLabel(_ count: Int) {
    countLabel.text = String(count)
}
}
```

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対応するボタン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

5.1.7 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 の場合とまったく同じ動きをしていたら成功です。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対応する処理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、ボタンとメソッドの個数が1：1になっている問題がまだ残っています。

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り、コード全体の見通しが悪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。

5.1.8 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おきましょう。
まずは RxSwift を用いた 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

```
// 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）

// ボタンの入力シーケンス
struct RxViewModelInput {
    let countUpButton: Observable<Void>
    let countDownButton: Observable<Void>
    let countResetButton: Observable<Void>
}
```

第5章 簡単なアプリを作ってみよう！

```
// ViewController に監視させる対象を定義
protocol RxViewModelOutput {
    var counterText: Driver<String> { get }
}

// ViewModel に継承させる protocol を定義
protocol RxViewModelType {
    var outputs: RxViewModelOutput? { get }
    init(input: RxViewModelInput)
}
```

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

ViewController も修正しましょう。

全ての IBAction と接続を削除して IBOutlet を定義し、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 {
```


第5章 簡単なアプリを作ってみよう！

```
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(
22:          countUpButton: countUpButton.rx.tap.asObservable(),
23:          countDownButton: countDownButton.rx.tap.asObservable(),
24:          countResetButton: countResetButton.rx.tap.asObservable()
25:      )
26:      viewModel = RxViewModel(input: input)
27:
28:      viewModel.outputs?.counterText
29:          .drive(countLabel.rx.text)
30:          .disposed(by: disposeBag)
31:  }
32: }
```

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。

Tips: あれ？ コード間違っていないのにクラッシュする？ そんな時は

新しい画面を作成・既存の画面をいじっていて、ふと Build & Run を実行したとき、あれ？ コードに手を加えてないのにクラッシュするようになった？？ と不思議になる場面は初心者の頃はあるあるな問題かと思います。

そんなときは、1 度いじっていた xib/storyboard の IBAction の接続・接続解除、IBOutlet の接続・接続解除が正しくできているか確認してみましょう。

ViewController 内では、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の通知を行わなくてもよい

- テストがかきやすくなった

- ・ **デメリット**

- 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

一番大きなメリットはやはり「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 を自動で変更させているため、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、今回は分離ができていたので書きやすいです。

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ようなイメージです。

また、RxSwift+MVVM の書き方は慣れるまで時間がかかるかと思うので、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1つの方法です。

5.1.9 まとめ

この章では、callback、delegate、RxSwift、3つのパターンでカウンターアプリを作りました。

callback、delegate パターンで課題であった UI とデータの分離できていない問題に関しては、RxSwift を用いたことで解決できました。

全ての開発において RxSwift を導入した書き方が正しいとは限りませんが、1つの解決策として覚えておくだけでもよいと思います。

おまけ：カウンターアプリを昇華させよう 1

この章はおまけです。さきほど作ったコードに加えて、次の機能を追加してみましょう！

- 追加の機能要件
 - +10 カウントアップできる
 - -10 カウントダウンできる
 - カウンターの値を DB に保存しておいて、復帰時に DB から参照させるように変更

5.2 WebView アプリを作ってみよう！

この章では WKWebView を使ったアプリをテーマに、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

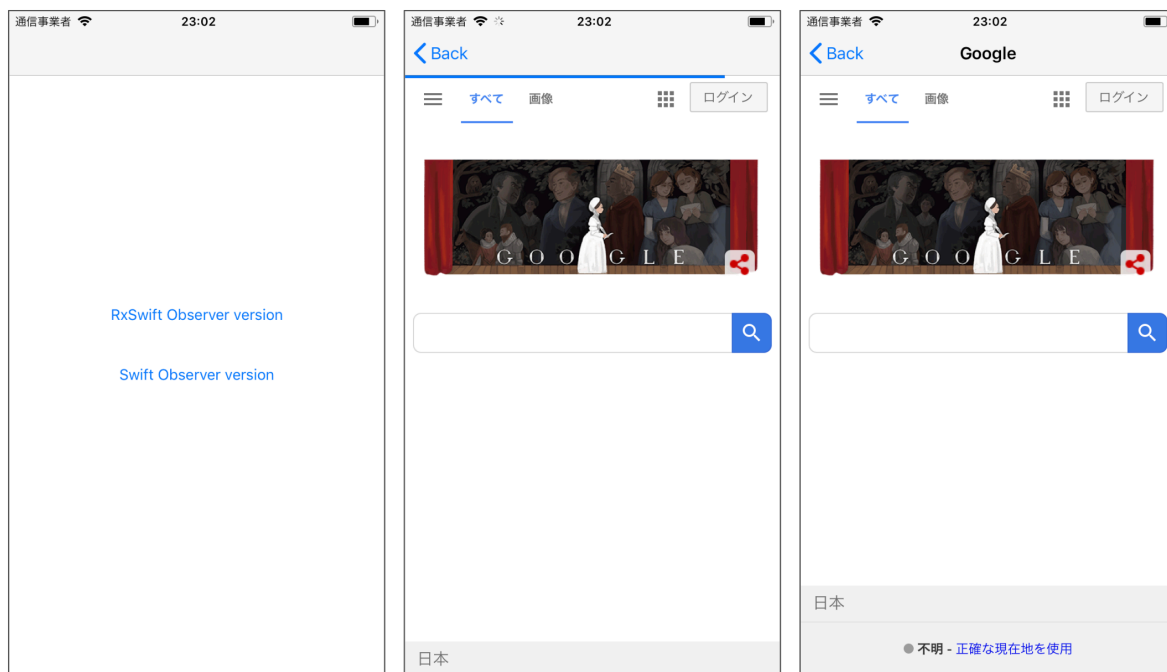


図 5.9 アプリのイメージ

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ようなアプリを作ります。

サクっといきましょ！ まずは新規プロジェクトを作成します。
プロジェクトの設定や ViewController の設定は第5章の「開発を加速させる設定」を参照してください。

ここでは WKWebViewController.xib という名前で画面を作成し、中に WKWebView と UIProgressView を配置します。

画面ができたなら、ViewController クラスを作っていきます。

リスト 5.14: KVO で実装する

```
import UIKit
import WebKit

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

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

private func setupWebView() {
    // webView.isLoading の値の変化を監視
    webView.addObserver(self, forKeyPath: "loading",
        options: .new, context: nil)
    // webView.estimatedProgress の値の変化を監視
    webView.addObserver(self, forKeyPath: "estimatedProgress",
        options: .new, context: nil)

    let url = URL(string: "https://www.google.com/")
    let urlRequest = URLRequest(url: url!)
    webView.load(urlRequest)
    progressView.setProgress(0.1, animated: true)
}

deinit {
    // 監視を解除
    webView?.removeObserver(self, forKeyPath: "loading")
    webView?.removeObserver(self, forKeyPath: "estimatedProgress")
}

override func observeValue(forKeyPath keyPath: String?,
    of object: Any?, change: [NSKeyValueChangeKey : Any]?,
    context: UnsafeMutableRawPointer?) {
    if keyPath == "loading" {
        UIApplication.shared
            .isNetworkActivityIndicatorVisible = webView.isLoading
        if !webView.isLoading {
            // ロード完了時に ProgressView の進捗を 0.0(非表示) にする
            progressView.setProgress(0.0, animated: false)
            // 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
            navigationItem.title = webView.title
        }
    }
    if keyPath == "estimatedProgress" {
        // ProgressView の進捗状態を更新
        progressView
            .setProgress(Float(webView.estimatedProgress), animated: true)
    }
}
```

```
}
```

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。KVO は 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。

そのため、NSObject を継承できない構造体 (struct) は KVO の仕組みが使いません。

KVO を Swift で使うためにはオブジェクトを class で定義し、プロパティに objc 属性と dynamic をつけます。WKWebView のプロパティのうち、title、url、estimatedProgress は標準で KVO に対応しているので、今回はそれを使います。

では実際コード内で何をしているかというと、viewDidLoad() 時に WebView のプロパティの値を監視させて、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。

この方法では全ての値変化の通知を 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ません。

さらに、KVO を使った場合の注意点として addObserver した場合、dealloc 時に removeObserver を呼ばないと、最悪の場合メモリリークを引き起こし、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である以上、絶対にいつか忘れます。

クラスが肥大化してくるにつれ、その確率は上がってきます。

こういった問題は RxSwift を使うことで簡単に解決できます！ 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なるライブラリです。

Podfile にライブラリを追加しましょう

リスト 5.15: Podfile の修正

```
pod 'RxSwift',      '~> 4.3.1'
pod 'RxCocoa',      '~> 4.3.1'
pod 'RxOptional',   '~> 3.5.0'
```

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)

// UINavigationController 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りました。

色々説明するポイントはありますが、この章ではじめ出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、子オブジェクトのみ監視できます。

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、`self` 内のパラメータに対して `rx.observe` してしまうと、循環参照を引き起こし最悪の場合アプリがクラッシュします。

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。

これは `NSValue` から値を手動で抽出する仕組みを使っていて、RxCocoa ライブラリ内の `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。

コードで比較するとわかりやすいです、次のコードを見てみましょう。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

```
// RxSwift
Observable<String?>
    .of("One",nil,"Three")
    .filter { $0 != nil }
    .map { $0! }
    .subscribe { print($0) }

// RxOptional
Observable<String?>
    .of("One", nil, "Three")
    .filterNil()
    .subscribe { print($0) }
```

share()

一言で説明すると、Cold な Observable を Hot な Observable へ変換する Operator です。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

```
let text = textField.rx.text
    .map { text -> String in
        print("call")
        return "☆☆\(text) ☆☆"
    }

text
    .bind(to: label1.rx.text)
    .disposed(by: disposeBag)

text
    .bind(to: label2.rx.text)
    .disposed(by: disposeBag)

text
    .bind(to: label3.rx.text)
    .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、ストリームの途中で値を加工して複数の Label へ bind しています。

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
```

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。

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みましょう。

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。(実態は Observable ですが。)

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が保証されています。

また、実はこれは 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ると説明しました。

そのため、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ね。

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。

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「1 2 3」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 Operator で、これ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 クラス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。

本題へ

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。
特に、`removeObserver` を気にしなくてもよくなるので多少は安全です。

`removeObserver` を気にしなくてもよくなったというよりは、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ない？ と思うかもしれませんが、RxSwift では呼び忘れると Warning が出るので `removeObserver` だったころより忘れる確率はグッと低くなります。

しかし、書きやすくなったといっても、まだこの書き方では次の問題が残っています。

- Key 値がベタ書きになっている
- 購読する値の型を指定してあげないといけない

自分で extension を定義するのも1つの方法としてありますが、
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

```
pod 'RxSwift',      '~> 4.3.1'
pod 'RxCocoa',      '~> 4.3.1'
pod 'RxOptional',   '~> 3.5.0'
pod 'RxWebKit',     '~> 0.3.7'
```

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:        // UINavigationController 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:    }
```

第5章 簡単なアプリを作ってみよう！

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

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あがりました。

RxWebKit は、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` や `bind` することもできるので、いろいろな用途に使えそうですね。

第6章

さまざまな RxSwift 系ライブラリ

この章では RxSwift の拡張ライブラリについて紹介していきます。拡張ライブラリといっても、かなり種類があるのでその中から私がよく使うものをピックアップして紹介していきます。

6.1 RxDataSources

スター数：1,521（2018年09月22日時点）

RxDataSources はざっくりいうと、UITableView、UICollectionView を RxSwift の仕組みを使ってイイ感じに差分更新をしてくれるライブラリです。

このライブラリを使うと、UITableView や UICollectionView を使ったアプリを作る際に delegate の実装の負担が少なく済むようになったり、セクションを楽に組み立てられるようになったりします。

6.1.1 作ってみよう！

RxDataSources を使って簡単な UITableView アプリを作ってみましょう。

6.1.2 イメージ



図 6.1 RxDataSources+UITableView のサンプル

- 新規プロジェクトを SingleViewApp で作成
- ライブラリの導入

```
pod init
vi Podfile
```

リスト 6.1: Podfile

```
platform :ios, '11.4'
use_frameworks!

target 'RxDataSourceExample' do
  pod 'RxSwift', '~> 4.3.1'
```

```
pod 'RxCocoa', '~> 4.3.1'
pod 'RxDataSources', '~> 3.1.0'
end
```

```
pod install
```

5章の開発を加速させる設定を済ませた前提で進めます。

まずは SectionModel というのを作成します

SectionModel は SectionModelType プロトコルに準拠する構造体で定義されており、これをうまく使うことでセクションとその中のセクションセルを表現できます。

リスト 6.2: SettingsSectionModel

```
import UIKit
import RxDataSources

typealias SettingsSectionModel = SectionModel<SettingsSection, SettingsItem>

enum SettingsSection {
    case account
    case common

    var headerHeight: CGFloat {
        return 40.0
    }

    var footerHeight: CGFloat {
        return 1.0
    }
}

enum SettingsItem {
    // account section
    case account
    case security
    case notification
    case contents
    // common section
```

```
case sounds
case dataUsing
case accessibility

// other
case description(text: String)

var title: String? {
    switch self {
    case .account:
        return "アカウント"
    case .security:
        return "セキュリティ"
    case .notification:
        return "通知"
    case .contents:
        return "コンテンツ設定"
    case .sounds:
        return "サウンド設定"
    case .dataUsing:
        return "データ利用時の設定"
    case .accessibility:
        return "アクセシビリティ"
    case .description:
        return nil
    }
}

var rowHeight: CGFloat {
    switch self {
    case .description:
        return 72.0
    default:
        return 48.0
    }
}

var accessoryType: UITableViewCell.AccessoryType {
    switch self {
    case .account, .security, .notification, .contents,
        .sounds, .dataUsing, .accessibility:
        return .disclosureIndicator
    case .description:
        return .none
    }
}
```

```
    }  
}
```

enum で定義した `SettingsSection` の各 case が1つのセクションで、`SettingsItem` がセクション内のセルデータ群です。

次に、ViewModel を作っていきましょう。

リスト 6.3: `SettingsViewModel.swift`

```
import RxSwift  
import RxDataSources  
  
class SettingsViewModel {  
  
    let items = BehaviorSubject<[SettingsSectionModel]>(value: [])  
  
    func updateItem() {  
        let sections: [SettingsSectionModel] = [  
            accountSection(),  
            commonSection()  
        ]  
        items.onNext(sections)  
    }  
  
    private func accountSection() -> SettingsSectionModel {  
        let items: [SettingsItem] = [  
            .account,  
            .security,  
            .notification,  
            .contents  
        ]  
        return SettingsSectionModel(model: .account, items: items)  
    }  
  
    private func commonSection() -> SettingsSectionModel {  
        let items: [SettingsItem] = [  
            .sounds,  
            .dataUsing,  
            .accessibility,  
            .description(text: "基本設定はこの端末でログインしている全てのアカウントに適用されます。")  
        ]  
        return SettingsSectionModel(model: .common, items: items)  
    }  
}
```

```
    ]  
    return SettingsSectionModel(model: .common, items: items)  
  }  
}
```

最後に ViewController を作ります。

リスト 6.4: SettingsViewController.swift

```
import UIKit  
import RxSwift  
import RxDataSources  
  
class SettingsViewController: UIViewController {  
  
    @IBOutlet weak var tableView: UITableView!  
  
    private var disposeBag = DisposeBag()  
  
    private lazy var dataSource =  
        RxTableViewSectionedReloadDataSource<SettingsSectionModel>(  
            configureCell: configureCell)  
  
    private lazy var configureCell:  
        RxTableViewSectionedReloadDataSource<SettingsSectionModel>.ConfigureCell =  
    { [weak self] (dataSource, tableView, indexPath, _) in  
        let item = dataSource[indexPath]  
        switch item {  
        case .account, .security, .notification, .contents,  
             .sounds, .dataUsing, .accessibility:  
            let cell = tableView  
                .dequeueReusableCell(withIdentifier: "cell", for: indexPath)  
            cell.textLabel?.text = item.title  
            cell.accessoryType = item.accessoryType  
            return cell  
        case .description(let text):  
            let cell = tableView  
                .dequeueReusableCell(withIdentifier: "cell", for: indexPath)  
            cell.textLabel?.text = text  
            cell.isUserInteractionEnabled = false  
            return cell  
        }  
    }  
}
```

```
private var viewModel: SettingsViewModel!

override func viewDidLoad() {
    super.viewDidLoad()
    setupViewController()
    setupTableView()
    setupViewModel()
}

private func setupViewController() {
    navigationItem.title = "設定"
}

private func setupTableView() {
    tableView
        .register(UITableViewCell.self, forCellReuseIdentifier: "cell")
    tableView.contentInset.bottom = 12.0
    tableView.rx.setDelegate(self).disposed(by: disposeBag)
    tableView.rx.itemSelected
        .subscribe(onNext: { [weak self] indexPath in
            guard let item = self?.dataSource[indexPath] else { return }
            self?.tableView.deselectRow(at: indexPath, animated: true)
            switch item {
            case .account:
                // 遷移させる処理
                // コンパイルエラー回避のために break をかいていますが処理を書いている break は
                // 必要ありません。
                break
            case .security:
                // 遷移させる処理
                break
            case .notification:
                // 遷移させる処理
                break
            case .contents:
                // 遷移させる処理
                break
            case .sounds:
                // 遷移させる処理
                break
            case .dataUsing:
                // 遷移させる処理
                break
            case .accessibility:
```

```
        // 遷移させる処理
        break
    case .description:
        break
    }
})
.dispose(by: disposeBag)
}

private func setupViewModel() {
    viewModel = SettingsViewModel()

    viewModel.items
        .bind(to: tableView.rx.items(dataSource: dataSource))
        .disposed(by: disposeBag)

    viewModel.updateItem()
}
}

extension SettingsViewController: UITableViewDelegate {
    func tableView(_ tableView: UITableView,
                   heightForRowAt indexPath: IndexPath) -> CGFloat {
        let item = dataSource[indexPath]
        return item.rowHeight
    }

    func tableView(_ tableView: UITableView,
                   heightForHeaderInSection section: Int) -> CGFloat {
        let section = dataSource[section]
        return section.model.headerHeight
    }

    func tableView(_ tableView: UITableView,
                   heightForFooterInSection section: Int) -> CGFloat {
        let section = dataSource[section]
        return section.model.footerHeight
    }

    func tableView(_ tableView: UITableView,
                   viewForHeaderInSection section: Int) -> UIView? {
        let headerView = UIView()
        headerView.backgroundColor = .clear
        return headerView
    }
}
```

```
func tableView(_ tableView: UITableView,
               viewForFooterInSection section: Int) -> UIView? {
    let footerView = UIView()
    footerView.backgroundColor = .clear
    return footerView
}
```

最後に ViewController.xib を作成し、画面幅いっぱいの TableView を設置、ViewController.swift の IBOutlet と接続しましょう。

Build & Run で実行し、イメージのようになっていたら成功です。

6.1.3 その他セクションを追加してみよう！

さきほど作った RxDataSources+UITableView のサンプルアプリを題材に、新しくセクションとセクションアイテムを追加する方法について学びます。

まずはセクションを追加するために、SettingsSection に case を追加します。

```
enum SettingsSection {
    case account
    case common
    case other // 追加
    // ...
}
```

次に、セクションアイテムを追加するため、SettingsItem に case を追加します。

リスト 6.5: SettingsItem

```
enum SettingsItem {
    // ...
    // common section
    case sounds
    case dataUsing
    case accessibility
    // other section
}
```



```
case credits // 追加
case version // 追加
case privacyPolicy // 追加
// ...

var title: String? {
    switch self {
    // ..
    // 追加
    case .credits:
        return "クレジット"
    case .version:
        return "バージョン情報"
    case privacyPolicy:
        reutrn "プライバシーポリシー"
    }
}

var accessoryType: UITableViewCell.AccessoryType {
    switch self {
    case .account, .security, .notification, .contents, .sounds,
        .dataUsing, .accessibility,
        .credits, .version, .privacyPolicy: // 追加
        return .disclosureIndicator
    case .description:
        return .none
    }
}
}
```

セクションとそのアイテムの定義ができたなら、実際に表示させるために ViewModel の items ヘデータを追加します。

リスト 6.6: ViewModel を編集

```
func updateItem() {
    let sections: [SettingsSectionModel] = [
        accountSection(),
        commonSection(),
        otherSection() // 追加
    ]
}
```

```
        items.onNext(sections)
    }

    // ...
    // 追加
    private func otherSection() -> SettingsSectionModel {
        let items: [SettingsItem] = [
            .credits,
            .version,
            .privacyPolicy
        ]
        return SettingsSectionModel(model: .other, items: items)
    }
```

データの追加ができたので、今度はそのセクションセルの UI を定義します。今回は他のメニューと同じ UI でよいので、switch 文を軽く対応させるだけです。

リスト 6.7: ViewController を編集

```
// ...
private lazy var configureCell:
    RxTableViewSectionedReloadDataSource<SettingsSectionModel>.ConfigureCell =
    { [weak self] (dataSource, tableView, indexPath, _) in
        let item = dataSource[indexPath]
        switch item {
        case .account, .security, .notification, .contents,
            .sounds, .dataUsing, .accessibility,
            .credits, .version, .privacyPolicy: // 追加
            let cell = tableView.dequeueReusableCell
                (withIdentifier: "cell", for: indexPath)
            cell.textLabel?.text = item.title
            cell.accessoryType = item.accessoryType
            return cell
        }
    }

// ...

private func setupTableView() {
    // ...
    tableView.rx.itemSelected
        .subscribe(onNext: { [weak self] indexPath in
```

```
guard let item = self?.dataSource[indexPath] else { return }
self?.tableView.deselectRow(at: indexPath, animated: true)
switch item {
// ...
// 追加
case .credits:
    // 遷移させる処理
    break
case .version:
    // 遷移させる処理
    break
case .privacyPolicy:
    // 遷移させる処理
    break
case .description:
    break
}
})
.disposed(by: disposeBag)
// ...
```

Build & Run で次のような画面になっていたら成功です。



図 6.2 その他セクション追加後の画面

RxDatasources と UICollectionView を組み合わせた書き方もほぼ同じ手順で実装できるので、参考にしてみてください。

6.2 RxKeyboard

RxKeyboard はキーボードの frame の値の変化を RxSwift で楽に購読できるようにする拡張ライブラリです。

キーボードの真上に View を配置して、キーボードの高さに応じて UIView を動かしたり、ScrollView を動かしたりできるようになります。

Qiita に「iMessage の入力 UI のようなキーボードの表示と連動する UI を作る with RxSwift, RxKeyboard」というタイトルで RxKeyboard を使ったサンプルコードの記事を書いているので、興味のあるかたは参照してください。

- Qiita 「iMessage の入力 UI のようなキーボードの表示と連動する UI を作る with RxSwift, RxKeyboard」
– <https://qiita.com/k0uhashi/items/671ec40520967bc3949f>

6.3 RxOptional

本書でも RxOptional について少し触れましたが、Optional な値が流れるストリームに対していろいろできるようにする拡張ライブラリです。

たとえば、次のように使うことができます。

リスト 6.8: filterNil

```
Observable<String?>
  .of("One", nil, "Three")
  .filterNil()
// Observable<String?> -> Observable<String>
  .subscribe { print($0) }
```

```
One
Three
```

replaceNilWith オペレータを使うことで nil 値の置き換え操作もできます。

リスト 6.9: replaceNilWith

```
Observable<String?>
  .of("One", nil, "Three")
  .replaceNilWith("Two")
// Observable<String?> -> Observable<String>
  .subscribe { print($0) }
```

```
One
Two
Three
```

他にも、`errorOnNil` オペレータを使うと `nil` が流れてきたときに `error` を流すことができます。

リスト 6.10: `errorOnNil`

```
Observable<String?>
    .of("One", nil, "Three")
    .errorOnNil()
// Observable<String?> -> Observable<String>
    .subscribe { print($0) }
```

```
One
Found nil while trying to unwrap type <Optional<String>>
```

`Array`、`Dictionary`、`Set` に対しても使うことができます。

リスト 6.11: `filterEmpty`

```
Observable<[String]>
    .of(["Single Element"], [], ["Two", "Elements"])
    .filterEmpty()
    .subscribe { print($0) }
```

```
["Single Element"]
["Two", "Elements"]
```

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！（自戒）

7.1 開発中のアプリに導入

次は、実際に動い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、一番手っ取り早い方法は個人で作っているアプリに導入してみることです。
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ももちろんよいですが、必ずプロジェクトのメンバーと相談の上、導入しましょう。
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 本を書いてみるのもよいです！ むしろ読みたいので書いてほしいです！ お願いします！
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、というサイクルを継続して回していくのが一番よい方法かと思いますが、早速やっていきましょう！

7.2 コミュニティへの参加

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。
日本国内での RxSwift 専用のコミュニティは筆者の観測内では見つけれませんでした、

国内外問わずの RxSwift Community (GitHub Project) という、RxSwift の公式 Slack ワークスペースであればあるようなので、興味があるかたは RxSwift のリポジトリの README.md に URL が載っているので参照してみてください。

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存在しているようです。

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、（というかりモート勉強会）よいソリューションだと思うんですがあまり無いんですよね・・・。Discord 上で開催されている「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。

自分が立ち上げている「iOS アプリ開発がんばるぞ！！」は定期的開催しているので、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！

- 参考 URL 一覧
 - swift-developers-japan
 - <https://medium.com/swift-column/discord-ios-20d586e373c0>
 - iOSDC
 - <https://iosdc.jp/2018/>
 - try! Swift
 - <https://www.tryswift.co/>
 - iOS アプリ開発がんばるぞ！！ の会
 - <https://ios-app-yaru.connpass.com/>

第8章

補足

8.1 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

k0uhashi （本名：高橋 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数十個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。

(メモとして使ってください)

これは完全な余談ですが、同人誌って紙の消費量の関係で PDF 化したときにページ数を 4 の倍数にしないと印刷所に怒られるんですよ。でも 4 の倍数にするのめっちゃむずかしくてやむを得ずこのような余白ページを作りました。

いろんな本を見てるとたまーに「ここはメモとして使ってください」っていうページがあって、??? となっていたんですが長年の謎がようやく解けました。

比較して学ぶ RxSwift4 入門

2018 年 10 月 8 日 技術書典 5 版 v1.0.0

著 者 k0uhashi

編 集 k0uhashi

発行所 iOS アプリ開発がんばるぞ！！の会

印刷所 日光企画 様

(C) 2018 iOS アプリ開発がんばるぞ！！の会