



Next Publishing レビュー

目次

はじめに	4
対象読者	4
必須知識	4
推奨知識	5
想定環境	5
お問い合わせ先	5
ツイート	5
免責事項	5
ライセンス	5
Swift	5
RxSwift	6
RxWebKit	6
RxDataSources	6
第1章 iOSアプリ開発とSwift	7
第2章 RxSwift入門	8
2.1 覚えておきたい用語と1行概要	8
2.2 RxSwiftって何?	8
2.3 Reactive Extensionsって何?	8
2.3.1 概念	8
2.3.2 歴史	8
2.4 リアクティブプログラミングとは?	9
2.4.1 リアクティブプログラミングとExcel	10
2.5 RxSwiftの特徴	11
2.6 RxSwiftは何が解決できる?	11
第3章 RxSwiftの導入	23
3.1 導入要件	23
3.2 導入方法	23
第4章 RxSwiftの基本的な書き方	25
4.1 メソッドチェーンのように直感的に書ける	25
4.2 Hello World	26
4.3 よく使われるクラス・メソッドについて	28
4.3.1 Observable	28
4.3.2 Dispose	30
4.3.3 Subject、Relay	32

4.3.4	バッファについて	32
4.3.5	それぞれの使い分け	32
4.3.6	bind	34
4.3.7	Operator	34
4.4	Hot な Observable と Cold な Observable	37
第5章	簡単なアプリを作ってみよう！	39
5.1	カウンターアプリを作ってみよう！	39
5.1.1	機能要件	39
5.1.2	画面のイメージ	40
5.1.3	プロジェクトの作成	40
5.1.4	環境設定	41
5.1.5	開発を加速させる設定	42
5.1.6	CallBack パターンで作るカウンターアプリ	46
5.1.7	Delegate で作るカウンターアプリ	49
5.1.8	RxSwift で作るカウンターアプリ	51
5.1.9	まとめ	55
5.2	WebView アプリを作ってみよう！	56
5.2.1	この章のストーリー	56
5.2.2	イメージ	56
第6章	さまざまな RxSwift 系ライブラリ	67
6.1	RxDataSources	67
6.1.1	作ってみよう！	67
6.1.2	イメージ	68
6.1.3	その他セクションを追加してみよう！	76
6.2	RxKeyboard	79
6.3	RxOptional	80
第7章	次のステップ	82
7.1	開発中のアプリに導入	82
7.2	コミュニティへの参加	82
第8章	補足	84
8.1	参考 URL ・ ドキュメント ・ 文献	84
	著者紹介	85

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、RxSwift それぞれを使った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。…主語が大きいですが、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。（もちろん使われていない現場もありますよ）

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ました。

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 ！ となって頂けたら幸いです。

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 の基礎知識や iOS アプリ開発における基礎知識については本書では解説しません。

- ・ 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、UITextView、UITableView、UICollectionView

推奨知識

以下を知っておくとより理解が進みます。

- ・ 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>

ツイート

本書に関するツイートはハッシュタグ『**#比較して学ぶRxSwift**』を付与してのツイートをお願いします。感想、批評、何でも良いのでつぶやいて頂けると作者が喜びます！

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。

ライセンス

Swift

License

Apache License 2.0

<https://github.com/apple/swift/blob/master/LICENSE.txt>

RxSwift

License

The MIT License Copyright © 2015 Krunoslav Zaher All rights reserved.

<https://github.com/ReactiveX/RxSwift/blob/master/LICENSE.md>

RxWebKit

License

The MIT License Copyright © 2016 Daichi Ichihara All rights reserved.

<https://github.com/RxSwiftCommunity/RxWebKit/blob/master/LICENSE.md>

RxDataSources

License

Copyright (c) 2017 RxSwift Community

<https://github.com/RxSwiftCommunity/RxDataSources/blob/master/LICENSE.md>

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、初心者でも更に簡単に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、「複雑な非同期処理を実装した場合、callback地獄で読み辛くなってしまう」「処理の成功・失敗の制御が統一しにくい（例：通信処理）」などがあります。これらを解決するひと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見ながら学んでいきましょう！

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 RxSwiftと一緒に導入されることが多い

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）の実装が用意に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。

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。その中のひと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: 疑似コード

```
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」と答えるのではないのでしょうか？。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。

リスト2.2: リアクティブプログラミングの観点から見た疑似コード

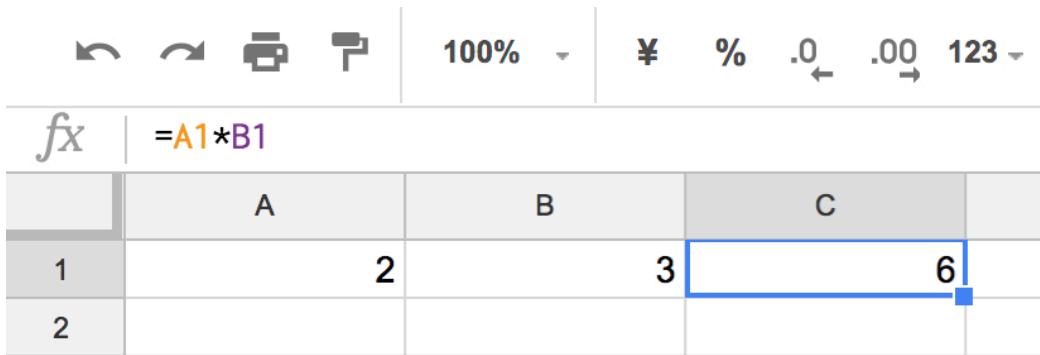
```
a = 2
b = 3
c = a * b
// a: 2, b:3, c:6
a = 3
// a: 3, b:3, c:9
```

```
print(c)
// 出力: 9
```

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。

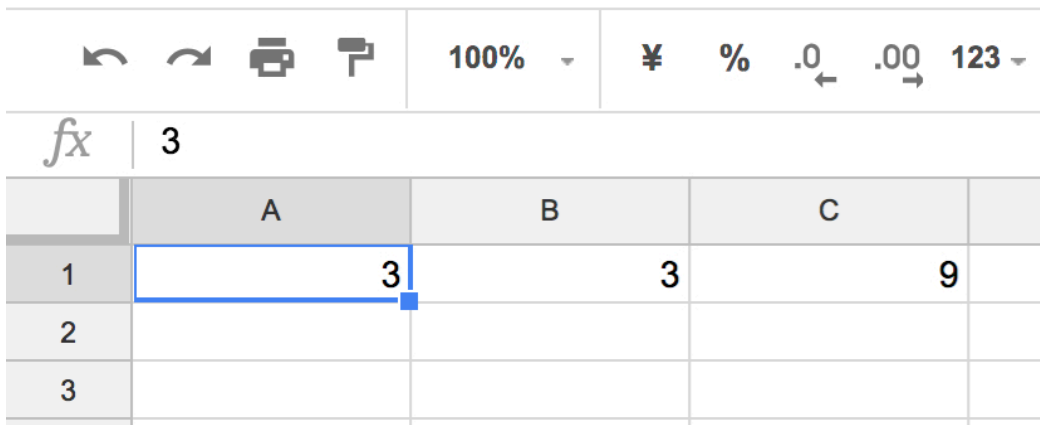
図2.1: Excel



	A	B	C
1	2	3	6
2			

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。

図2.2: Excel



	A	B	C
1	3	3	9
2			
3			

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、概念的にはこう考えると理解しやすいかも知れません。

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
- ・ 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
- ・ callbackを減らせる

—インデントの浅いコードに

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？

わかりやすいのは「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



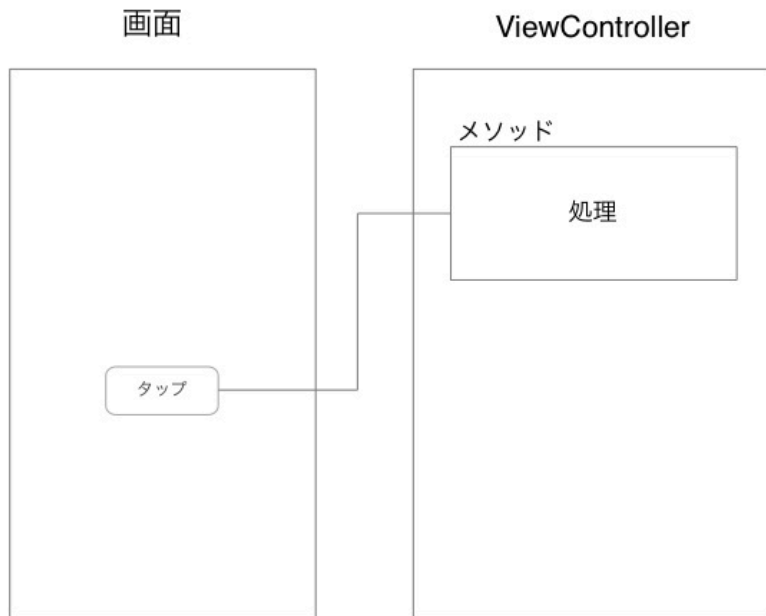
まずは従来のIBActionを使った方法で作ってみましょう。

リスト 2.3: IBAction を用いたコード

```
1: class SimpleTapViewController: UIViewController {
2:
3:   @IBOutlet weak var messageLabel: UILabel!
4:
5:   @IBAction func tapLoginButton(_ sender: Any) {
6:       messageLabel.text = "Tap Login Button!"
7:   }
8:
9: }
```

通常の書き方だと、ひとつのボタンに対してひとつの関数を定義します。図に表してみましょう。

図 2.4: IBAction を用いたコード (イメージ)



仕様がシンプルなため、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、ふたつボタンを増やすコードを書いてみましょう

リスト 2.4: IBAction を使ったコードを拡張する

```
1: class SimpleTapViewController: UIViewController {
2:
3:     @IBOutlet weak var messageLabel: UILabel!
4:
5:     @IBAction func tapLoginButton(_ sender: Any) {
6:         messageLabel.text = "Tap Login Button!"
7:     }
8:
9:     @IBAction func tapResetPasswordButton(_ sender: Any) {
10:         messageLabel.text = "Tap Reset Password Button!"
11:     }
12:
13:     @IBAction func tapExitButton(_ sender: Any) {
14:         messageLabel.text = "Tap Exit Button!"
```

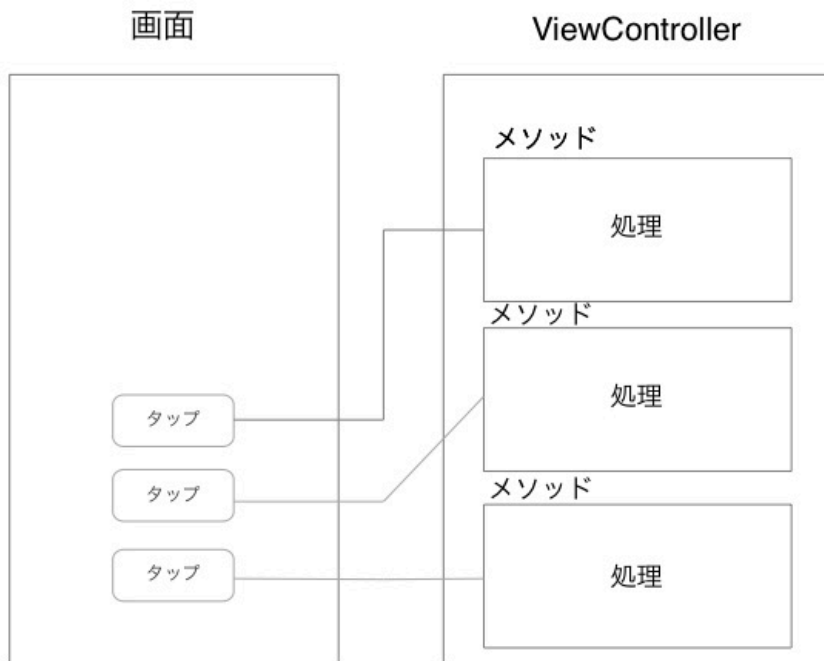
```

15:     }
16:
17:     @IBAction func tapHelpButton(_ sender: Any) {
18:         messageLabel.text = "Tap Help Button!"
19:     }
20: }

```

図で表してみます。

図 2.5: IBAction を使ったコードを拡張する (イメージ)



ボタンをひとつ増やすたびに対応する関数がひとつずつ増えていき、肥大化するとだんだん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 2.5: RxSwift を用いたコード

```

1: import RxSwift
2: import RxCocoa
3:

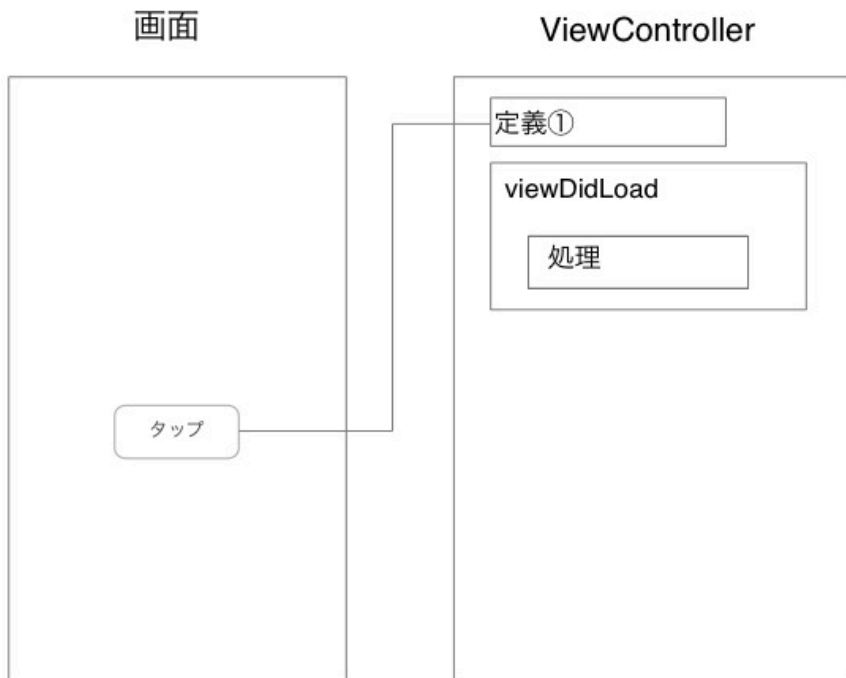
```

```

4: class SimpleRxTapViewController: UIViewController {
5:
6:     @IBOutlet weak var loginButton: UIButton!
7:     @IBOutlet weak var messageLabel: UILabel!
8:
9:     private let disposeBag = DisposeBag()
10:
11:     override func viewDidLoad() {
12:         super.viewDidLoad()
13:         loginButton.rx.tap
14:             .subscribe(onNext: { [weak self] in
15:                 self?.messageLabel.text = "Tap Login Button!"
16:             })
17:             .disposed(by: disposeBag)
18:     }
19: }

```

図2.6: RxSwiftを用いたコード (図)



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、ひとつのボタンとひとつの関数が強く結合していたのが、ひとつのボタンとひとつのプロパティの結合で済むようになっていて、UI とコード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ひとつ増やすたびにに対応するプロパティが 1 行増えるだけなので、アプリが大きくなってくると RxSwift で書いたほうが読みやすくなります。(もちろん、例外もあります。)

実際にもう少しボタンを増やしてみましょう。

リスト 2.6: RxSwift を用いたコードを拡張

```
1: import RxSwift
2: import RxCocoa
3:
4: class SimpleRxTapViewController: UIViewController {
5:
6:     @IBOutlet weak var loginButton: UIButton!
7:     @IBOutlet weak var resetPasswordButton: UIButton!
8:     @IBOutlet weak var exitButton: UIButton!
9:     @IBOutlet weak var helpButton: UIButton!
10:    @IBOutlet weak var messageLabel: UILabel!
11:
12:    private let disposeBag = DisposeBag()
13:
14:    override func viewDidLoad() {
15:        super.viewDidLoad()
16:        loginButton.rx.tap
17:            .subscribe(onNext: { [weak self] in
18:                self?.messageLabel.text = "Tap Login Button!"
19:            })
20:            .disposed(by: disposeBag)
21:
22:        resetPasswordButton.rx.tap
23:            .subscribe(onNext: { [weak self] in
24:                self?.messageLabel.text = "Tap Reset Password Button!"
25:            })
26:            .disposed(by: disposeBag)
27:
28:        exitButton.rx.tap
29:            .subscribe(onNext: { [weak self] in
30:                self?.messageLabel.text = "Tap Exit Button!"
31:            })
```

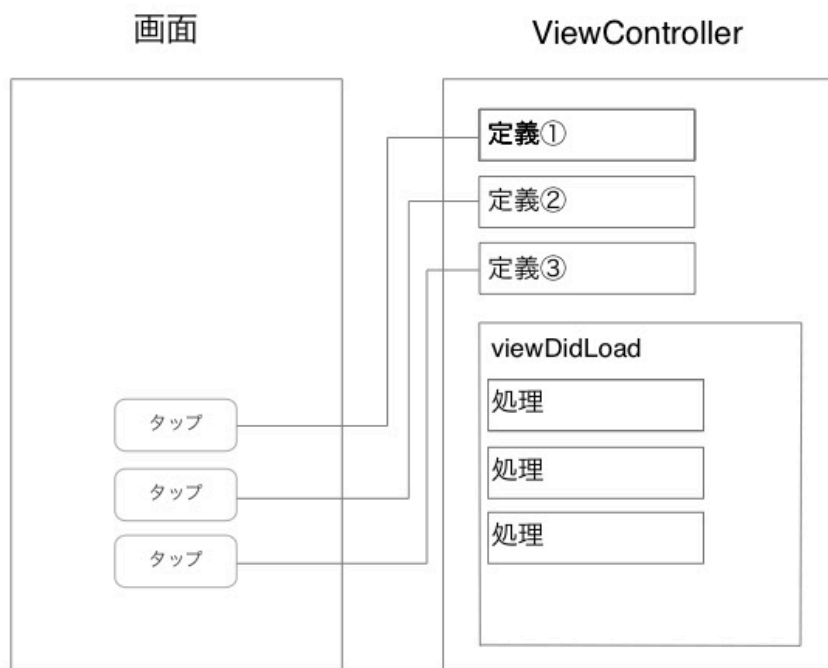


```

32:         .disposed(by: disposeBag)
33:
34:         helpButton.rx.tap
35:             .subscribe(onNext: { [weak self] in
36:                 self?.messageLabel.text = "Tap Help Button!"
37:             })
38:         .disposed(by: disposeBag)
39:     }
40: }

```

図2.7: RxSwiftを用いたコードを拡張（イメージ）



addTargetを利用する場合のコードも見てみましょう。UILabel, UITextFieldを画面にふたつずつ配置し、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 2.8: 画面のイメージ



リスト 2.7: addTarget を用いたコード

```
class ExampleViewController: UIViewController {

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

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

    private let maxNameFieldSize = 10
    private let maxAddressFieldSize = 50

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

    override func viewDidLoad() {
        super.viewDidLoad()
        nameField.addTarget(self, action:
```

```

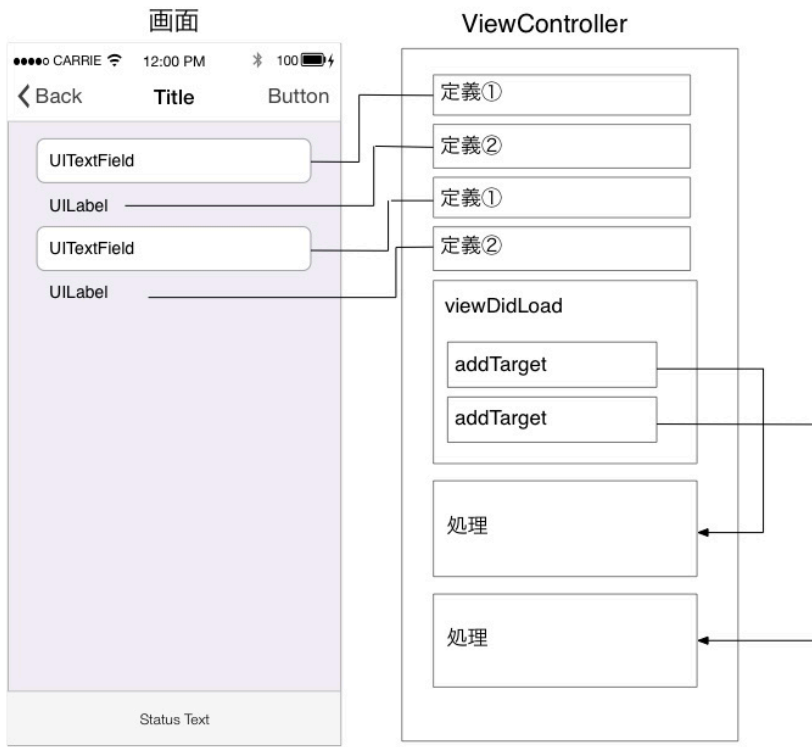
        #selector(nameFieldEditingChanged(sender:)),
        for: .editingChanged)
addressField.addTarget(self, action:
    #selector(addressFieldEditingChanged(sender:)),
    for: .editingChanged)
}

@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)
}
}

```

図 2.9: addTarget を用いたコード (図)



viewDidLoad()内で定義しているのに実際の処理が別関数として離れているので処理の流れがほんの少しだけイメージしにくくなっています。

次にRxSwiftを用いて書いてみます。

リスト 2.8: 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()
```

```

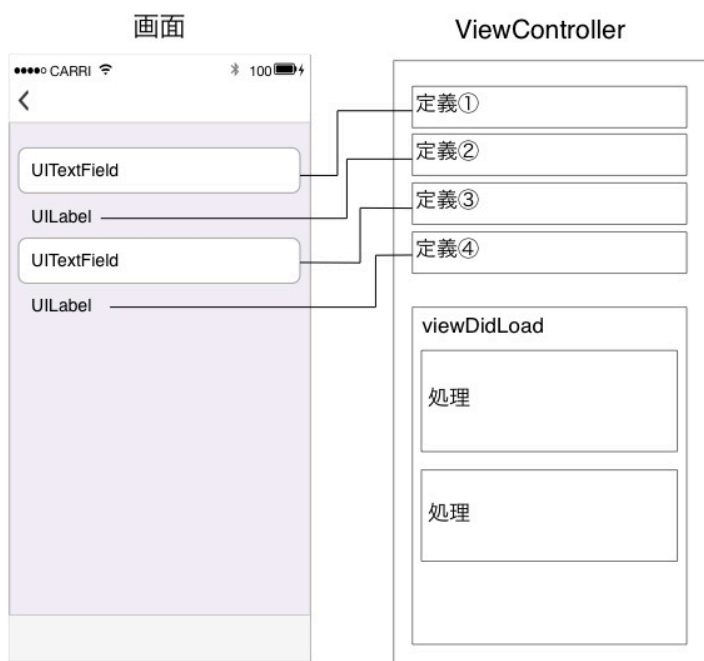
override func viewDidLoad() {
    super.viewDidLoad()

    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
        .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
        .bind(to: addressLabel.rx.text)
        .disposed(by: disposeBag)
    }
}

```

図 2.10: RxSwift version (図)



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、ひとつに集約できたのでちょっと読みやすいですね。

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が、Macでは標準でインストールされているのであまり気にしなくてもよいです。

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. 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
```

次のようにファイルを編集します。

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

CocoaPods は Ruby で構築されており、Podfile は Ruby の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 を指定すると有効になります。

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、一般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、その名前のと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を扱って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: loginButtonのイベント購読

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

それぞれの戻り値の型をコメントで表してみます。

リスト4.2: loginButtonのイベント購読

```
1: loginButton
2:   .rx      // Reactive<UIButton>
3:   .tap     // ControlEvent<Void>
4:   .subscribe(onNext: { /* 処理 */ }) // Disposable
5:   .disposed(by: disposeBag) // Void
```

loginButton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。(仕組みは後述します)

まとめると、次の順で処理を定義しています。

1. ストリームの購読
2. ストリームにイベントが流れてきた時にどうするかを定義
3. クラスが破棄されると同時に購読を破棄させるように設定

4.2 Hello World

RxSwiftでのHelloWorld的なものを書いてみます。

リスト4.3: subject-example

```
1: import UIKit
2: import RxSwift
3: import RxCocoa
4:
5: class ViewController: UIViewController {
6:
7:     private let disposeBag = DisposeBag()
8:
9:     override func viewDidLoad() {
10:         super.viewDidLoad()
11:
12:         let helloWorldSubject = PublishSubject<String>()
13:
14:         helloWorldSubject
15:             .subscribe(onNext: { message in
16:                 print("onNext: \(message)")
17:             }, onCompleted: {
18:                 print("onCompleted")
19:             }, onDisposed: {
20:                 print("onDisposed")
21:             })
22:             .disposed(by: disposeBag)
23:
24:         helloWorldSubject.onNext("Hello World!")
25:         helloWorldSubject.onNext("Hello World!!")
26:         helloWorldSubject.onNext("Hello World!!!")
27:         helloWorldSubject.onCompleted()
28:     }
29: }
```

結果

```
onNext: Hello World!
onNext: Hello World!!
onNext: Hello World!!!
onCompleted
onDisposed
```

※ onDisposed が呼ばれる仕組みは後述します。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ように定義
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. N 回イベントを流す
6. (3.)で定義したクロージャがN回実行される

このような書き方は 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、RxSwift の強みが生かせていないので、実際に ViewController/ViewModel に分けて書いてみましょう。

リスト 4.4: ViewController/ViewModel に分けて書く

```
import RxSwift

class HogeViewController: UIViewController {

    private let disposeBag = DisposeBag()
    private var viewModel: HogeViewModel!

    override func viewDidLoad() {
        super.viewDidLoad()

        viewModel = HogeViewModel()

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

        viewModel.updateItem()
    }
}

class HogeViewModel {

    var helloWorldObservable: Observable<String> {
        return helloWorldSubject.asObservable()
    }
}
```

```

private let helloWorldSubject = PublishSubject<String>()

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

```

出力結果自体はさきほどと同じです。

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。また、ストリームとも表現されたりします。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 4.5: hogeObservable の購読の仕方

```

1: // Observable<Void>
2: hogeObservable
3:   .subscribe(onNext: {
4:       print("next")
5:   }, onError: { error in

```

```

6:         print("error")
7:     }, onComplete: {
8:         print("completed")
9:     }, onDisposed: {
10:        print("disposed")
11:    })

```

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。disposeされるとonDisposed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、その場合は次のように書きます。

リスト 4.6: onNext 以外を省略する

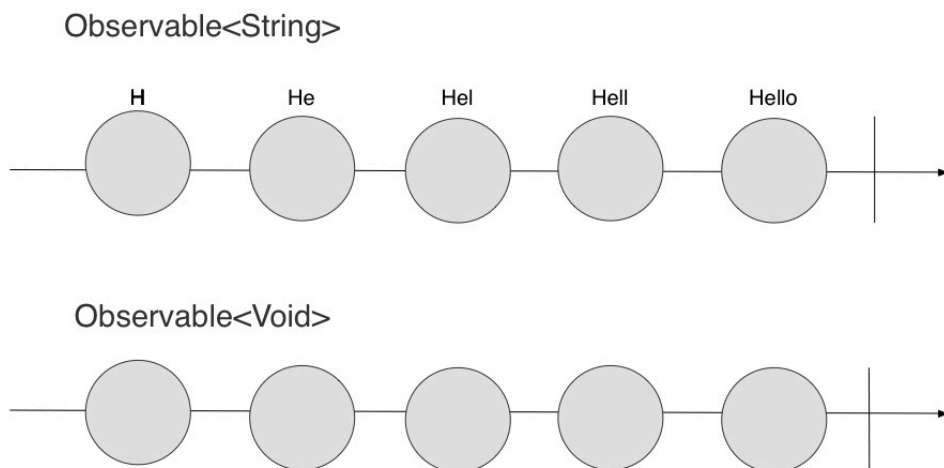
```

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 4.1: マーブルダイアグラム



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「入門」向けではないと考えているので本

書では使用していません。

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 4.7: 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のは覚えておいて下さい。

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、Disposable インスタンスが帰ってきます。

リスト 4.8: Disposable

```
let disposable = hogeButton.rx.tap
    .subscribe(onNext: {
        // ..
    })
```

Disposable 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明示的に購読を破棄できます。今回はその Disposable 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、引数にそのインスタンスを渡します。引数として渡すと自身 (Disposable インスタンス) を DisposeBag の中へ挿入し、DisposeBag インスタンスが解放 (deinit) されたときに自身 (Disposable インスタンス) を含め管理している Disposable を全て自動で dispose 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。

コードで見てみましょう。

リスト 4.9: Dispose のサンプルコード

```

import RxSwift
import RxCocoa
import UIKit

class HogeFooViewController: 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) //Disposable①

        fooButton.rx.tap
            .subscribe(onNext: {
                // ..
            })
            .disposed(by: disposeBag) //Disposable②
    }
}

```

リスト 4.9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扱う時は注意が必要です。DisposeBag の仕組みはそれを保持している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、入門とは脱線してしまうのでここでは省きます。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

大まかな違いは次のテーブル図のとおりです。

表 4.1: Subject と Relay の主な種類

	流せるイベント	バッファ
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 バッファについて

BehaviorSubject/Relayはsubscribe時にひとつ過去のイベントを受け取ることができます。最初にsubscribeするときは宣言時に設定した初期値を受け取ります。

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すが、逆にコードが複雑になってしまうことがあります。

ということかという、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にくくなり、デバッグをするのが大変になってきます。

なので、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です。

次のコードのように定義します。

リスト 4.10: BehaviorRelay のサンプル

```
1: private 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 4.11: 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 4.12: 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して値をセットするの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 4.13: 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: // ①bindを利用
11: nameTextField.rx.text
12:   .bind(to: nameLabel.rx.text)
13:   .disposed(by: disposeBag)
14:
15: // ②subscribeを利用
16: nameTextField.rx.text
17:   .subscribe(onNext: { [weak self] text in
18:     self?.nameLabel.text = text
19:   })
20:   .disposed(by: disposeBag)
```

このコードでは①bindを利用した場合と②subscribeを利用した場合それぞれ定義しました。ふた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) することよりも、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というような仕組みはよくあるパターンのひとつです。

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、ふたつの 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、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（異なる型では不可能）

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4.14: Operator - map 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)
```

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4.15: Operator - map 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)
```

filter

リスト4.16: 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)
```

zip

(例) 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することがあります。

リスト4.17: 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を主に扱っています。

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の使い所として、非同期通信処理で使われることが多いと感じます。試しにsubscribe時にひとつの要素を返すObservableを作成する関数を定義してみましょう

リスト4.18: 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

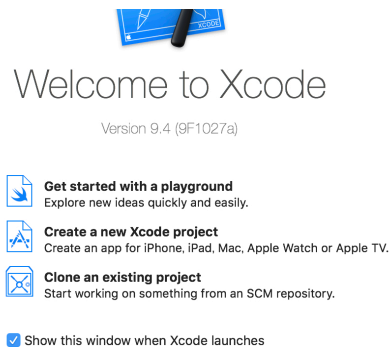
図5.1: 作成するアプリのイメージ



5.1.3 プロジェクトの作成

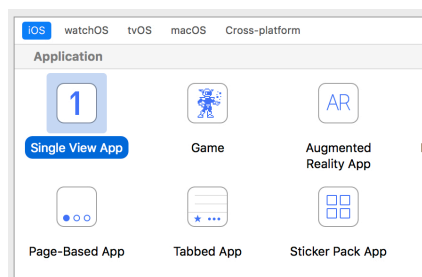
まずはプロジェクトを作成します。ここは特別なことをやっていないのでサクサクといきます。

図5.2: プロジェクトの作成



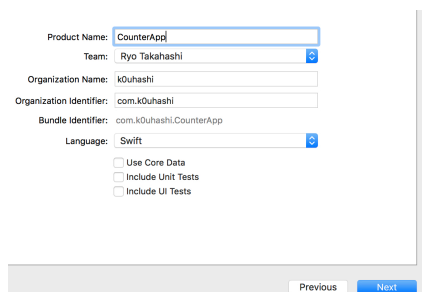
Xcodeを新規で起動して、Create a new Xcode project を選択します。

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れば、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。

```
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 dependenciesDownloading dependenciesInstalling RxCocoa
(4.3.1)Installing RxSwift (4.3.1)Generating Pods projectIntegrating
client project[!] Please close any current Xcode sessions and
use'CounterApp.xcworkspace' for this project from now on.Sending statsPod
installation complete! There are 2 dependenciesfrom the Podfile and
2 total pods installed.[!] Automatically assigning platform 'ios'
with version '11.4' ontarget '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

1. Main.storyboardの削除
2. Info.plistの設定
3. AppDelegateの整理
4. ViewController.xibの作成

順番にやっていきましょう。

1. Main.storyboardの削除

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

2. 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

3. 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の設定が失われました。このままではアプリが正しく起動できないので、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: }

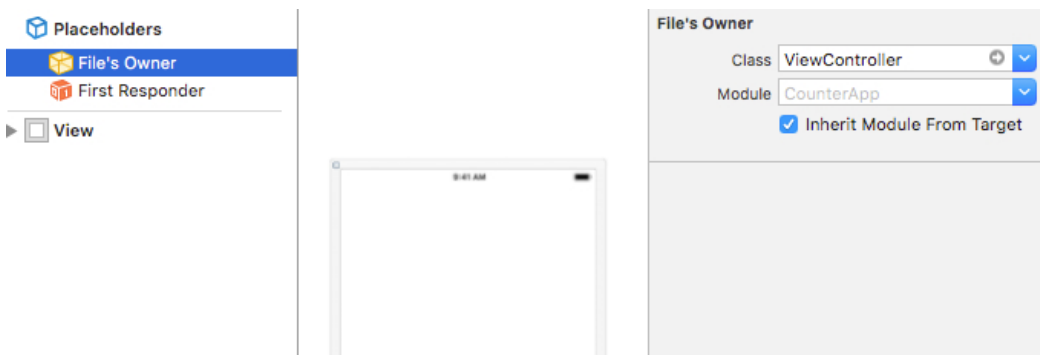
```

4. ViewController.xibの作成

Main.storyboardを削除する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。

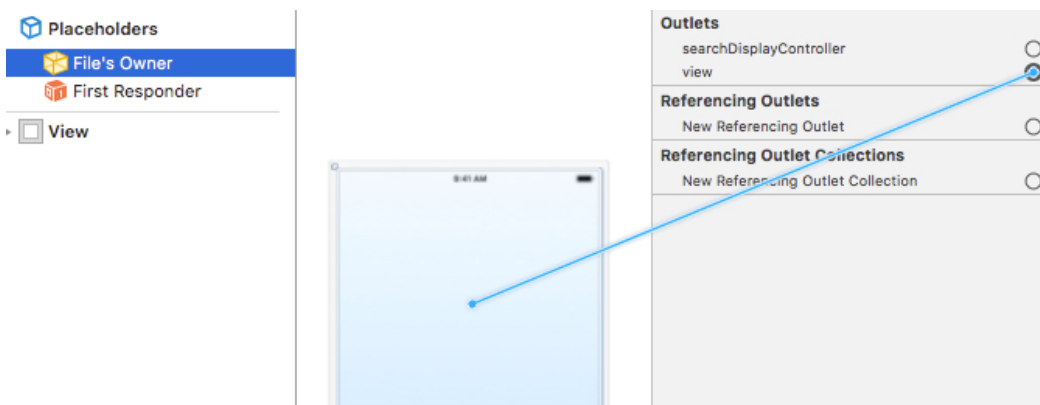
- New File > View > Save As: ViewController.xib > Create
- ViewController.xibを開く
- Placeholders > File's Owner を選択
- Class に ViewController を指定

図 5.5: ViewController.xib の設定 1



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

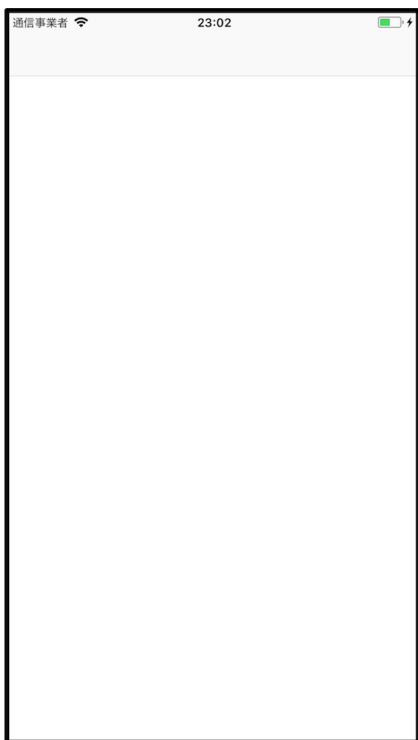
図 5.6: ViewController.xib の設定 2



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

- ・ ViewController.swift の作成
- ・ ViewController.xib の作成
- ・ ViewController.xib の設定
 - Class の指定
 - 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 CounterViewModel {
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: CounterViewModel!  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
        viewModel = CounterViewModel()  
    }  
  
    @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。

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）  
import RxSwift  
import RxCocoa  
  
struct CounterViewModelInput {  
    let countUpButton: Observable<Void>  
    let countDownButton: Observable<Void>  
    let countResetButton: Observable<Void>  
}  
  
protocol CounterViewModelOutput {  
    var counterText: Driver<String?> { get }  
}  
  
protocol CounterViewModelType {  
    var outputs: CounterViewModelOutput? { get }
```

```
func setup(input: CounterViewModelInput)
}
```

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: ViewModelの作成

```
1: import RxSwift
2: import RxCocoa
3:
4:
5: struct CounterViewModelInput {
6:     let countUpButton: Observable<Void>
7:     let countDownButton: Observable<Void>
8:     let countResetButton: Observable<Void>
9: }
10:
11: protocol CounterViewModelOutput {
12:     var counterText: Driver<String?> { get }
13: }
14:
15: protocol CounterViewModelType {
16:     var outputs: CounterViewModelOutput? { get }
17:     func setup(input: CounterViewModelInput)
18: }
19:
20: class CounterRxViewModel: CounterViewModelType {
21:     var outputs: CounterViewModelOutput?
22:
23:     private let countRelay = BehaviorRelay<Int>(value: 0)
24:     private let initialCount = 0
25:     private let disposeBag = DisposeBag()
26:
27:     init() {
28:         self.outputs = self
29:         resetCount()
30:     }
31:
32:     func setup(input: CounterViewModelInput) {
33:         input.countUpButton
34:             .subscribe(onNext: { [weak self] in
```

```

35:         self?.incrementCount()
36:     })
37:     .disposed(by: disposeBag)
38:
39:     input.countDownButton
40:     .subscribe(onNext: { [weak self] in
41:         self?.decrementCount()
42:     })
43:     .disposed(by: disposeBag)
44:
45:     input.countResetButton
46:     .subscribe(onNext: { [weak self] in
47:         self?.resetCount()
48:     })
49:     .disposed(by: disposeBag)
50:
51: }
52:
53: private func incrementCount() {
54:     let count = countRelay.value + 1
55:     countRelay.accept(count)
56: }
57:
58: private func decrementCount() {
59:     let count = countRelay.value - 1
60:     countRelay.accept(count)
61: }
62:
63: private func resetCount() {
64:     countRelay.accept(initialCount)
65: }
66: }
67:
68: extension CounterRxViewModel: CounterViewModelOutput {
69:     var counterText: Driver<String?> {
70:         return countRelay
71:             .map { "Rxパターン:\($0)" }
72:             .asDriver(onErrorJustReturn: nil)
73:     }
74: }

```

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 ViewController: 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:     private var viewModel: CounterRxViewModel!
14:
15:     override func viewDidLoad() {
16:         super.viewDidLoad()
17:         setupViewModel()
18:     }
19:
20:     private func setupViewModel() {
21:         viewModel = CounterRxViewModel()
22:         let input = CounterViewModelInput(
23:             countUpButton: countUpButton.rx.tap.asObservable(),
24:             countDownButton: countDownButton.rx.tap.asObservable(),
25:             countResetButton: countResetButton.rx.tap.asObservable()
26:         )
27:         viewModel.setup(input: input)
28:
29:         viewModel.outputs?.counterText
30:             .drive(countLabel.rx.text)
31:             .disposed(by: disposeBag)
32:     }
33: }
```

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

- ・ 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

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。

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 と処理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！

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

ーカウンターの値を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！

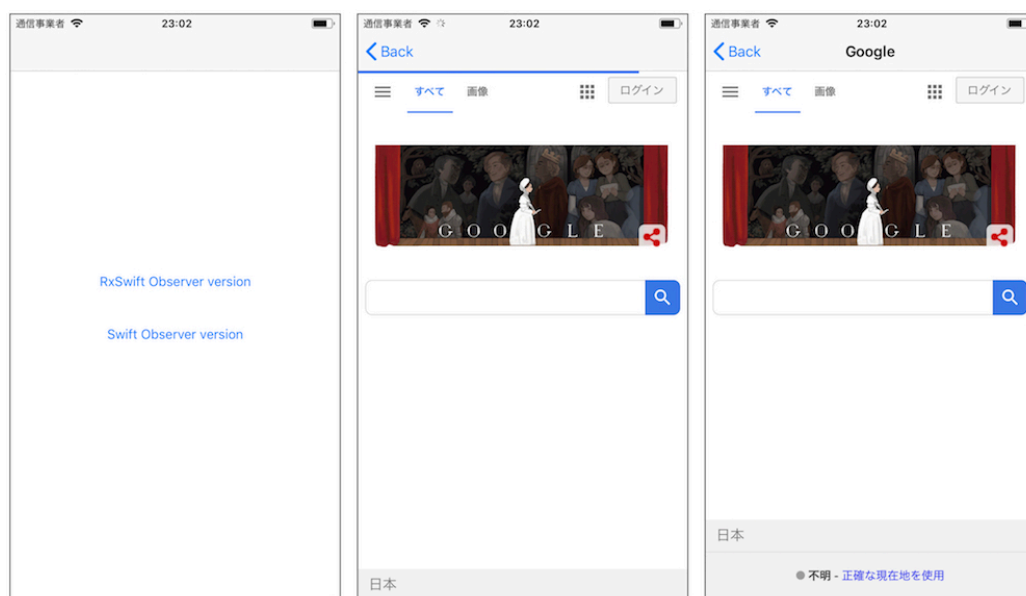
この章ではWKWebViewを使ったアプリをテーマに、KVOの実装パターンをRxSwiftに置き換える方法について学びます。

5.2.1 この章のストーリー

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

5.2.2 イメージ

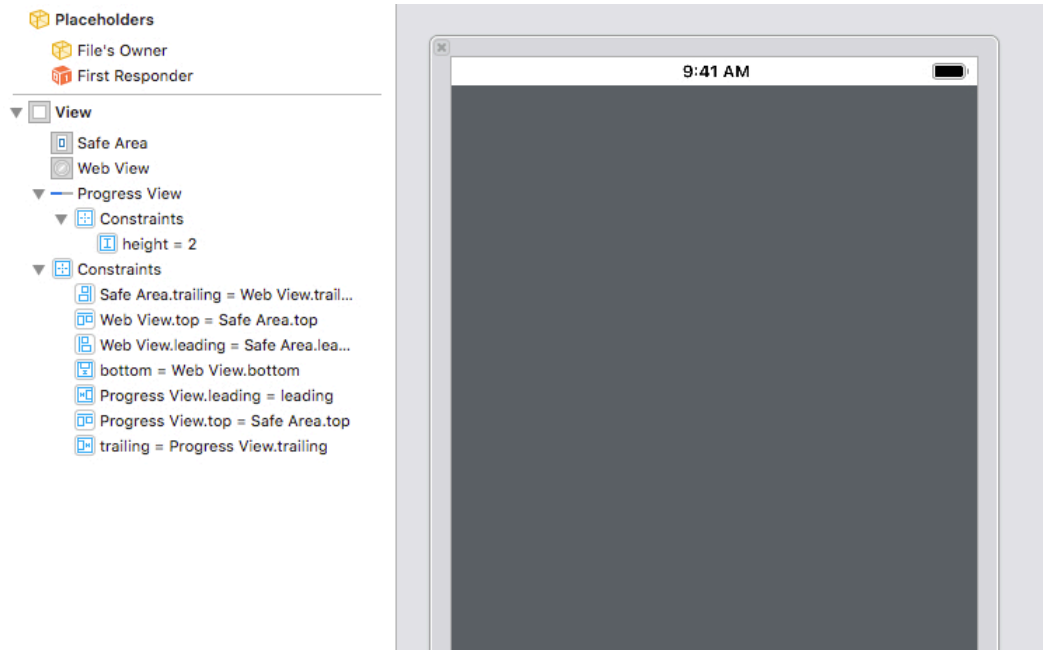
図5.9: アプリのイメージ



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。

サクッといきましょう！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。

図 5.10: UI を配置する



画面ができたら、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の仕組みが使えません。

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 }
            .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 }
            .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) }
            .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がありますが、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KVORespresentable+CoreGraphics.swiftにKVORespresentable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考にとよさそうです。

```
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)
```

このコードは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。(Observableのように扱うことができますが。) `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
```

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 WKWebViewController: 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:
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で使いやすい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いそうですね。

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の仕組みを使っていい感じに差分更新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 initvi 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がひとつのセクションで、SettingsItemがセクション内のセルデータ群です。

次に、ViewModelを作っていきます。

リスト 6.3: SettingsViewModel.swift

```
import RxSwift
import RxCocoa
import RxDataSources

class SettingsViewModel {

    private let items = BehaviorRelay<[SettingsSectionModel]>(value: [])

    var itemsObservable: Observable<[SettingsSectionModel]> {
        return items.asObservable()
    }

    func setup() {
        updateItems()
    }

    private func updateItems() {
        let sections: [SettingsSectionModel] = [
            accountSection(),
            commonSection()
        ]
        items.accept(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)
}
}

```

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れば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SettingsViewController.xibを作成し、画面幅いっぱいに広げたTableViewを設置、TableViewの色を変更後、SettingsViewController.swiftのIBOutletと接続しましょう。Build & Runで実行し、セクションの初めにあった画像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！

図 6.2: その他セクション追加後の画面



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:
            return "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

```

private func updateItems() {
    let sections: [SettingsSectionModel] = [
        accountSection(),

```

```

        commonSection(),
        otherSection()
    ]
    items.accept(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で図6.2の画面になっていたら成功です。

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 を流すことができたり、

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で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、早速やってみましょう！

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、（というかりリモート勉強会）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」はめっちゃめっちゃ活発なので、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！

- ・ 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

高橋 凌 (たかはし りょう)

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。とにかくアプリを作ることが好きで学生時代からWeb・モバイル問わず多種多様なアプリを作り、モバイルアプリでは複数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。

◎本書スタッフ

アートディレクター/装丁：岡田章志+GY

編集協力：飯嶋玲子

デジタル編集：栗原 翔

技術書典シリーズ・刊行によせて

今もっとも注目すべきエンジニアによるアウトプットの形である、技術同人誌の頒布イベント「技術書典」(<https://techbookfest.org/>) では、エンジニアの最新の知見に直接触れることができます。インプレスR&Dでは、この「技術書典」で頒布された技術系同人誌を底本とした『技術書典シリーズ』を刊行します。エンジニアのアウトプットを商業書籍の形で、より多くの読者の皆様に電子書籍と印刷書籍のハイブリッド出版でお届けするものです。エンジニアの“知の結晶”である技術同人誌の世界に、より多くの方が触れていただくきっかけになれば幸いです。

株式会社インプレスR&D

技術書典シリーズ 編集長 山城 敬

●お断り

掲載したURLは2018年12月1日現在のものです。サイトの都合で変更されることがあります。また、電子版ではURLにハイパーリンクを設定していますが、端末やビューアー、リンク先のファイルタイプによっては表示されないことがあります。あらかじめご了承ください。

●本書の内容についてのお問い合わせ先

株式会社インプレスR&D メール窓口

np-info@impress.co.jp

件名に「『本書名』問い合わせ係」と明記してお送りください。

電話やFAX、郵便でのご質問にはお答えできません。返信までには、しばらくお時間をいただく場合があります。

なお、本書の範囲を超えるご質問にはお答えしかねますので、あらかじめご了承ください。

また、本書の内容についてはNextPublishingオフィシャルWebサイトにて情報を公開しております。

<https://nextpublishing.jp/>

技術書典シリーズ

比較して学ぶRxSwift4入門

2019年1月18日 初版発行Ver.1.0（PDF版）

著 者 高橋 凌

編集人 山城 敬

発行人 井芹 昌信

発 行 株式会社インプレスR&D

〒101-0051

東京都千代田区神田神保町一丁目105番地

<https://nextpublishing.jp/>

●本書は著作権法上の保護を受けています。本書の一部あるいは全部について株式会社インプレスR&Dから文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。

©2019 Ryo Takahashi. All rights reserved.

ISBN978-4-8443-9879-0



NextPublishing®

●本書はNextPublishingメソッドによって発行されています。

NextPublishingメソッドは株式会社インプレスR&Dが開発した、電子書籍と印刷書籍を同時発行できるデジタルファースト型の新出版方式です。 <https://nextpublishing.jp/>

NextPublishing Sample