RxSwift VS ReactiveSwift

自己紹介

発表者:

- 深見龍一
- iOS Engineer
- 21卒
- 東京都在住
- 福岡出身

アカウント:

- Twitter: @ryu1_ryu421
- Github: @fukami421
- Qiita: @ryu1_f

difference between RxSwift and ReactiveSwift

	RxSwift	ReactiveSwift
Hot	Obserbable<Int>	Signal<Int, Never>
Cold	Obserbable<Int>	SignalProducer<Int, Never>

1. ReactiveSwiftは、Hot / Coldを明確に区別。
2. ReactiveSwiftは、Errorの方を指定できる。

ReactiveSwiftは、Hot / Coldを明確に区別(1)

```
// Observerを生成する
let signalObserver = Signal<Int, NoError>.Observer(
value: { value in
    print("Time elapsed = \(value)")
}, completed: {
    print("completed")
}, interrupted: {
    print("interrupted")
})

// Signalを生成する
let (output, input) = Signal<Int, NoError>.pipe()
//Send value to signal
for i in 0..<10 {
    DispatchQueue.main.asyncAfter(deadline: .now() + 5.0 *  Double(i)) {
        input.send(value: i)
    }
}

// Signalを監視する
output.observe(signalObserver)
```

observable(Single)のinputに値を流し、
observable(Single)のoutputをobserverが監視する。

ReactiveSwiftは、Hot / Coldを明確に区別(2)

```
let someRequest = SignalProducer<Void, NSError> { (innerObserver, disposable) in
    sendRequest(request) { result in
        switch result {
        case .success(value):
            innerObserver.send(value: value)
            innerObserver.sendCompleted()
        case .failure(error):
            innerObserver.send(error: error)
    }
}
someRequest.startWithResult { result in
    switch result {
    case let .success(value):
        print("value: \(value)")
    case let .failure(error):
        print("error: \(error)")
    }
}
```

startすることで、Signalを生成し、observerが登録される。

Hot / Coldを明確に区別することの何が嬉しいか

ex. RxSwift

```
let subject = PublishSubject<ApiModel>() // PublishSubjectはHot

let userName = subject.map{ "Mr." + $0.items[0].login } // mapしてColdに変換

// 2回subscribeする
userName
    .subscribe(onNext:{ name in
    print(name)
    })
.disposed(by: self.disposeBag)

userName
    .subscribe(onNext:{ name in
    print(name)
    })
.disposed(by: self.disposeBag)
```

subscribeした分だけObserbableが生成され、メモリの無駄遣い、バグの原因になりかねない。
(mapが2回実行される。)

ReactiveSwiftは、Errorの方を指定できる(1)

ex. RxSwift

```
// RxSwift の場合
let myObservable: Observable<Int> = ...

myObservable
    .subscribe(onError: { error in
        guard let myError = error as? MyError {
            // ここに来ることはないという"暗黙の了解"が発生する
            return
        }
        // 何かする
    })
    .disposed(by: disposeBag)
```

エラーの型を間違えるとアプリが落ちる可能性あり。

ReactiveSwiftは、Errorの方を指定できる(2)

ex. ReactiveSwift

```
// ReactiveSwift の場合
let myProducer: SignalProducer<Int, MyError> = ...

myProducer.startWithFailed { (myError: MyError) in
    // myError は MyError 型ということがコンパイル時に保証される！
}
```

エラーの型をコンパイル時に保証できる！

```
// エラーの型が NoError
let noErrorProducer: SignalProducer<Int, NoError> = ...
```

エラーが起きないことを型で宣言できる！

おまけ

Observing changes to a simple UI control

```
-priceSignal
+priceLabel.reactive.text <~ priceSignal
      .map { "\($0) USD" }
-     .bindTo(priceLabel.rx.text)
-     .addDisposableTo(disposeBag)
```

disposed(by: self.disposeBag)
を書かなくて済む。

Creating a Network

```
-func load<A>(_ resource: Resource<A>) -> Observable<A> {
-    return Observable.create { observer in
+func load<A>(_ resource: Resource<A>) -> SignalProducer<A, AnyError> {
+    return SignalProducer { observer, disposable in
         print("start loading")
         self.load(resource) { result in
             sleep(1)
             switch result {
             case .error(let error):
-                observer.onError(error)
+                observer.send(error: AnyError(error))
             case .success(let value):
-                observer.onNext(value)
-                observer.onCompleted()
+                observer.send(value: value)
+                observer.sendCompleted()
             }
         }
-        return Disposables.create()
     }
 }
```

APIを叩く
例で比較

Action クラス(1)

```
class APIClient {

    func searchUsers(name: String) -> SignalProducer<[User], APIError> {
        return ...
    }
}

let apiClient = APIClient()

// searchUsers を呼び出す Action を定義しています
let action = Action<String, [User], APIError> { input in
    return apiClient.searchUsers(name: input) // ②
}

// searchUsers の値を購読
action.values.observeValues { (users: [User]) in
    print(users) // ③-A
}

// searchUsers のエラーを購読
action.errors.observeValues { (error: APIError) in
    print(error) // ③-B
}

// "John" をトリガーに action を実行
action.apply("John").start() // ①
```

Action クラス(2)

特長

- 1度に実行できるActionは１つまで

→APIを2重に叩かずに済む

- エラーが発生しても購読が終了しない

→あまり理解できていない

- Actionが実行中かどうか判定できる

→UIAcitivityIndicatorを回しやすい

おしまい