# TAKASHI FUN CLUB Vol.2
# iOS 開発篇

たかしファンクラブ　著

**2017-10-22** 版　　発行

# はじめに

本書は、iOS をテーマにそれぞれが興味のある技術を持ち寄って執筆しました。各技術の基本を抑えつつ、解説を加えながら少し踏み込んだところを集めた一冊です。

（発行代表 @roana0229）

## 本書の内容

本書は下記の内容によって構成されています。

- RxSwift 再入門
- Apple Pencil に対応したアプリを作る
- 動画再生機能を作る

## 免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

# 目次


**はじめに** **2**

本書の内容 . . . . . 2

免責事項 . . . . . 2

**第 1 章 RxSwift 再入門 ～雰囲気で使っている状態から抜け出す～** **5**

1.1 はじめに . . . . . 5

1.2 登場人物とそれぞれの関係 . . . . . 6

1.3 Observable について . . . . . 7

1.4 Operator について . . . . . 7

1.5 Observer について . . . . . 7

1.6 Scheduler . . . . . 8

1.7 Subject . . . . . 12

1.8 Hot と Cold . . . . . 13

1.9 Observable.subscribe() の動作 . . . . . 17

1.10 Operator を自作してみる . . . . . 19

1.11 おわりに . . . . . 20

**第 2 章 Apple Pencil に対応したお絵かきアプリを作ってみよう** **21**

2.1 はじめに . . . . . 21

2.2 Apple Pencil の特徴 . . . . . 22

2.3 Apple Pencil と指での、取得できる情報の違い . . . . . 22

2.4 値の取得 . . . . . 23

2.5 画面に線を引いてみる . . . . . 26

2.6 線を滑らかにする . . . . . 27

2.7 ペンの傾きにより太さを決める . . . . . 28

2.8 ペンの方位角と線を引く方向により線の太さを変える . . . . . 29

2.9 筆圧によって線の太さを変える . . . . . 30

2.10 完成形 . . . . . 30

**第 3 章 動画再生機能を作ってみよう 33**

3.1 はじめに . . . . . 33
3.2 動画再生するには？ . . . . . 33
3.3 シンプルな動画再生機能実装 . . . . . 34
3.4 発展編 -youtube の動画再生- . . . . . 37
3.5 おわりに . . . . . 43

# 第 1 章

# RxSwift 再入門 ～雰囲気で使っている状態から抜け出す～

## 1.1 はじめに

みなさん、RxSwift[*1]使ってますか？ プロミス・データバインディング・イベントバス・リストをあれこれする処理など色々できて良いですよね。ただ、その使い方ってあってるんでしょうか？「全然分からない。俺は雰囲気で RxSwift を使っている。」状態で使っていませんか？ 恥ずかしながら私は完全に雰囲気で使ってしまっていました。

本章は（私含めて）その状態からの脱却を目指す第一弾です。今回は **RxSwift** を使う上での基本を簡単に説明し、どうやって動いているのかを中心に学んでいきます。また、本章の内容は主に RxSwift の github リポジトリにある **Rx.playground** [*2]を参考にしています。

対象とする読者のイメージ

- RxSwift を使ったことがある
- いまいち理解せず雰囲気で使っている

執筆時点の環境

- Xcode: 9.0
- Swift: 3.1
- ReactiveX/RxSwift: master（bd5a9657b9a7cf52f583eecf00dc8b7c0cb9ebaa）

[*1] https://github.com/ReactiveX/RxSwift
[*2] https://github.com/ReactiveX/RxSwift/tree/master/Rx.playground

## 1.2 登場人物とそれぞれの関係

本章には、RxSwift を説明する上で Observable,Observer,Operator,Scheduler,Subject が登場します。

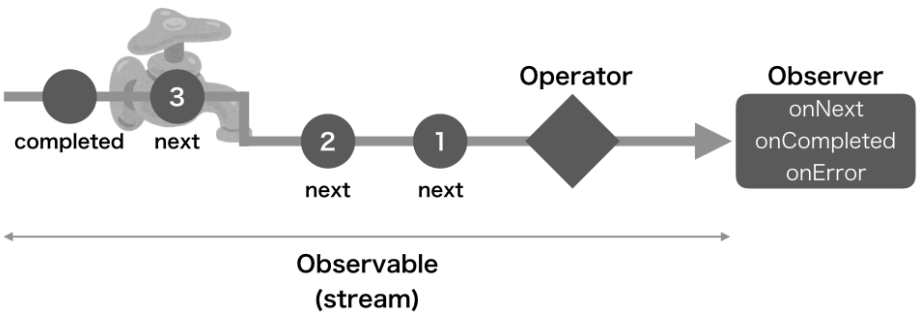
まず、起点となる Observable について、公式サイト[*3]には

> In ReactiveX an observer subscribes to an Observable

「**Observer** は **Observable** を購読します」と書かれています。ReaxtiveX での Observer は、Observer パターンと同じ意味合いで使われ、通知される立場にあります。Observable に流れているイベントが、n 個以上の Operator を通って、Observer へ到達します。RxSwift を使うと何が嬉しいのかというとあらかじめ処理とゴールを決めておき、そこに値が流れてくるように書くことができるため、データフローが明確になります。

イメージの付きやすいように例えると「Observable は蛇口とそこから流れる水」「Operator は水路にあるフィルタ」のような感じです。蛇口から水が流れ、水路でキレイにしたり・混ぜたり様々な処理が行われながら終着点まで流れます。

重要なのは、「蛇口は開かないと水が流れない」という点です。Observable は subscribe が呼び出される（蛇口が開く）ことで、初めてストリームが流れ出します。（例外も存在しますが、これについてはまた後で説明します。）そして、Observable に流れているイベントは Operator を通り、subscribe に引数として渡した Observer の onNext,onError,onComplete に流れ着きます。




▲図 1.1　それぞれの関係と全体のイメージ

[*3] http://reactivex.io/documentation/observable.html

## 1.3 Observable について

Observable はストリームであり、subscribe されることで初めてストリームにイベントが流れます。そのストリームに値を onNext,onError,onCompleted というイベントに包んで、購読している Observer へ通知します。そして onNext は複数回、onEror,onComplete は一度だけ通知することができます。onError もしくは onCompleted を呼んだ後は一切イベントが流れません。

## 1.4 Operator について

Operator は Observable を subscribe し、新しく Observable を生成します。そのため、Operator は通知をする側 (Observable)、購読する側 (Observer) 両方の性質を持ちます。

```
Observable.of(1, 2, 3).map { $0 * $0 }.filter { $0 % 2 == 0 }
```

例えば、上記のコードの **of**, **map**, **filter** が **Operator** です。他にも、Observable を生成する Operator は just,create、変換・結合する Operator は map,flatMap,scan など他にも様々な種類があります。Operator は生成・変換すると新しい Observable が返ってくるため、Operator 同士を繋ぐことができます。

## 1.5 Observer について

登場人物を紹介したとき

> Observable は subscribe が呼び出される（蛇口が開く）ことで、初めてイベントが流れ出します。

このよう説明した通り、先程 Operator について紹介したコードだけでは何も起こりません。下記のように、subscribe することで初めてストリームが流れ出します。

```
Observable.of(1, 2, 3).map { $0 * $0 }.subscribe(onNext: { print($0) })
```

このコードでは、1,2,3 という値が onNext というイベントに包まれ、それぞれ 1,4,9 と map が変換し、Observer の onNext へ通知されます。

### Observer と RxSwift3.x で追加された派生系の種類

RxSwift3.x 系からは Observer の他に、**Single**,**Completable**,**Maybe** が追加され、通知される **onXXX** が違います。Observer を含めて表にするとこのようになります。

| Observer の種類 | 動作 |
|---|---|
| Obsever | onNext(value) が 1 回以上、onCompleted,onError(error) どちらか 1 回 |
| Single | onSuccess(value),onError(error) のどちらが 1 回 |
| Compaletable | onCompleted,onError(error) がどちらか 1 回 |
| Maybe | onSuccess(value),onCompleted,onError(error) のどれかが 1 回 |

## 1.6 Scheduler

ここで、実際に今まで説明した Observable,Operator,Observer を使う時のことを考えてみましょう。例えば、通信処理で下記のような処理をするとします。

1. ツイートの ID を指定して、それを取得するリクエストを投げる
2. レスポンスを受け取る
3. 受け取った JSONString を Tweet モデルに変換する
4. できがったモデルを UI へ反映する

この時、1~3 はバックグラウンドで、4 はメインスレッドで処理をするのが適切です。そのスレッドの制御をする役割を持つのが **Scheduler** です。**Scheduler** は iOS の仕組みとして存在する GCD（Grand Central Dispatch）を利用することでスレッド制御を実現しています。その Scheduler を指定するためには **observeOn** , **subscribeOn** という Operator を利用します。

```
TweetService.requestTweet(by: ツイートの ID) // Observable<String>を返す  // 1
            .map { JSONString を Tweet モデルへ変換 ($0) }                  // 2,3
            .subscribeOn(background-scheduler)
            .observeOn(main-scheduler)
            .subscribe(onNext: { label.text = $0. ツイートのテキスト変数 }) // 4
```

これは実際に Scheduler を指定した 1~4 の流れのコードの例です。これで 1~3 はバックグラウンドスレッド、4 はメインスレッドで動作します。きちんと observeOn,subscribeOn の動作を理解をするためには、 **Observable.subscribe()** の動作を理解する必要があり、後で詳しくみていきます。この章を見る上では「observeOn は下方向」に適応し、「subscribeOn は上方向」に適応されるぐらいの認識で大丈夫です。

## Scheduler の種類

| Scheduler | 動作 |
|---|---|
| MainScheduler | メインスレッドで動きます（observeOn に最適化） |
| ConcurrentMainScheduler | メインスレッドで動きます（subscribeOn に最適化） |
| CurrentThreadScheduler | 現在のスレッドで動きます |
| SerialDispatchQueueScheduler | 直列な Queue 上でバックグラウンドで動きます |
| ConcurrentDispatchQueueScheduler | 並列な Queue 上でバックグラウンドで動きます |

それぞれ DispatchQueue を持つ仕組みになっていて、イニシャライザには引数として DispatchQoS を渡すものと DispatchQueue を渡すものがあります。DispatchQoS を引数として渡すイニシャライザは iOS8 から追加されていて、独自のラベルの設定などの必要がなければ DispatchQoS を渡す方が良いです。

## Scheduler で注意すべきこと

まず、Concurrent（並列）な Scheduler で処理されていても、1 つの Observable に流れる値は順序が保証されています。そのため、このコードのようにランダムなスリープを挟んでも実行すると 1,2,3 と値は順に出力されています。

```
let observeOnScheduler = ConcurrentDispatchQueueScheduler(qos: .default)

var count = 0
Observable.from(1...3)
    .observeOn(observeOnScheduler)
    .do(onNext: { i in
        let time = arc4random_uniform(3) + 1 // 1~3 秒のスリープ
        print("observable sleep \(time)")
        sleep(time)
        count += 1
        print("observable \(i): \(count)")
    })
    .subscribe()

// 出力
A observable sleep 2
A observable 1: 1
A observable sleep 1
A observable 2: 2
A observable sleep 1
A observable 3: 3
```

では、同じ並列な Scheduer で 2 つの Observable を subscribe したらどうなるでしょうか？

```
let observeOnScheduler = ConcurrentDispatchQueueScheduler(qos: .default)

var count = 0
Observable.from(1...3)
    .observeOn(observeOnScheduler)
    .do(onNext: { i in
        let time = arc4random_uniform(3) + 1 // 1~3 秒のスリープ
        print("A observable sleep \(time)")
        sleep(time)
        count += 1
        print("A observable \(i): \(count)")
    })
    .subscribe()

Observable.from(4...6)
    .observeOn(observeOnScheduler)
    .do(onNext: { i in
        let time = arc4random_uniform(3) + 1 // 1~3 秒のスリープ
        print("B observable sleep \(time)")
        sleep(time)
        count += 1
        print("B observable \(i): \(count)")
    })
    .subscribe()

// 出力
B observable sleep 2
A observable sleep 1
A observable 1: 1
A observable sleep 1
B observable 4: 2
B observable sleep 3
A observable 2: 3
A observable sleep 3
B observable 5: 4
B observable sleep 1
A observable 3: 5
B observable 6: 6
```

Concurrent（並列）な Scheduler で処理しているため、共有された変数 count と Observable に流れる値が同じにならず、並列で動いていることがわかります。では、Serial（直列）な Scheduler に変えてみましょう。

```
let observeOnScheduler = SerialDispatchQueueScheduler(qos: .default)

var count = 0
Observable.from(1...3)
    .observeOn(observeOnScheduler)
    .do(onNext: { i in
        let time = arc4random_uniform(3) + 1
        print("A observable sleep \(time)")
        sleep(time)
        count += 1
        print("A observable \(i): \(count)")
```

```
    })
    .subscribe()

Observable.from(4...6)
    .observeOn(observeOnScheduler)
    .do(onNext: { i in
        let time = arc4random_uniform(3) + 1
        print("B observable sleep \(time)")
        sleep(time)
        count += 1
        print("B observable \(i): \(count)")
    })
    .subscribe()

// 出力
A observable sleep 2
A observable 1: 1
A observable sleep 1
A observable 2: 2
A observable sleep 2
A observable 3: 3
B observable sleep 3
B observable 4: 4
B observable sleep 1
B observable 5: 5
B observable sleep 1
B observable 6: 6
```

それぞれランダムにスリープがかかっていることにも関わらず、共有された変数 count と Observable に流れる値が同じになっていて、直列に動いていることがわかります。

さて、この Scheduler の指定で大事なのが Observable を subscribe した時、observeOn,subscribeOn で Scheduler を指定していない場合は **CurrentThreadScheduler** (今いるスレッドで実行する Scheduler) で実行されるということです。つまり、何も考えずにメインスレッドで動いている処理中に subscribe してしまうと、重い処理をメインスレッドで処理してしまいます。

普段コードを書いている時に、これまでに示してきたように Observable をそれぞれ生成して subscribe することなんてないからあまり関係なさそうだと思う人がいるかもしれませんが、実は普段からよく発生している処理です。それがどんなときかというと、zip,merge など Observable を結合して処理を行う場合です。

```
let observeOnScheduler =
  // ConcurrentDispatchQueueScheduler or SerialDispatchQueueScheduler

let aObservable = Observable.from(1...3)
    .observeOn(observeOnScheduler)
    .do(onNext: { i in
        sleep(UInt32(3))
    })
```

```
let bObservable = Observable.from(4...6)
    .observeOn(observeOnScheduler)
    .do(onNext: { i in
        sleep(UInt32(3))
    })

Observable.zip(aObservable, bObservable, resultSelector: { e1, e2 in
    print("e1: \(e1), e2: \(e2)")
}).subscribe()
```

このコードは、今までに説明で利用してきた Observable のスリープ処理だけを残し変数化して zip で結合したものを subscribe しています。ConcurrentDispatchQueueScheduler を指定した場合は、並列で走るため「3 秒スリープ× 3 回分」の時間で完了します。しかし、SerialDispatchQueueScheduler を指定した場合は、直列で走るため「3 秒スリープ× 3 回× 2 つ Observable 分」の時間がかかってしまいます。通信処理の待ち合わせなどで zip,merge などを使っていて、なぜか遅いなと思ったら Scheduler を疑ってみると良いかもしれません。

## 1.7 Subject

登場人物を紹介したとき

> Observable は subscribe が呼び出される（蛇口が開く）ことで、初めて流れ出します。（例外も存在しますが、これについてはまた後で説明します。）

この時説明した例外が、この章で説明する **Subject** です。**Subject** も Observable です。しかし、初めから蛇口が開いている状態、つまり初めからストリームが流れています。というと語弊があるかもしれませんが、subscribe されていなくてもイベントを流すことができるということです。

```
let subject = PublishSubject<String>()
subject.onNext("1") // subscribe されていなくてもイベントを流せる
subject.subscribe(onNext: { print("subscribe: \($0)") })
subject.onNext("2")
subject.onCompleted()
subject.onNext("3") // Observable と同じく既に onCompleted が呼ばれているのでイベント
が流れない

出力
A subscribe: 2
```

基本は Observable と同じ性質であるため、onNext,onCompleted,onError を呼ぶことができます。また、onCompleted,onError のどちらかが呼ばれてしまうと、その後は

イベントを流すことができません。そのため、上記のコードでは subscribe した後から onCompleted が呼ばれる前までの onNext のみの通知を Observer が受け取っています。

### Subject の種類

| Subject | 動作 |
|---|---|
| PublishSubject | キャッシュせず、来たイベントをそのまま通知する |
| ReplaySubject | 指定した値だけキャッシュを持ち、<br>subscribe 時に直近のキャッシュしたものを通知する |
| BehaviorSubject | 初期値を持つことができ、1 つだけキャッシュを持ち、<br>subscribe 時に直近のキャッシュしたものを通知する |
| Variable | 変数のように扱うことができ、<br>value プロパティを変更すると onNext を通知する ※ RxSwift 独自 |

## 1.8 Hot と Cold

基本的に Observable は、subscribe するまでストリームが流れません。その例外として Subject を利用した Observable は、常にストリームが流れています。これを **ReactiveX** の概念では **Hot,Cold** と言います。

```
let hotObservable = Observable
    .from(1...3)
    .publish()

hotObservable.subscribe(onNext: { i in
    print("subscribe onNext \(i)")
})

hotObservable.connect()
```

上記の **publish** という Operator は **Cold->Hot** に変換します。Cold->Hot の変換は、multicast というメソッドで「Cold な Observable を subscribe 時に Subject で包む」仕組みです。また、Hot な Observable は自身が参照している（1 つ前にチェーンされている）Observable の subscribe を呼び出す性質を持ちます。

しかし、上記のコードで hotObservable.subscribe() した時点ではイベントが流れていません。publish で返ってくるのは ConnectableObservable という型で、connect されることで初めてストリームが流れ出し、その時に subscribe されているものに対してイベントが流れます。

```
let hotObservable = Observable
        .from(1...3)
        .do(onNext: { print("onNext: \($0)") })
        .publish()

hotObservable.connect()

出力
onNext: 1
onNext: 2
onNext: 3
```

また、subscribe されなくとも値は流れることは上記のコードの出力見ればわかるでしょう。

Hot,Cold の違いは、他にもあります。例えば、Hot な Observable は複数の subscribe に対してストリームを共有しています。

```
var count = 0
let hotObservable = Observable
    .from(1...3)
    .do(onNext: { i in
        count += 1
        print("doOnNext \(i), count:\(count)")
    })
    .publish()

hotObservable.subscribe(onNext: { i in
    print("1 subscribe onNext \(i)")
})
hotObservable.subscribe(onNext: { i in
    print("2 subscribe onNext \(i)")
})

hotObservable.connect()


出力
doOnNext 1, count:1
1 subscribe onNext 1
2 subscribe onNext 1
doOnNext 2, count:2
1 subscribe onNext 2
2 subscribe onNext 2
doOnNext 3, count:3
1 subscribe onNext 3
2 subscribe onNext 3
```

```
var count = 0
let coldObservable = Observable
    .from(1...3)
    .do(onNext: { i in
```

```
        count += 1
        print("doOnNext \(i), count:\(count)")
    })

coldObservable.subscribe(onNext: { i in
    print("1 subscribe onNext \(i)")
})
coldObservable.subscribe(onNext: { i in
    print("2 subscribe onNext \(i)")
})


出力
doOnNext 1, count:1
1 subscribe onNext 1
doOnNext 2, count:2
1 subscribe onNext 2
doOnNext 3, count:3
1 subscribe onNext 3
doOnNext 1, count:4
2 subscribe onNext 1
doOnNext 2, count:5
2 subscribe onNext 2
doOnNext 3, count:6
2 subscribe onNext 3
```

ストリームを共有しているため、上記の出力のように複数の subscribe があった場合に 1,2,3 がそれぞれイベントが共有され、2 つの subscribe に対して「同時」に流れています。Cold の場合はイベントが複製され、それぞれの subscribe に対して「別々」に流れています。

### Cold -> Hot 変換で注意すべきこと

Cold->Hot 変換した ConnectableObservable では注意しないといけないことがあります。それは、connect した Observable を dispose(unSubscribe) しないと開放されないということです。しかし、正しいタイミングを気にしながら dispose するのはとても難しいことです。そこで、ConnectableObservable は **refCount()** というメソッドを持っています。

```
let hotObservable = Observable
    .from(1...3)
    .publish()
    .refCount()

hotObservable.subscribe(onNext: { i in
    print("subscribe onNext \(i)")
})
```

```
出力
subscribe onNext 1
subscribe onNext 2
subscribe onNext 3
```

**refCount()** を呼び出すと、内部で connect が呼び出しています。そして、全ての subscribe が dispose されると、自動で Observable を dispose してくれます。これは iOS で使われているメモリ管理の ARC と同じ仕組みで「subscribe 毎にカウントアップ」「dispose 毎にカウントダウン」されます。そしてカウントが 0 になった時に dispose されるため、dispose のタイミングを意識しなくて良くなります。

> publish で返ってくるのは ConnectableObservable という型で、connect されることで初めてストリームが流れ出し、その時に subscribe されているものに対してイベントが流れます。

先程はこのように説明しました。上記のコードは refCount 内で connect されていて、その時点では subscribe されていません。しかし、その後に呼び出した subscribe にイベントが流れています。これは refCount() による作用で、refCount() を呼び出すと connect されているのにも関わらず、それ以降 subscribe されるまでイベントが流れないようになります。

では、次に複数回 subscribe してみましょう。

```
let hotObservable = Observable
    .from(1...3)
    .publish()
    .refCount()

hotObservable.subscribe(onNext: { i in
    print("1 subscribe onNext \(i)")
})
hotObservable.subscribe(onNext: { i in
    print("2 subscribe onNext \(i)")
})

出力
1 subscribe onNext 1
1 subscribe onNext 2
1 subscribe onNext 3
```

2 つ subscribe しているはずが、最初の subscribe にしかイベントが流れていません。これはストリームを共有しているため、初めに subscribe した時点でそれに対してイベントが流れ、2 つ目の subscribe の時点では、既に流れたイベントは終えているため流れないということです。

```
var count = 0
let hotObservable = Observable
    .from(1...3)
    .do(onNext: { i in
        count += 1
        print("doOnNext \(i), count:\(count)")
    })
    .replay(3)
    .refCount()

hotObservable.subscribe(onNext: { i in
    print("1 subscribe onNext \(i)")
})

hotObservable.subscribe(onNext: { i in
    print("2 subscribe onNext \(i)")
})

出力
doOnNext 1, count:1
1 subscribe onNext 1
doOnNext 2, count:2
1 subscribe onNext 2
doOnNext 3, count:3
1 subscribe onNext 3
2 subscribe onNext 1
2 subscribe onNext 2
2 subscribe onNext 3
```

これは reply という Cold->Hot 変換の Operator を使ったコードです。publish は PublishSubject を利用するのに対して、replay は ReplaySubject を利用しているため、上記の出力のように 2 つ目の subscribe にもキャッシュしてあるイベントが流れます。

また、 **.publish().refCount()** は **share()** 、 **.replay().refCount()** は **shareReplay()** というエイリアス的なメソッドが用意されています。ここまでは動作を理解するためにそれぞれを呼んでいましたが、動作を理解した上では share(),shareReplay() を使った方が良いでしょう。

## 1.9 Observable.subscribe() の動作

ではここで振り返りながら、Observable.subscribe() するとどのように動作するのか、もう少し細かく見ていきましょう。

```
TweetService.requestTweet(by: ツイートの ID) // Observable<String>を返す
            .map { JSONString を Tweet モデルへ変換 ($0) }
            .subscribeOn(concurrentDispatchQueueScheduler)
            .observeOn(mainScheduler)
            .subscribe(onNext: { label.text = $0. ツイートのテキスト変数 })
```

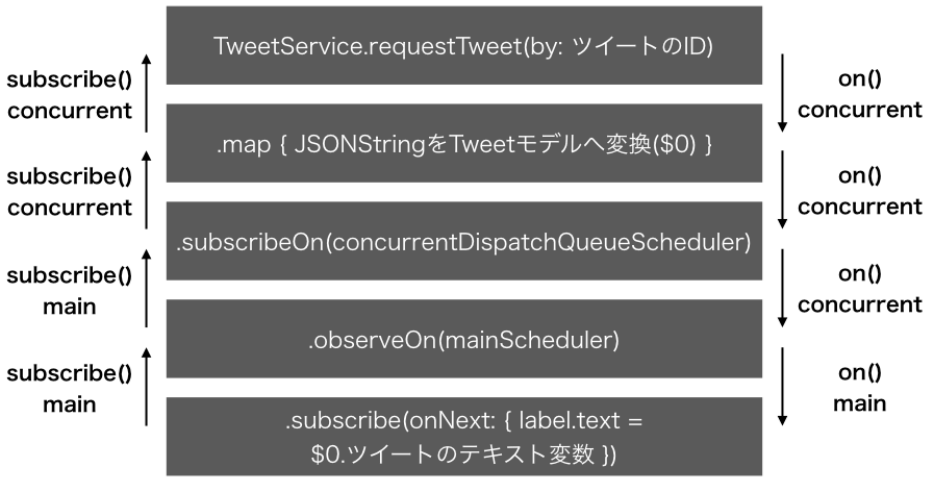
これは Scheduler の章で示した、ツイートを取得するコードです。このコードを実行すると、バックグラウンドでツイートを取得し、文字列からモデルへ変換します。そして、メインスレッドでそのモデルを使い描画します。

まず当然ですが、実行するためには最低限必要な Observable と Observer が出てきます。Observable は通知する、Observer は購読する立場にあります。この「通知する」というのは、 **on(event)** メソッドを呼び出すということです。また、「購読する」というのは、 **subscribe()** メソッドを呼び出すということです。そして、Cold な Obervable は購読しなければ通知することはありません。

何が言いたいかというと、一連の Operator が繋がれた最後に subscribe すると、繋がれた Operator を最初に Observable を生成したところまで購読 (subscribe) していきます。これがコードでは上るようなイメージになります。

そして、頂点に着くと subscribe を呼び出した Observer に対して on(event) によってイベントを通知していきます。この時初めの on(event) が呼ばれるスレッドは、最後に subscribe() されたスレッドになります。これがコード上では下るようなイメージになります。

その中で、subscribeOn は subscribe() の呼び出しに Scheduler を適応し、observeOn は on(event) の呼び出しに Scheduler を適応します。そのため、subscribeOn は上方向に適応され、observeOn は下方向に適応されます。




▲図 1.2　Observable.subscribe() の動作の流れ

また、Observable に流れるイベントが複数ある場合はそれぞれが繋がれた Operator を 1 つずつ通り、最後の Observer まで到達します。繋がれた Operator で全てのイベントが処理されてから次の Operator へ行くのではないということです。

冒頭でも言いましたが、あらかじめ Operator で処理を Obeserber でゴールを決めておき、そこに値がイベントに包まれて流れてくることができるため、データフローが明確になります。

## 1.10 Operator を自作してみる

ここでもう少し理解を深めるために、今までに使ってきた Operator を作ってみましょう。Operator は ObservableType の extension として実装されています。ちなみに Observable を最初に生成する of,from,create などのメソッドは Observable の extension として実装されています。

```
import RxSwift

public struct Logger {
    static func debug(_ string: String) {
        print(string)
    }
}

extension ObservableType {
    public func myDebug(identifier: String) -> Observable<Self.E> {
        return Observable.create { observer in
            Logger.debug("subscribed \(identifier)")
            let subscription = self.subscribe { e in
                Logger.debug("event \(identifier) \(e)")
                switch e {
                case .next(let value):
                    observer.on(.next(value))
                case .error(let error):
                    observer.on(.error(error))
                case .completed:
                    observer.on(.completed)
                }
            }
            return Disposables.create {
                Logger.debug("disposing \(identifier)")
                subscription.dispose()
            }
        }
    }
}

Observable.from(1...3).myDebug(identifier: "from").subscribe()


出力
```

```
subscribed from
event from  next(1)
event from  next(2)
event from  next(3)
event from  completed
disposing from
```

上記のコードは既存で実装されている debug という Operator の動作を模して、独自ロガーで出力するようにしたものです。

既存の Operator を見ると **Debug(Observable)**,**DebugSink(Observer)** というクラスで実装されていますが、create を利用することで簡単に独自の Operator を作ることができます。create に渡すクロージャが subscribe された時の動作です。subscribe されるとログを流し自身が参照している（1 つ前にチェーンされている）Observable を subscribe します。その subscribe 時に subscribe されたこと、イベントが通知された時にイベントの内容、dispose された時に dispose されたことをそれぞれログを流しています。

## 1.11 おわりに

どうでしたか？「全然分からない。俺は雰囲気で Observable を使っている。」状態からは抜け出せたでしょうか？

実は RxSwift について書き始めた時は「ユースケースで学ぶ RxSwift」というタイトルで進めていたのですが、まず基本を理解しないといけないなと思って書いているとあっという間にページが埋まってしまいました。また訪れるであろう技術書典で、第二弾として書きたいなと思っています。

もし本章の内容に間違い・紛らわしい内容があれば Twitter: @roana0229 までメンションしていただけると嬉しいです。感想もお待ちしております。

# 第 2 章

# Apple Pencil に対応したお絵かきアプリを作ってみよう

## 2.1　はじめに

　こんにちは。shinfkd です。
普段はとある目黒の Web 企業で、人事としてエンジニア採用担当をやっています。
採用担当であるがゆえに、普段はほとんどコードを書く仕事をしておりません。
が、新卒採用イベントなどにはよく行きます。
人事が新卒採用イベントに何を持っていくのか…?
そう、iPad Pro です。
iPad Pro は、学生に向けて会社紹介のプレゼン資料を見せたり、オフィスの様子の写真を見せたり、メモを取るのに最適です。
　すみません、、、嘘をつきました。
プレゼン資料を見せたり、オフィスの様子を見せるのに、iPad Pro はとても便利なのですが、さすがにメモを取るのには苦労をしてしまいます。
PC でメモをとるのとは違い、タッチパネルで文字入力をするのには、どうしても時間がかかってしまうのです。
でも、待ってください。iPad Pro にはアレがあります。
そう、Apple Pencil です。
　そう思った私は、思い立ったが吉日とばかりに、Apple Pencil をポチりました。
前置きが長くなりましたが、今回はせっかく Apple Pencil を手に入れたので、自分でも Apple Pencil を用いたアプリ開発をしてみようと思った次第です。

## 2.2 Apple Pencil の特徴

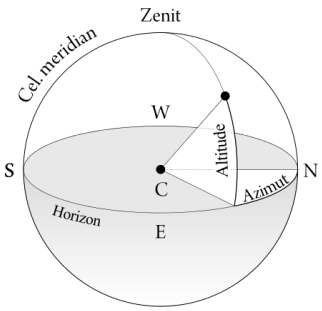
Apple Pencil は 2015 年の 11 月に発売され、2017 年 9 月現在のところ、対応しているのは iPad Pro のみとなっています。
いくつかの特徴がありますが、代表的なものを挙げると下記のようになります。

- 指では取得できない、ペンの傾きと、方向を取得できる
- 秒間 240 回に及ぶスキャン
  - 指の場合は秒間 120 回。ディスプレイのリフレッシュレートは、10.5 インチモデル及び、第 2 世代 12.9 インチモデルは 120Hz。それ以前は 60Hz。
- 筆圧を取得できる
  - iPhone 6s 以降に搭載されている 3D Touch とは異なり、Apple Pencil 側で筆圧を計測している

中でも、一番の特徴となる「指との違い」について見ていきましょう。

## 2.3 Apple Pencil と指での、取得できる情報の違い

上述もしたとおり、Apple Pencil と指とでは、その取得できる情報に違いがあります。具体的には、Apple Pencil を利用した場合、`Altitude(高度)`と `Azimuth(方位角)`、それから `Force(筆圧)`の 3 つが取得できるようになります。




▲図 2.1　Altitude&Azimuth

Altitude は、高さを意味します。
図 3.1 を見ていただくとわかるように、C をペン先と見た場合に、Altitude はペンの傾きを意味しています。

また Azimuth は、方位角を意味します。
同じく図で見た場合に、Azimuth はペンの向きを指していることがわかります。

## 2.4 値の取得

上述した情報を取得するには、指で画面をタッチした場合と同じく、UITouch クラスを用います。
指と Apple Pencil を用いた場合の違いは、UITouchType の違いとなります。

### Apple Pencil を識別する (UITouchType)

UITouchType に、iOS9.1 から新たな種別である Stylus が追加されました。
これにより、Apple Pencil の場合のみを識別して値を取得することができます。

▼リスト 2.1　UITouchType

```
@available(iOS 9.0, *)
public enum UITouchType : Int {

    case direct // A direct touch from a finger (on a screen)

    case indirect // An indirect touch (not a screen)

    @available(iOS 9.1, *)
    case stylus // A touch from a stylus
}
```

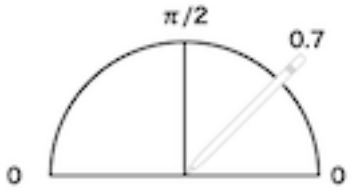
- **direct**: 指でタッチした場合
- **indirect**: 画面以外でデバイスを用いてタッチした場合
- **stylus**: Apple Pencil を用いた場合

indirect は使用シーンがわかりづらいですが、英語版ドキュメントに下記のようにあったため、Apple TV などのリモコンを用いた際に使用されるようです (手元に実機がないので未検証)。

> Indirect touches are generated by touch input devices that are separate from the screen. For example, the trackpad of an Apple TV remote generates indirect touches.
> https://developer.apple.com/documentation/uikit/uitouchtype/uitouchtypeindirect

## Altitude を取得する

Altitude は、UITouch.altitudeAngleより取得することができます。Altitude の値はラジアンで表現され、ペンが iPad のスクリーン面に対して水平になったときに値は 0 となり、スクリーンに対して直角のときに π/2 となります。




▲図 2.2　AltitudeAngle

▼リスト 2.2　UITouch.altitudeAngle の取得

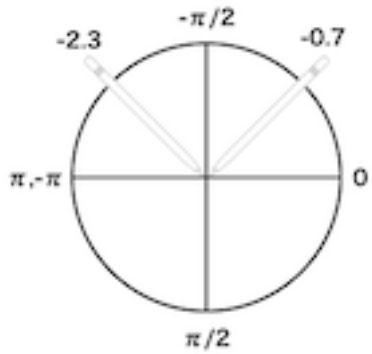
```
override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
    guard let altitudeAngle = touches.first?.altitudeAngle else { return }
    print("altitudeAngle = \(altitudeAngle)") // altitudeAngle = 0.380229310193334
}
```

## Azimuth を取得する

Azimuth の値には、「方位角」で取得する方法と「ベクトル」で取得する方法の 2 通りが存在します。どちらを利用するかは、シーンによって異なると思いますので、お好きな方をお選びください。

### 方位角で取得する

Azimuth を Altitude と同じくラジアンで取得するには、UITouch.azimuthAngleInViewを使用します。
方位角は、画面右を 0 とし、そこから時計回りにペンを向けた場合値が増えていき、6 時の方向でπ/2、9 時でπとなります。反時計回りの場合は、マイナス値となり、0 時で -π/2、9 時でπとなります。



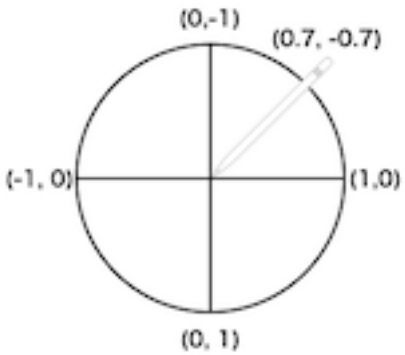

▲図 2.3　AzimuthAngle

▼リスト 2.3　UITouch.azimuthAngleInView の取得

```
override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
    guard let azimuthAngle = touches.first?.azimuthAngle(in: self) else { return }
    print("azimuthAngle = \(azimuthAngle)")
    // azimuthAngle = -1.98289489746094
}
```

ベクトルで取得する

Azimuth をベクトル値で取得するには、`UITouch.azimuthUnitVector`を使用します。




▲図 2.4　AzimuthVector

▼リスト 2.4　UITouch.azimuthUnitVector の取得

```
override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
    guard let azimuthVector =
        touches.first?.azimuthUnitVector(in: self) else { return }
    print("azimuthVector = \(azimuthVector)")
    // azimuthVector = CGVector(dx: -0.400533091690527, dy: -0.916282294088906)
}
```

### 筆圧を取得する

筆圧は、Apple Pencil をスクリーンに押し付ける強さによって上下し、取得するには `UITouch.force` を使用します。
この値は、iPad Pro の場合、Apple Pencil を使わずに指で押した場合は常に 0 になり取得できませんが、3D Touch を搭載している iPhone の場合は取得可能になります。

▼リスト 2.5　UITouch.force の取得

```
override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
    guard let force = touches.first?.force else { return }
    print("force = \(force)") // force = 2.30229288736979
}
```

## 2.5　画面に線を引いてみる

さて、ここまで Apple Pencil の特性について解説をしてまいりましたが、実際に画面に線を引いてみましょう。
まずは、Apple Pencil の特性を使わず、指でも書けるものから。
今回のテーマは、Apple Pencil を使う部分メインのため、ここまでの詳細解説は省略します。

▼リスト 2.6　基本形

```
class CanvasView: UIImageView {
    private let minLineWidth: CGFloat = 6
    private let maxLineWidth: CGFloat = 15
    private let drawColor: UIColor = UIColor.orange

    override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
        guard let touch = touches.first else { return }

        UIGraphicsBeginImageContextWithOptions(bounds.size, false, 0.0)
        let context = UIGraphicsGetCurrentContext()
```

```
            image?.draw(in: bounds)
            drawStroke(context: context, touch: touch)

            image = UIGraphicsGetImageFromCurrentImageContext()
            UIGraphicsEndImageContext()
        }

        private func drawStroke(context: CGContext?, touch: UITouch) {
            let previousLocation = touch.previousLocation(in: self)
            let currentLocation = touch.location(in: self)

            drawColor.setStroke()

            // 線の属性を設定をする
            context?.setLineWidth(minLineWidth)
            context?.setLineCap(.round)

            // 線のポイントを設定する
            context?.move(to: CGPoint(x: previousLocation.x, y: previousLocation.y))
            context?.addLine(to: CGPoint(x: currentLocation.x, y: currentLocation.y))

            // 線を引く
            context?.strokePath()
        }
    }
```

それでは、このコードに手を加えて、Apple Pencil を有効に使っていってみましょう。

## 2.6 線を滑らかにする

Apple Pencil の特徴で書いた通り、Apple Pencil は、指で書いたのに比べて倍の回数である秒間 240 回のスキャンを行います。
これは通常アプリにタッチが届けられるよりも高いレートであり、多くのアプリはこれほどまでの正確性を必要としておらず、余計なオーバーヘッドを招きたくありません。
そのため、通常の `touches`で得られるタッチは Apple Pencil がスキャンした全ての情報を含んでいません。

しかし、Apple Pencil が活躍するシーンである、ペインティングアプリケーションなどではより精緻なスキャン情報を必要とします。
そこで使われるのが `UIEvent.coalescedTouches` メソッドです。
このメソッドを用いると、システムが前回受信したものの、配信されていない追加のタッチ情報を得ることができます。実際に利用して書き換えたコードが下記になります。

▼リスト 2.7　coalescedTouches

```
override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
    guard let touch = touches.first else { return }
    guard let coalescedTouches = event?.coalescedTouches(for: touch) else { return }

    UIGraphicsBeginImageContextWithOptions(bounds.size, false, 0.0)
    let context = UIGraphicsGetCurrentContext()

    image?.draw(in: bounds)

    for touch in coalescedTouches {
        drawStroke(context: context, touch: touch)
    }

    image = UIGraphicsGetImageFromCurrentImageContext()
    UIGraphicsEndImageContext()
}
```

これで、より正確な描画ができるようになりました。次からはいよいよ Apple Pencil を活用した機能を追加しましょう。

## 2.7 ペンの傾きにより太さを決める

まずはペンの傾きにより、線の太さを変えるところからです。
以下のコードでは、15° から 90° 内での、Apple Pencil の傾き度合いに応じ、比率を算出して、最大線幅に掛け合わせることにより線の幅を決定させています。

▼リスト 2.8 Altitude による線の太さ変更

```
private func lineWidth(_ touch: UITouch) -> CGFloat {
    // ペンの傾きにより線の太さを決める
    return maxLineWidth * altitudeAngleRate(touch)
}

private func altitudeAngleRate(_ touch: UITouch) -> CGFloat {
    let minAltitudeAngle:CGFloat = CGFloat(Double.pi / 12) // 15°
    let maxAltitudeAngle:CGFloat = CGFloat(Double.pi / 2)  // 90°

    let altitudeAngle = touch.altitudeAngle < minAltitudeAngle
        ? minAltitudeAngle : touch.altitudeAngle

    // 傾きにより変化させる太さの比率を求める。
    // 最低傾き（15°）以下の場合に線が最も太くなるようにし、ペンが直立時（90°）に最も細く
なるようにする。
    return 1 - ((altitudeAngle - minAltitudeAngle)
        / (maxAltitudeAngle - minAltitudeAngle))
}
```

## 2.8 ペンの方位角と線を引く方向により線の太さを変える

次に、ペンの方位角と線を引く方向により、線の太さを変えるための記述です。
文章だけですとわかりづらいかと思いますので、下記の図をご覧ください。


▲図 2.5　AzimuthLineWidth

この図のように、ペンが向いている方角に向かって線を引く場合は線を細く、異なる方向に線を引く場合は太くするためのものとなります。この際、MAX 値は 90° となりますので、ペンの方位と、線を引いている方向の差分の角度を取り、それが 90° に対してどの程度の割合かで太さを決定します。

▼リスト 2.9　AzimuthlineWidth

```
private func lineWidth(_ touch: UITouch) -> CGFloat {
    // ペンの傾きにより線の太さを決める
    var lineWidth = maxLineWidth * altitudeAngleRate(touch)
    // 方位角で求めた太さに傾きの係数をかけ合わせる
    lineWidth = lineWidth * azimuthAngleRate(touch)

    return lineWidth > minLineWidth ? lineWidth : minLineWidth
}

private func azimuthAngleRate(_ touch: UITouch) -> CGFloat {
    let previousLocation = touch.previousLocation(in: self)
    let location = touch.location(in: self)
    let pi: CGFloat = CGFloat(Double.pi)
    // 現在のペンの方位角を取る
    let penAngle = touch.azimuthAngle(in: self)

    // 線の向きから角度を取る
```

```
    let strokeVector = CGPoint(x: location.x - previousLocation.x,
                               y: location.y - previousLocation.y)
    let strokeAngle = atan2(strokeVector.y, strokeVector.x)

    // 線の角度とペンの方位角の差分を絶対値で取る
    var angle = abs(strokeAngle - penAngle)

    // 差分角度が MAX で 90° になるようにする
    if angle > pi {
        angle = 2 * pi - angle
    }
    if angle > pi / 2 {
        angle = pi - angle
    }

    // 90° に対する差分角度の割合を求める
    return angle / (pi / 2)
}
```

## 2.9 筆圧によって線の太さを変える

最後に、筆圧です。筆圧についてはすごく単純で、最後に筆圧自身を係数として掛けています。

ただし注意するのが、iPad Pro の場合に `UITouch.force` は指を使用すると常に 0 になるという点です。また、Apple Pencil を用いた場合でも、値は 0 から始まるため、普通にそのまま掛け合わせてしまうと非常に線が細くなってしまいます。そのため、両方の課題を解決するため、1 以上の場合のみ実行するよう分岐を入れます。

▼リスト 2.10　筆圧の反映

```
private func lineWidth(_ touch: UITouch) -> CGFloat {
    ...
    lineWidth = lineWidth > minLineWidth ? lineWidth : minLineWidth
    if touch.force >= 1 {
        lineWidth = lineWidth * touch.force
    }

    return lineWidth
}
```

## 2.10 完成形

これで、一通りの処理が書けました。以下が、完成形となります。

正直、ペイントアプリとして利用するにはまだまだ改善をしなければならないことが多いのですが、まずは Apple Pencil を利用する際の取っ掛かりとして、ご参考までに。;)

▼リスト 2.11　CanvalView

```
class CanvasView: UIImageView {
    private let minLineWidth: CGFloat = 4
    private let maxLineWidth: CGFloat = 16
    private let drawColor: UIColor = UIColor.orange

    override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
        guard let touch = touches.first else { return }
        guard let coalescedTouches =
            event?.coalescedTouches(for: touch) else { return }

        UIGraphicsBeginImageContextWithOptions(bounds.size, false, 0.0)
        let context = UIGraphicsGetCurrentContext()

        image?.draw(in: bounds)

        for touch in coalescedTouches {
            drawStroke(context: context, touch: touch)
        }

        image = UIGraphicsGetImageFromCurrentImageContext()
        UIGraphicsEndImageContext()
    }

    private func drawStroke(context: CGContext?, touch: UITouch) {
        let previousLocation = touch.previousLocation(in: self)
        let currentLocation = touch.location(in: self)

        drawColor.setStroke()

        // 線の属性を設定をする
        context?.setLineWidth(lineWidth(touch))
        context?.setLineCap(.round)

        // 線のポイントを設定する
        context?.move(to: CGPoint(x: previousLocation.x, y: previousLocation.y))
        context?.addLine(to: CGPoint(x: currentLocation.x, y: currentLocation.y))

        // 線を引く
        context?.strokePath()
    }

    private func lineWidth(_ touch: UITouch) -> CGFloat {
        // ペンの傾きにより線の太さを決める
        var lineWidth = maxLineWidth * altitudeAngleRate(touch)

        // 方位角で求めた太さに傾きの係数をかけ合わせる
        lineWidth = lineWidth * azimuthAngleRate(touch)

        lineWidth = lineWidth > minLineWidth ? lineWidth : minLineWidth
        if touch.force >= 1 {
            lineWidth = lineWidth * touch.force
        }

        return lineWidth
    }
```

```
    private func altitudeAngleRate(_ touch: UITouch) -> CGFloat {
        let minAltitudeAngle:CGFloat = CGFloat(Double.pi / 12) // 15°
        let maxAltitudeAngle:CGFloat = CGFloat(Double.pi / 2)  // 90°

        let altitudeAngle = touch.altitudeAngle < minAltitudeAngle
            ? minAltitudeAngle : touch.altitudeAngle

        // 傾きにより変化させる太さの比率を求める。
        // 最低傾き（15°）以下の場合に線が最も太くなるようにし、ペンが直立時（90°）に最
も細くなるようにする。
        return 1 - ((altitudeAngle - minAltitudeAngle)
            / (maxAltitudeAngle - minAltitudeAngle))
    }

    private func azimuthAngleRate(_ touch: UITouch) -> CGFloat {
        let previousLocation = touch.previousLocation(in: self)
        let location = touch.location(in: self)
        let pi: CGFloat = CGFloat(Double.pi)
        // 現在のペンの方位角を取る
        let penAngle = touch.azimuthAngle(in: self)

        // 線の向きから角度を取る
        let strokeVector = CGPoint(x: location.x - previousLocation.x,
                                   y: location.y - previousLocation.y)
        let strokeAngle = atan2(strokeVector.y, strokeVector.x)

        // 線の角度とペンの方位角の差分を絶対値で取る
        var angle = abs(strokeAngle - penAngle)

        // 差分角度が MAX で 90° になるようにする
        if angle > pi {
            angle = 2 * pi - angle
        }
        if angle > pi / 2 {
            angle = pi - angle
        }

        // 90° に対する差分角度の割合を求める
        return angle / (pi / 2)
    }
}
```

# 第 3 章

# 動画再生機能を作ってみよう

## 3.1　はじめに

皆さんこんにちは。最近動画配信アプリが多い気がするのは私だけでしょうか。自分でもやって見たくなったので、動画再生をテーマに記事を書いて見ました。動画再生入門の方はこれを機にやってみましょう！

## 3.2　動画再生するには？

### 動画再生方法

アプリ上で動画を再生する手段としては 2 つあります。ざっと紹介していきます。

- UIWebView
- AVPlayer

### UIWebView

最も簡単な方法でいうと、こちらの方法になります。`WebView`は、Web ページをアプリ内で表示することができる `View`です。
使い方はシンプルで、URL があればすぐに実装できてしまいます。
例えば、`WebView`で `YouTube`を指定すれば、そのアプリは Web 版の YouTube とまるっきり同じになります。なので、動画再生もアプリ上でそのままされるということになります。アプリ内にブラウザを置けるようなものですね。
今回 `WebView`はメインで取り上げません。以下サンプルコードで動作すると思うので、試してみてください。

▼リスト 3.1　swift:WebView サンプルコード

```
import UIKit

class SampleWebViewController: UIViewController {
    @IBOutlet weak var webView: UIWebView!

    override func viewDidLoad() {
        super.viewDidLoad()

        let requestURL = URL(string: "https://www.youtube.com/";;)
        let req = URLRequest(url: requestURL!)
        webView.loadRequest(req)
    }
}
```

注意点としては、実際に表示されるのは Web ページなので、ボタン等のオブジェクトに対して操作をすることができません。単純に Web ページを表示させたい時に利用してみてください。

### AVPlayer

WebViewではカスタマイズができませんでしたが、こちらのライブラリでは色々カスタマイズすることができます。本格的に動画再生機能を実装したい場合、現在だとこちらを使うことになると思います。例えば、動画再生・停止を制御したり、音声を調整したりです。実装方法は WebViewと比べるとコード量は増えますが、圧倒的にできることが増えます。
今回は AVPlayerを使った実装を試したので、後ほどの節でサンプルコードとともに紹介します。
ちなみに、AVPlayerViewControllerオブジェクトが StoryBoardに用意されているので、そちらを使えば簡単に Viewを作ることができますよ。

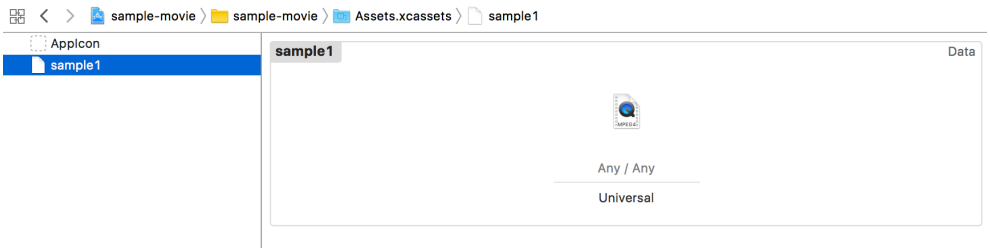
## 3.3　シンプルな動画再生機能実装

では AVPlayerを使った動画再生機能の実装手順を、以下のような流れで紹介します。今回は、アセットに動画ファイルを登録し、アプリロード時直後に動画を再生する機能を実装してみます。

- 動画ファイルの登録
- ViewControllerの設定
- ストーリーボードの実装
- アプリ起動

### アセットに動画ファイルの登録

今回は、mp4 形式の動画ファイルを用意しています。mov ファイルでも問題ありません。まずは、動画ファイルをアセットに登録しましょう。ドラックするだけで OK です。登録したアセット名はこの後利用するので、覚えておきましょう。




▲図 3.1　asset

### ViewControllerの設定

ViewControllerは AVPlayerViewControllerを継承したクラスを作成します。そして今回は、ロード時に動画再生をするので viewDidLoadに動画読み込み処理を記述していきます。
まずは動画ファイルを参照します。動画ファイルは、アセットから NSDataAssetとして取り出し、一時ファイルとして NSTemporaryDirectoryに一度保存します。その保存先 URL を参照する形になります。
まず、一時ファイルとして保存するサンプルコードは以下のようになります。

```
let asset = NSDataAsset(name:"アセット名")
let videoUrl = URL(fileURLWithPath: NSTemporaryDirectory()).
                    appendingPathComponent("アセット名.ファイル形式")
try! asset!.data.write(to: videoUrl)
```

次は URL を参照して動画を再生させるコードになります。
サンプルコードは以下のようになります。動画ファイル URL から AVPlayerItemを作成し、AVPlayerにセットする感じですね。そして、その AVPlayerインスタンスを AVPlayerViewControllerがもつプロパティ playerに代入します。あとは、AVPlayerオブジェクトがもつ制御メソッドを実行するだけで、動画ファイルの制御ができます。今回は

再生するだけにしておきます。

```
let item = AVPlayerItem(url: videoUrl)
let videoPlayer = AVPlayer(playerItem: item)
player = videoPlayer
player?.play()
```

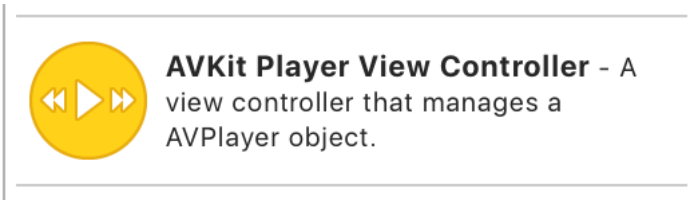
クラスの全体像は以下の通りです。

▼リスト 3.2　swift:AVPlayerViewController のサンプルコード

```
import UIKit
import AVKit

class SampleAVPlayerViewController: AVPlayerViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        let asset = NSDataAsset(name:"アセット名")
        let videoUrl = URL(fileURLWithPath: NSTemporaryDirectory()).
                        appendingPathComponent("アセット名. ファイル形式")
        try! asset!.data.write(to: videoUrl)
        let item = AVPlayerItem(url: videoUrl)
        let videoPlayer = AVPlayer(playerItem: item)
        player = videoPlayer
        player?.play()
    }
}
```

## ストーリーボードの実装

AVPlayerを使って再生するために用意されている、AVKitPlayerViewControllerを使います。まず、オブジェクトライブラリーから以下のアイコンを選択して、引っ張り出しましょう。




▲図 3.2　object

ロード直後に動画を再生させるため、initial ViewControllerに設定しておきましょう。そして、先ほど作成した ViewControllerを Custom Classに設定します。

### アプリ起動

これでアプリを起動してみましょう。アプリが起動後、動画の読み込みが始まり、再生されます。シンプルな再生機能はこれで実装できますが、AVPlayerオブジェクトが持つメソッドを駆使すれば、ボタンパーツ押下時に動画を一時停止させるなど、任意のタイミングで動画を制御することができるので、試してみてください。

## 3.4 発展編 -youtube の動画再生-

### youtube の動画を再生してみよう

シンプルな動画再生機能は実装できたので、次は youtube の動画を再生する機能を実装してみましょう。この章では、youtube の動画再生機能だけでなく、以下のような機能を有するアプリを作ってみます。

- リポジトリ検索
- 再生機能
- ループ再生
- バックグラウンド再生

### ライブラリ紹介

youtube の動画はストリーミング再生する必要がありますが、実は YouTube-Player-iOS-Helperという公式ライブラリが存在します。最終更新が 2016 年でリリースビルドバージョンが 0.1.6 となっており、これ大丈夫か？ 感が漂っていますが、大丈夫です（笑）また、今回のサンプルコードでは、通信用ライブラリとして Alamofireを、パースライブラリとして SwiftyJSONを利用しています。これらは、cocoapods でインストールしておきましょう。

### 必要な設定

実装へ入る前に Goole APIsへアプリの登録をしなければなりません。というのも、ライブラリ検索時に利用する YouTube Data APIを使うにはトークンが必要になるためです。もしも、ライブラリ検索をせず、特定の動画再生のみできれば良いという方は Goole

APIsへの登録は必要ありません。
トークンが手にはいれば、以下のようなフォーマットでライブラリを検索することができます。後ほど利用するので、トークンを覚えておいてください。

```
https://www.googleapis.com/youtube/v3/search?key=【　　ト　　ー　　ク　　ン　　】
&q=youtuber&part=snippet&maxResults=3&order=date
```

## シンプルな動画再生

　では実際に動画再生機能を実装してみます。
まずは、肝となるライブラリを importしましょう。次に、ストーリーボードのメイン View 上に新しい Viewを作成し、Custom classに YTPlayerViewを設定します。この Viewが Youtube 再生プレイヤー描画領域になります。
そして YTPlayerViewを ViewControllerにアウトレット接続します。あとはこのオブジェクトに動画を loadさせるだけなんですが、そこで必要になってくるのが、VideoId です。
VideoIdとは Youtube の動画再生ページに必ずついている ID です。以下のような URL フォーマットになっていると思います。ここの vパラメーターが VIdeoIdです。

```
https://www.youtube.com/watch?v=【VideoID】
```

　YouTube Data APIを使うことで取得できる情報には、VideoIdが含まれています。静的に特定の動画のみを再生したい場合は、ハードコーディングすれば OK です。
VideoIdを使った再生は loadメソッドで実行できます。
さらに今回は、自動再生させるためにデリゲートメソッドを実装しておきましょう。YTPlayerViewDelegateを採用し、playerViewDidBecomeReadyメソッドを実装することで、API の受付が可能になったタイミングで動画が再生されるようになります。（このデリゲートメソッドがない場合、自身でプレイヤーの再生ボタンを押さなくてはいけません。）
シンプルな動画再生機能は以上で実装できます。サンプルコードは以下の通りです。
<br><br>

▼リスト 3.3　swift:youtube 動画再生機能

```
import UIKit
import AVFoundation
import youtube_ios_player_helper

class SampleViewController: UIViewController, YTPlayerViewDelegate {

    @IBOutlet weak var playerView: YTPlayerView!
    var videoId = 【任意の VideoId】

    override func viewDidLoad() {
        super.viewDidLoad()

        playerView.delegate = self
        playerView.load(withVideoId: videoId)
    }

    func playerViewDidBecomeReady(_ playerView: YTPlayerView) {
        playerView.playVideo()
    }
}
```

## リポジトリ検索機能実装

次はリポジトリ検索機能を実装してみましょう。
まずエンドポイントとなる URL を確認します。今回は、以下のフォーマットを利用します。検索キーワードを設定できるようにし、テキストフィールドから入力されたキーワードで検索をかけます。

```
let url = "https://www.googleapis.com/youtube/v3/search?key=【トークン】&q=【検索
キーワード】&part=snippet&order=date";
```

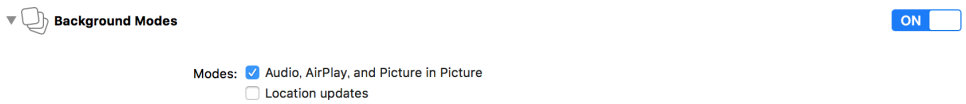
このとき、検索結果が 5 件以上ある場合は、すべてのレスポンスは一度に取得できません。そんなときは、nextPageTokenがあるかどうかで判定します。
nextPageTokenがあるときは、まだレスポンスが残っているので再度リクエストする必要があるのです。次のレスポンスは、nextPageTokenをパラメーターに含めてリクエストします。前のレスポンスを取得するときは、prevPageTokenをパラメーターに含める必要があります。今回は、テーブルビューで 5 件表示し、残りはボタン押下時に再取得するようにしてみましたので、後ほど載せるサンプルコードで確認してみてください。
今回は、以下の情報を取得して利用しています。欲しい情報がどのキーで取得できるのか、詳細はドキュメントを見ておきましょう。

- 動画タイトル（"items" -> "snippet" -> "title"）

- VideoId（"items" -> "id" -> "videoId"）

## バックグラウンド再生 & ループ再生

このままで動画再生はできるのですが、画面にロックをかけると動画が停止してしまいます。せっかくなので、バックグラウンドにも対応させました。
Capabilities -> Background Modesを ON にして、Audio, AirPlay and Picture in Pictureにチェックと入れることで対応できます。
また、コード上に以下の記述をしておきましょう。これでバックグラウンドでも音楽は再生することができます。




▲図 3.3　background-setting

```
let session = AVAudioSession.sharedInstance()
do {
    try session.setCategory(AVAudioSessionCategoryPlayback)
} catch  {
    fatalError("---ng---")
}
```

また、動画にはループ機能を実装しています。ドキュメントに詳細が記載されていますが、ロード時のパラメーターを以下のようにすれば OK です

```
playerView.load(withVideoId: videoId, playerVars: [
    "loop": 1,
    "playlist": [videoId]
])
```

## 両方あわせた実装全体

リポジトリ検索 -> videoId 抽出 -> 動画再生となるように調整してみました。最低限の情報しか含めていないので、その他いろいろ調整してみてください。

▼リスト 3.4　swift:youtube 検索・再生機能のサンプルコード

```
import UIKit
import Alamofire
import AVFoundation
import AVKit
import SwiftyJSON

class ViewController: UIViewController {
    @IBOutlet weak var txtFSearchQuery: UITextField!

    var videoPlayer: AVPlayer!
    let baseUrl = "https://www.googleapis.com/youtube/v3/search";

    override func viewDidLoad() {
        super.viewDidLoad()

        txtFSearchQuery.inputAccessoryView = getToolBar()

        let session = AVAudioSession.sharedInstance()
        do {
            try session.setCategory(AVAudioSessionCategoryPlayback)
        } catch  {
            fatalError("---ng---")
        }
    }

    @IBAction func searchRepo(_ sender: Any) {
        if !createUrl(txtFSearchQuery.text!).isEmpty {
            Alamofire.request(createUrl(txtFSearchQuery.text!))
                .responseJSON { response in
                    let data = JSON(response.result.value)
                    let storyboard =
                         UIStoryboard(name: "SearchResult", bundle: nil)
                    let viewController =
                         storyboard.instantiateInitialViewController()!
                          as! SearchResultViewController
                    viewController.requestUrl =
                         self.createUrl(self.txtFSearchQuery.text!)
                    viewController.response = data
                    self.navigationController?.
                         pushViewController(viewController, animated: true)
            }
        }
    }

    func createUrl(_ query: String) -> String {
        guard !query.isEmpty else {
            return ""
        }
        let url = "\(baseUrl)?key=【トークン】&q=\(query)&part=snippet&order=date"
        return url.addingPercentEncoding(withAllowedCharacters: .urlQueryAllowed)!
    }

    func getToolBar() -> UIToolbar {
        let toolbar =
         UIToolbar(frame: CGRect(x: 0, y: 0, width: view.frame.width, height: 35))
        let doneItem = UIBarButtonItem(title: "閉                                じ
る", style: .plain, target: self, action: #selector(self.actionBtnClose))
        toolbar.setItems([doneItem], animated: true)
```

```
        return toolbar
    }

    func actionBtnClose() {
        txtFSearchQuery.resignFirstResponder()
    }
}

class SearchResultViewController: UIViewController, UITableViewDelegate,
 UITableViewDataSource {
    @IBOutlet weak var prefButton: UIButton!
    @IBOutlet weak var nextButton: UIButton!
    @IBOutlet weak var baseTableView: UITableView!

    var requestUrl = ""
    var response = JSON()

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

    func initButton() {
        prefButton.isEnabled =
             response["prevPageToken"].string != nil ? true : false
        nextButton.isEnabled =
             response["nextPageToken"].string != nil ? true : false
    }

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
    -> Int {
        return response["items"].count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
    -> UITableViewCell {
        let cell = UITableViewCell()
        cell.textLabel?.text =
         "* \(response["items"][indexPath.row]["snippet"]["title"]))"
        return cell
    }

    func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
        let storyboard = UIStoryboard(name: "Player", bundle: nil)
        let viewController = storyboard.instantiateInitialViewController()!
         as! PlayerViewController
        viewController.videoId =
         "\(response["items"][indexPath.row]["id"]["videoId"])"
        self.navigationController?
         .pushViewController(viewController, animated: true)
    }

    @IBAction func prevPage(_ sender: Any) {
        let url = "\(requestUrl)&pageToken=\(response["prevPageToken"])"
        requestApi(url)
    }

    @IBAction func nextPage(_ sender: Any) {
```

```
        let url = "\(requestUrl)&pageToken=\(response["nextPageToken"])"
        requestApi(url)
    }

    func requestApi(_ url: String) {
        Alamofire.request(url)
            .responseJSON { response in
                self.response = JSON(response.result.value)
                self.initButton()
                self.baseTableView.reloadData()
        }
    }
}

import youtube_ios_player_helper

class PlayerViewController: UIViewController, YTPlayerViewDelegate {
    @IBOutlet weak var playerView: YTPlayerView!
    var videoId = ""

    override func viewDidLoad() {
        super.viewDidLoad()

        playerView.delegate = self
        playerView.load(withVideoId: videoId, playerVars: [
            "loop": 1,
            "playlist": [videoId]
            ])
    }

    func playerViewDidBecomeReady(_ playerView: YTPlayerView) {
        playerView.playVideo()
    }
}
```

## 3.5 おわりに

アプリ内で動画を再生するサンプルを載せつつ紹介してみましたが、いかがだったでしょうか。youtubeの動画再生は、ライブラリが用意されているのでとても簡単に実装できましたね。リポジトリ検索の方が処理が多いくらいです。
いずれは生放送をアプリ上で再生できる機能を作ってみたいものです！
本記事が動画機能入門に役立てれば幸いです。

# TAKASHI FUN CLUB Vol.2　iOS 開発篇

2017 年 10 月 22 日　発行
著　者　　たかしファンクラブ
イラスト　@onunu
編　集　　@roana0229
印刷所　　日光企画