

目錄

RxSwift 中文文档	1.1
1. 为什么要使用 RxSwift?	1.2
2. 你好 RxSwift!	1.3
3. 函数响应式编程	1.4
3.1 函数式编程	1.4.1
3.2 函数式编程 -> 函数响应式编程	1.4.2
3.3 数据绑定	1.4.3
4. RxSwift 核心	1.5
4.1 Observable - 可监听序列	1.5.1
Single	1.5.1.1
Completable	1.5.1.2
Maybe	1.5.1.3
Driver	1.5.1.4
Signal	1.5.1.5
ControlEvent	1.5.1.6
4.2 Observer - 观察者	1.5.2
AnyObserver	1.5.2.1
Binder	1.5.2.2
4.3 Observable & Observer 既是可监听序列也是观察者	1.5.3
AsyncSubject	1.5.3.1
PublishSubject	1.5.3.2
ReplaySubject	1.5.3.3
BehaviorSubject	1.5.3.4
Variable (已弃用)	1.5.3.5
ControlProperty	1.5.3.6
4.4 Operator - 操作符	1.5.4
4.5 Disposable - 可被清除的资源	1.5.5
4.6 Schedulers - 调度器	1.5.6
4.7 Error Handling - 错误处理	1.5.7
5. 如何选择操作符?	1.6
amb	1.6.1
buffer	1.6.2
catchError	1.6.3

combineLatest	1.6.4
concat	1.6.5
concatMap	1.6.6
connect	1.6.7
create	1.6.8
debounce	1.6.9
debug	1.6.10
deferred	1.6.11
delay	1.6.12
delaySubscription	1.6.13
dematerialize	1.6.14
distinctUntilChanged	1.6.15
do	1.6.16
elementAt	1.6.17
empty	1.6.18
error	1.6.19
filter	1.6.20
flatMap	1.6.21
flatMapLatest	1.6.22
from	1.6.23
groupBy	1.6.24
ignoreElements	1.6.25
interval	1.6.26
just	1.6.27
map	1.6.28
merge	1.6.29
materialize	1.6.30
never	1.6.31
observeOn	1.6.32
publish	1.6.33
reduce	1.6.34
refCount	1.6.35
repeatElement	1.6.36
replay	1.6.37
retry	1.6.38
sample	1.6.39

scan	1.6.40
shareReplay	1.6.41
single	1.6.42
skip	1.6.43
skipUntil	1.6.44
skipWhile	1.6.45
startWith	1.6.46
subscribeOn	1.6.47
take	1.6.48
takeLast	1.6.49
takeUntil	1.6.50
takeWhile	1.6.51
timeout	1.6.52
timer	1.6.53
using	1.6.54
window	1.6.55
withLatestFrom	1.6.56
zip	1.6.57
6. 更多示例	1.7
ImagePicker - 图片选择器	1.7.1
TableViewSectionedViewController - 多层级的列表页	1.7.2
Calculator - 计算器	1.7.3
7. RxSwift 常用架构	1.8
7.1 MVVM	1.8.1
Github Signup (示例)	1.8.1.1
7.2 RxFeedback	1.8.2
Github Search (示例)	1.8.2.1
7.3 ReactorKit	1.8.3
Github Search (示例)	1.8.3.1
8. RxSwift 生态系统	1.9
9. 学习资源	1.10
10. 关于本文档	1.11
10.1 文档更新日志	1.11.1
食谱	1.12
RxSwift 5 更新了什么?	1.12.1
RxRelay	1.12.2

纯函数	1.12.3
附加作用	1.12.4
共享附加作用	1.12.5



RxSwift: ReactiveX for Swift

ReactiveX（简写: Rx）是一个可以帮助我们简化异步编程的框架。

RxSwift 是 Rx 的 Swift 版本。

它尝试将原有的一些概念移植到 iOS/macOS 平台。

你可以在这里找到跨平台文档 [ReactiveX.io](#)。

KVO, 异步操作 和 流 全部被统一成抽象序列。这就是为什么 Rx 会如此简单, 优雅和强大。

操作

加入 RxSwift QQ 交流群: **871293356**

[下载文档电子书](#)

文档更新日志

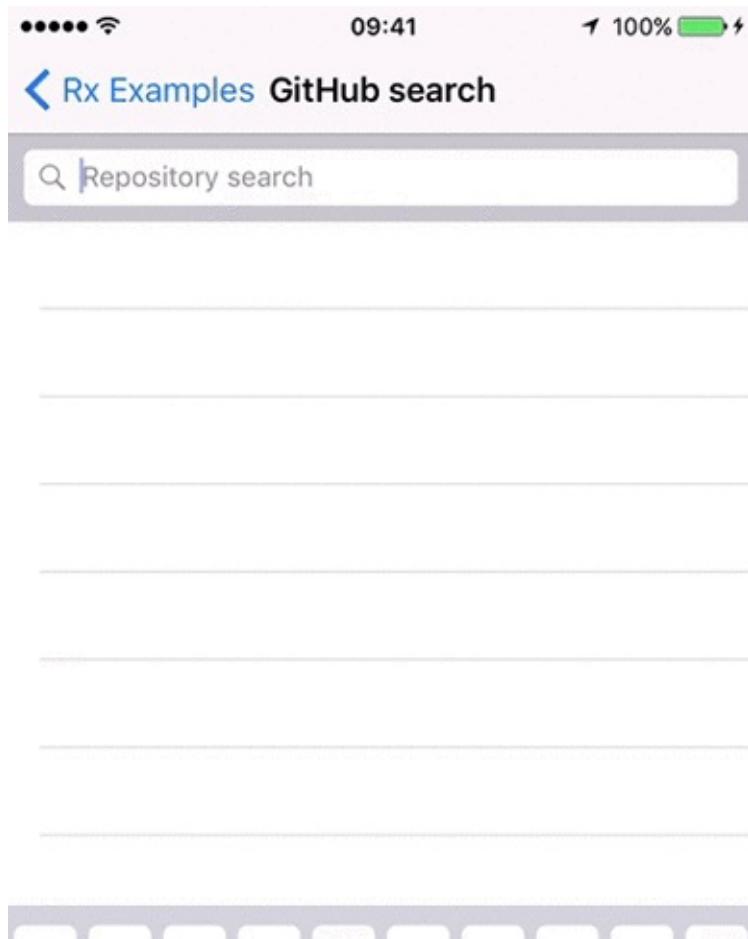
19年5月21日 (RxSwift 5)

- RxSwift 5 更新了什么?
- 引入[食谱章节](#)
- [Signal](#)
- [RxRelay](#)
- 纯函数
- 附加作用
- [共享附加作用](#)
- 更新文档以适配 RxSwift 5
- 更新 QQ 群号为: 871293356

[查看更多...](#)

示例

[Github 搜索...](#)



定义搜索结果 ...

```
let searchResults = searchBar.rx.text.orEmpty
    .throttle(0.3, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<[Repository]> in
        if query.isEmpty {
            return .just([])
        }
        return searchGitHub(query)
            .catchErrorJustReturn([])
    }
    .observeOn(MainScheduler.instance)
```

... 然后将结果绑定到 tableview 上

```
searchResults
    .bind(to: tableView.rx.items(cellIdentifier: "Cell")) {
        (index, repository, cell) in
        cell.textLabel?.text = repository.name
        cell.detailTextLabel?.text = repository.url
    }
    .disposed(by: disposeBag)
```

必备条件

- Xcode 10.2
- Swift 5.0

对于 Xcode 10.1 以下版本, 请使用 [RxSwift 4.5](#)。

安装

安装 RxSwift 不需要任何第三方依赖。

以下是当前支持的安装方法:

手动

打开 Rx.xcworkspace, 选中 RxExample 并且点击运行。此方法将构建所有内容并运行示例应用程序。

CocoaPods

pod --version : 1.3.1 已通过测试

```
# Podfile
use_frameworks!

target 'YOUR_TARGET_NAME' do
    pod 'RxSwift', '~> 5.0'
    pod 'RxCocoa', '~> 5.0'
end

# RxTests 和 RxBlocking 将在单元/集成测试中起到重要作用
target 'YOUR_TESTING_TARGET' do
    pod 'RxBlocking', '~> 5.0'
    pod 'RxTest', '~> 5.0'
end
```

替换 `YOUR_TARGET_NAME` 然后在 `Podfile` 目录下, 终端输入:

```
$ pod install
```

Carthage

官方支持 0.33 及以上版本。

添加到 `Cartfile`

```
github "ReactiveX/RxSwift" ~> 5.0
```

```
$ carthage update
```

Carthage 作为静态库。

如果您希望使用 Carthage 将 RxSwift 构建为静态库, 在使用 Carthage 构建之前, 您可以使用以下脚本手动修改框架类型:

```
carthage update RxSwift --platform iOS --no-build
sed -i -e 's/MACH_O_TYPE = mh_dylib/MACH_O_TYPE = staticlib/g' Carthage/Checkouts/RxSwift/Rx.xcodeproj/project.pbxproj
carthage build RxAlamofire --platform iOS
```

Swift Package Manager

创建 `Package.swift` 文件。

```
// swift-tools-version:5.0

import PackageDescription

let package = Package(
    name: "RxTestProject",
    dependencies: [
        .package(url: "https://github.com/ReactiveX/RxSwift.git", from: "5.0.0")
    ],
    targets: [
        .target(name: "RxTestProject", dependencies: ["RxSwift", "RxCocoa"])
    ]
)
```

```
$ swift build
```

如果构建或测试一个模块对 RxTest 存在依赖，设置 `TEST=1`。

```
$ TEST=1 swift test
```

使用 git submodules 手动集成

- 添加 RxSwift 作为子模块

```
$ git submodule add git@github.com:ReactiveX/RxSwift.git
```

- 拖拽 `Rx.xcodeproj` 到项目中
- 前往 `Project > Targets > Build Phases > Link Binary With Libraries`，点击 `+` 并且选中 `RxSwift-[Platform]` 和 `RxCocoa-[Platform]`

为什么要使用 RxSwift ?

我们先看一下 RxSwift 能够帮助我们做些什么：

Target Action

传统实现方法：

```
button.addTarget(self, action: #selector(buttonTapped), for: .touchUpInside)
```

```
func buttonTapped() {
    print("button Tapped")
}
```

通过 Rx 来实现：

```
button.rx.tap
    .subscribe(onNext: {
        print("button Tapped")
    })
    .disposed(by: disposeBag)
```

你不需要使用 Target Action，这样使得代码逻辑清晰可见。

代理

传统实现方法：

```
class ViewController: UIViewController {
    ...
    override func viewDidLoad() {
        super.viewDidLoad()
        scrollView.delegate = self
    }
}

extension ViewController: UIScrollViewDelegate {
    func scrollViewDidScroll(_ scrollView: UIScrollView) {
        print("contentOffset: \(scrollView.contentOffset)")
    }
}
```

通过 Rx 来实现：

```
class ViewController: UIViewController {
    ...
    override func viewDidLoad() {
        super.viewDidLoad()

        scrollView.rx.contentOffset
            .subscribe(onNext: { contentOffset in
                print("contentOffset: \(contentOffset)")
            })
            .disposed(by: disposeBag)
    }
}
```

你不需要书写代理的配置代码，就能获得想要的结果。

闭包回调

传统实现方法：

```
URLSession.shared.dataTask(with: URLRequest(url: url)) {
    (data, response, error) in
    guard error == nil else {
        print("Data Task Error: \(error!)")
        return
    }

    guard let data = data else {
        print("Data Task Error: unknown")
        return
    }

    print("Data Task Success with count: \(data.count)")
}.resume()
```

通过 Rx 来实现：

```
URLSession.shared.rx.data(request: URLRequest(url: url))
    .subscribe(onNext: { data in
        print("Data Task Success with count: \(data.count)")
    }, onError: { error in
        print("Data Task Error: \(error)")
    })
    .disposed(by: disposeBag)
```

回调也变得十分简单

通知

传统实现方法：

```
var ntfObserver: NSObjectProtocol!

override func viewDidLoad() {
    super.viewDidLoad()

    ntfObserver = NotificationCenter.default.addObserver(
        forName: .UIApplicationWillEnterForeground,
        object: nil, queue: nil) { [notification] in
        print("Application Will Enter Foreground")
    }
}

deinit {
    NotificationCenter.default.removeObserver(ntfObserver)
}
```

通过 Rx 来实现：

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

    NotificationCenter.default.rx
        .notification(.UIApplicationWillEnterForeground)
        .subscribe(onNext: { [notification] in
            print("Application Will Enter Foreground")
        })
        .disposed(by: disposeBag)
}
```

你不需要去管理观察者的生命周期，这样你就有更多精力去关注业务逻辑。

多个任务之间有依赖关系

例如，先通过用户名密码取得 Token 然后通过 Token 取得用户信息，

传统实现方法：

```
/// 用回调的方式封装接口
enum API {

    /// 通过用户名密码取得一个 token
    static func token(username: String, password: String,
                      success: (String) -> Void,
                      failure: (Error) -> Void) { ... }

    /// 通过 token 取得用户信息
}
```

1. 为什么要使用 RxSwift?

```
static func userinfo(token: String,
    success: (UserInfo) -> Void,
    failure: (Error) -> Void) { ... }
}
```

```
/// 通过用户名和密码获取用户信息
API.token(username: "beeth0ven", password: "987654321",
    success: { token in
        API.userInfo(token: token,
            success: { userInfo in
                print("获取用户信息成功: \(userInfo)")
            },
            failure: { error in
                print("获取用户信息失败: \(error)")
            })
    },
    failure: { error in
        print("获取用户信息失败: \(error)")
    })
}
```

通过 Rx 来实现：

```
/// 用 Rx 封装接口
enum API {
    /// 通过用户名密码取得一个 token
    static func token(username: String, password: String) -> Observable<String> { .
    . }

    /// 通过 token 取得用户信息
    static func userInfo(token: String) -> Observable<UserInfo> { ... }
}
```

```
/// 通过用户名和密码获取用户信息
API.token(username: "beeth0ven", password: "987654321")
    .flatMapLatest(API.userInfo)
    .subscribe(onNext: { userInfo in
        print("获取用户信息成功: \(userInfo)")
    }, onError: { error in
        print("获取用户信息失败: \(error)")
    })
    .disposed(by: disposeBag)
```

这样你可以[避免回调地狱](#)，从而使得代码易读，易维护。

等待多个并发任务完成后处理结果

例如，需要将两个网络请求合并成一个，

通过 Rx 来实现：

```
/// 用 Rx 封装接口
enum API {

    /// 取得老师的详细信息
    static func teacher(teacherId: Int) -> Observable<Teacher> { ... }

    /// 取得老师的评论
    static func teacherComments(teacherId: Int) -> Observable<[Comment]> { ... }
}
```

```
/// 同时取得老师信息和老师评论
Observable.zip(
    API.teacher(teacherId: teacherId),
    API.teacherComments(teacherId: teacherId)
).subscribe(onNext: { (teacher, comments) in
    print("获取老师信息成功: \(teacher)")
    print("获取老师评论成功: \(comments.count) 条")
}, onError: { error in
    print("获取老师信息或评论失败: \(error)")
})
.disposed(by: disposeBag)
```

这样你可用寥寥几行代码来完成相当复杂的异步操作。

那么为什么要使用 RxSwift ?

- 复合 - Rx 就是复合的代名词
- 复用 - 因为它易复合
- 清晰 - 因为声明都是不可变更的
- 易用 - 因为它抽象的了异步编程，使我们统一了代码风格
- 稳定 - 因为 Rx 是完全通过单元测试的

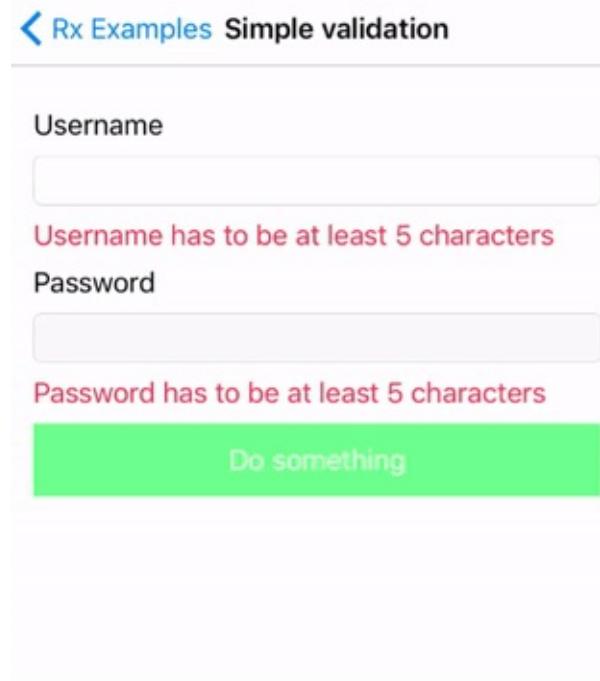
你好 RxSwift!

我的第一个 RxSwift 应用程序 - 输入验证:

这是一个模拟用户登录的程序。

- 当用户输入用户名时，如果用户名不足 5 个字就给出红色提示语，并且无法输入密码，当用户名符合要求时才可以输入密码。
- 同样的当用户输入的密码不到 5 个字时也给出红色提示语。
- 当用户名和密码有一个不符合要求时底部的绿色按钮不可点击，只有当用户名和密码同时有效时按钮才可点击。
- 当点击绿色按钮后弹出一个提示框，这个提示框只是用来做演示而已。

你可以下载[这个例子](#)并在模拟器上运行，这样可以帮助于你理解整个程序的交互：



这个页面主要由 5 各元素组成：

- 用户名输入框
- 用户名提示语（红色）
- 密码输入框
- 密码提示语（红色）
- 操作按钮（绿色）

```
class SimpleValidationViewController : ViewController {  
  
    @IBOutlet weak var usernameOutlet: UITextField!  
    @IBOutlet weak var usernameValidOutlet: UILabel!
```

```

@IBOutlet weak var passwordOutlet: UITextField!
@IBOutlet weak var passwordValidOutlet: UILabel!

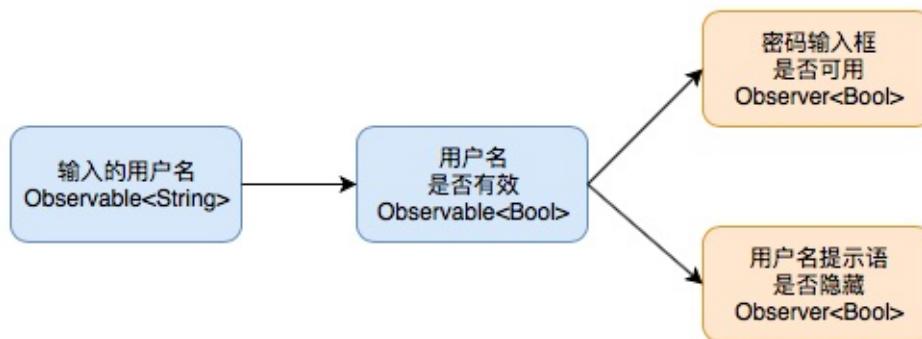
@IBOutlet weak var doSomethingOutlet: UIButton!
...

}

```

这里需要完成 4 个交互：

- 当用户名输入不到 5 个字时显示提示语，并且无法输入密码



```

override func viewDidLoad() {
    super.viewDidLoad()

    ...

    // 用户名是否有效
    let usernameValid = usernameOutlet.rx.text.orEmpty
        // 用户名 -> 用户名是否有效
        .map { $0.count >= minimalUsernameLength }
        .share(replay: 1)

    ...

    // 用户名是否有效 -> 密码输入框是否可用
    usernameValid
        .bind(to: passwordOutlet.rx.isEnabled)
        .disposed(by: disposeBag)

    // 用户名是否有效 -> 用户名提示语是否隐藏
    usernameValid
        .bind(to: usernameValidOutlet.rx.isHidden)
        .disposed(by: disposeBag)

    ...
}

```

```
}
```

当用户修改用户名输入框的内容时就会产生一个新的用户名, 然后通过 `map` 方法将它转化成用户名是否有效, 最后通过 `bind(to: ...)` 来决定密码输入框是否可用以及提示语是否隐藏。

- 当密码输入不到 5 个字时显示提示文字



```

override func viewDidLoad() {
    super.viewDidLoad()

    ...

    // 密码是否有效
    let passwordValid = passwordOutlet.rx.text.orEmpty
        // 密码 -> 密码是否有效
        .map { $0.count >= minimalPasswordLength }
        .share(replay: 1)

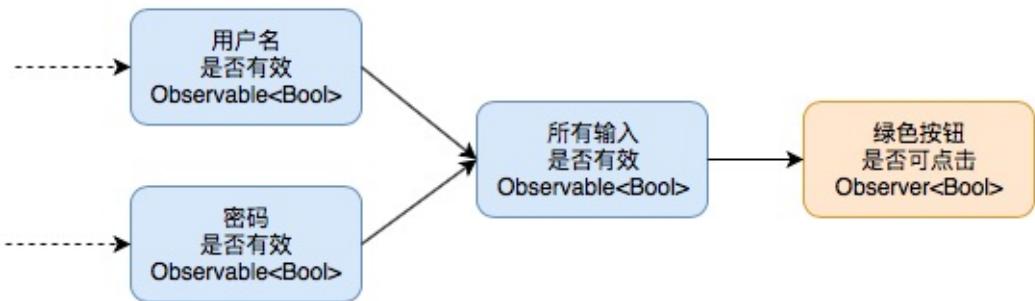
    ...

    // 密码是否有效 -> 密码提示语是否隐藏
    passwordValid
        .bind(to: passwordValidOutlet.rx.isHidden)
        .disposed(by: disposeBag)

    ...
}
  
```

这个和用用户名来控制提示语的逻辑是一样的。

- 当用户名和密码都符合要求时, 绿色按钮才可点击



```

override func viewDidLoad() {
    super.viewDidLoad()

    ...

    // 用户名是否有效
    let usernameValid = ...

    // 密码是否有效
    let passwordValid = ...

    ...

    // 所有输入是否有效
    let everythingValid = Observable.combineLatest(
        usernameValid,
        passwordValid
    ) { $0 && $1 } // 取用户名和密码同时有效
    .share(replay: 1)

    ...

    // 所有输入是否有效 -> 绿色按钮是否可点击
    everythingValid
        .bind(to: doSomethingOutlet.rx.isEnabled)
        .disposed(by: disposeBag)

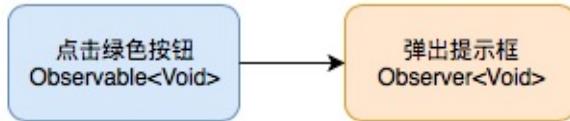
    ...
}

}

```

通过 `Observable.combineLatest(...){ ... }` 来将 用户名是否有效 以及 密码是否有效 合并出 两者是否同时有效 ,然后用它来控制绿色按钮是否可点击。

- 点击绿色按钮后，弹出一个提示框



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

    ...

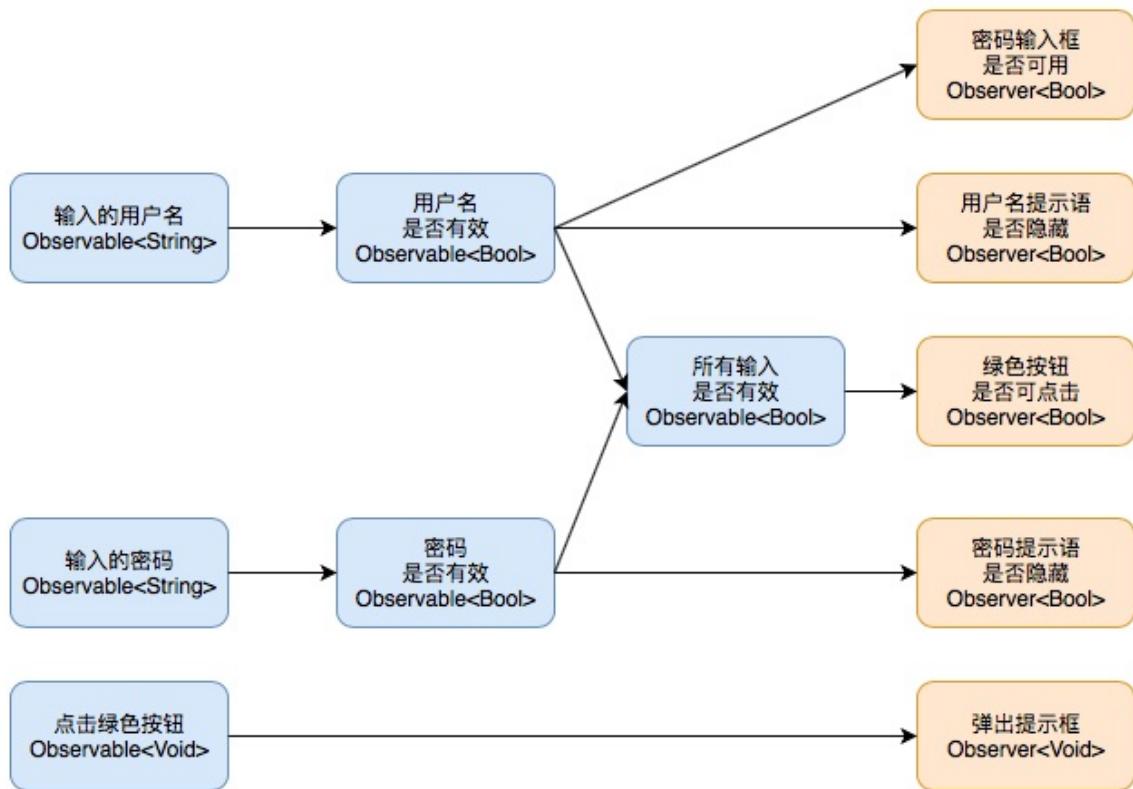
    // 点击绿色按钮 -> 弹出提示框
    doSomethingOutlet.rx.tap
        .subscribe(onNext: { [weak self] in self?.showAlert() })
        .disposed(by: disposeBag)
}

func showAlert() {
    let alertView = UIAlertView(
        title: "RxExample",
        message: "This is wonderful",
        delegate: nil,
        cancelButtonTitle: "OK"
    )

    alertView.show()
}
```

在点击绿色按钮后，弹出一个提示框

这样 4 个交互都完成了，现在我们纵观全局看下这个程序是一个什么样的结构：



然后看一下完整的代码:

```

override func viewDidLoad() {
    super.viewDidLoad()

    usernameValidOutlet.text = "Username has to be at least \(minimalUsernameLength) characters"
    passwordValidOutlet.text = "Password has to be at least \(minimalPasswordLength) characters"

    let usernameValid = usernameOutlet.rx.text.orEmpty
        .map { $0.count >= minimalUsernameLength }
        .share(replay: 1)

    let passwordValid = passwordOutlet.rx.text.orEmpty
        .map { $0.count >= minimalPasswordLength }
        .share(replay: 1)

    let everythingValid = Observable.combineLatest(
        usernameValid,
        passwordValid
    ) { $0 && $1 }
        .share(replay: 1)

    usernameValid
        .bind(to: passwordOutlet.rx.isEnabled)
  
```

```

    .disposed(by: disposeBag)

usernameValid
    .bind(to: usernameValidOutlet.rx.isHidden)
    .disposed(by: disposeBag)

passwordValid
    .bind(to: passwordValidOutlet.rx.isHidden)
    .disposed(by: disposeBag)

everythingValid
    .bind(to: doSomethingOutlet.rx.isEnabled)
    .disposed(by: disposeBag)

doSomethingOutlet.rx.tap
    .subscribe(onNext: { [weak self] in self?.showAlert() })
    .disposed(by: disposeBag)
}

func showAlert() {
    let alertView = UIAlertView(
        title: "RxExample",
        message: "This is wonderful",
        delegate: nil,
        cancelButtonTitle: "OK"
    )

    alertView.show()
}

```

你会发现你可以用几行代码完成如此复杂的交互。这可以大大提升我们的开发效率。

更多疑问

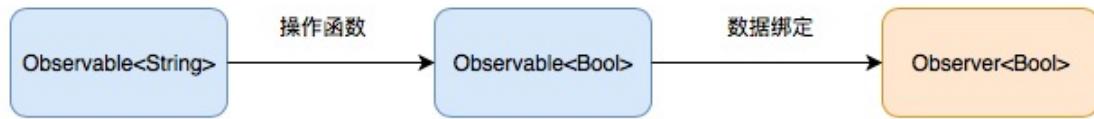
- `share(replay: 1)` 是用来做什么的?

我们用 `usernameValid` 来控制用户名提示语是否隐藏以及密码输入框是否可用。`shareReplay` 就是让他们共享这一个源，而不是为他们单独创建新的源。这样可以减少不必要的开支。

- `disposed(by: disposeBag)` 是用来做什么的?

和我们所熟悉的对象一样，每一个绑定也是有生命周期的。并且这个绑定是可以被清除的。`disposed(by: disposeBag)` 就是将绑定的生命周期交给 `disposeBag` 来管理。当 `disposeBag` 被释放的时候，那么里面尚未清除的绑定也就被清除了。这就相当于是在用 [ARC](#) 来管理绑定的生命周期。这个内容会在 [Disposable](#) 章节详细介绍。

函数响应式编程



函数响应式编程是一种编程范式。它是通过构建函数操作数据序列，然后对这些序列做出响应的编程方式。它结合了**函数式编程**以及**响应式编程**

这里先介绍一下**函数式编程**。

函数式编程

```
[🍔, 🍔, 🍗, 🍗].map(cook) // [🍔, 🍔, 🍗, 🍗]
[🍔, 🍔, 🍗, 🍗].filter(isVegetarian) // [🍟, 🍪]
[🍔, 🍔, 🍗, 🍗].reduce(😴, eat) // 😊
```

函数式编程是一种编程范式，它需要我们将函数作为参数传递，或者作为返回值返还。我们可以通过组合不同的函数来得到想要的结果。

我们来看一下这几个例子：

```
// 全校学生
let allStudents: [Student] = getSchoolStudents()

// 三年二班的学生
let gradeThreeClassTwoStudents: [Student] = allStudents
    .filter { student in student.grade == 3 && student.class == 2 }
```

由于我们想要得到三年二班的学生，所以我们把三年二班的判定函数作为参数传递给 `filter` 方法，这样就能从全校学生中过滤出三年二班的学生。

```
// 三年二班的每一个男同学唱一首《一剪梅》
gradeThreeClassTwoStudents
    .filter { student in student.sex == .male }
    .forEach { boy in boy.singASong(name: "一剪梅") }
```

同样的我们将性别的判断函数传递给 `filter` 方法，这样就能从三年二班的学生中过滤出男同学，然后将唱歌作为函数传递给 `forEach` 方法。于是每一个男同学都要唱《一剪梅》。

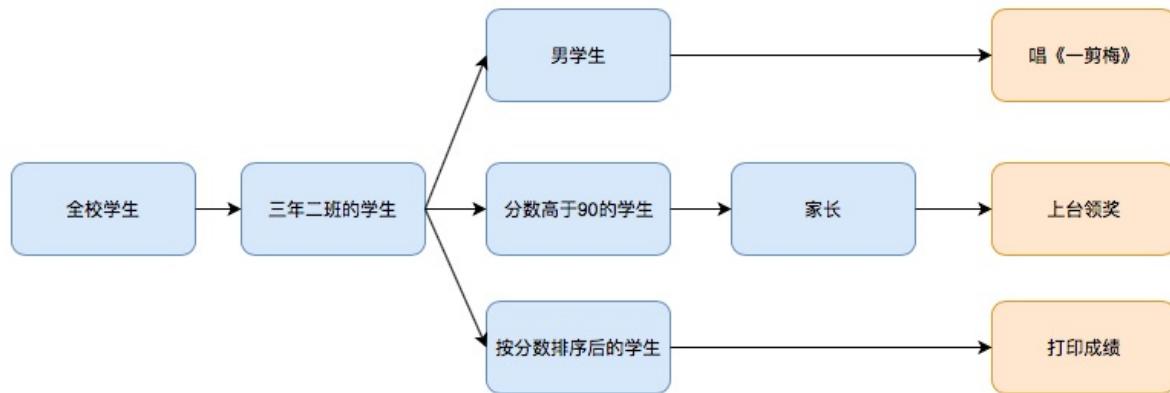
```
// 三年二班学生成绩高于90分的家长上台领奖
gradeThreeClassTwoStudents
    .filter { student in student.score > 90 }
    .map { student in student.parent }
    .forEach { parent in parent.receiveAPrize() }
```

用分数判定来筛选出90分以上的同学，然后用 `map` 转换为学生家长，最后用 `forEach` 让每个家长上台领奖。

```
// 由高到低打印三年二班的学生成绩
gradeThreeClassTwoStudents
    .sorted { student0, student1 in student0.score > student1.score }
    .forEach { student in print("score: \(student.score), name: \(student.name)") }
```

将排序逻辑的函数传递给 `sorted` 方法，这样学生就按成绩高低排序，最后用 `foreach` 将成绩和学生名字打印出来。

整体结构



值得注意的是，我们先从三年二班筛选出男同学，后来又从三年二班筛选出分数高于90的学生。都是用的 `filter` 方法，只是传递了不同的判定函数，从而得出了不同的筛选结果。如果现在要实现这个需求：二年一班分数不足60的学生唱一首《我有罪》。

相信大家要不了多久就可以找到对应的实现方法。

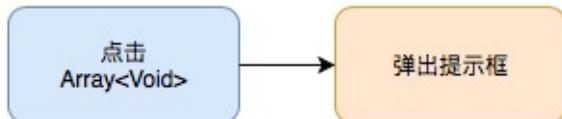
这就是**函数式编程**，它使我们可以通过组合不同的方法，以及不同的函数来获取目标结果。你可以想象如果我们用传统的 `for` 循环来完成相同的逻辑，那将会是一件多么繁琐的事情。所以函数式编程的优点是显而易见的：

- 灵活
- 高复用
- 简洁
- 易维护
- 适应各种需求变化

如果想了解更多有关于**函数式编程**的知识。可以参考这本书籍 [《函数式 Swift》](#)。

函数式编程 -> 函数响应式编程

现在大家已经了解我们是如何运用**函数式编程**来操作序列的。其实我们可以把这种操作序列的方式再升华一下。例如，你可以把一个按钮的点击事件看作是一个序列：



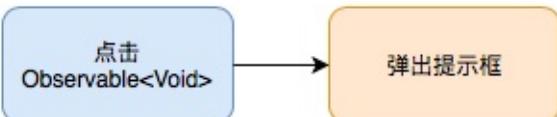
```

// 假设用户在进入页面到离开页面期间，总共点击按钮 3 次

// 按钮点击序列
let taps: Array<Void> = [(), (), ()]

// 每次点击后弹出提示框
taps.forEach { showAlert() }
  
```

这样处理点击事件是非常理想的，但是问题是这个序列里面的元素（点击事件）是异步产生的，传统序列是无法描述这种元素异步产生的情况。为了解决这个问题，于是就产生了**可监听序列 Observable<Element>**。它也是一个序列，只不过这个序列里面的元素可以是同步产生的，也可以是异步产生的：

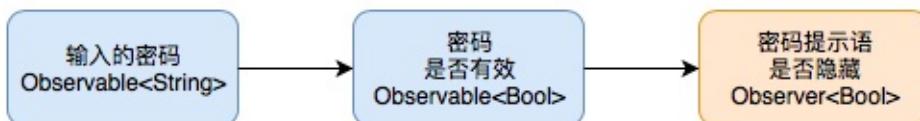


```

// 按钮点击序列
let taps: Observable<Void> = button.rx.tap.asObservable()

// 每次点击后弹出提示框
taps.subscribe(onNext: { showAlert() })
  
```

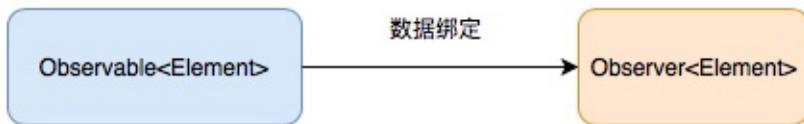
这里 `taps` 就是按钮点击事件的序列。然后我们通过弹出提示框，来对每一次点击事件做出响应。这种编程方式叫做**响应式编程**。我们结合**函数式编程**以及**响应式编程**就得到了**函数响应式编程**：



```
passwordOutlet.rx.text.orEmpty
    .map { $0.characters.count >= minimalPasswordLength }
    .bind(to: passwordValidOutlet.rx.isHidden)
    .disposed(by: disposeBag)
```

我们通过不同的构建函数，来创建所需要的数据序列。最后通过适当的方式来响应这个序列。这就是**函数响应式编程**。

数据绑定（订阅）



在 RxSwift 里有一个比较重要的概念就是数据绑定（订阅）。就是指将可监听序列绑定到观察者上：

我们对比一下这两段代码：

```
let image: UIImage = UIImage(named: ...)
imageView.image = image
```

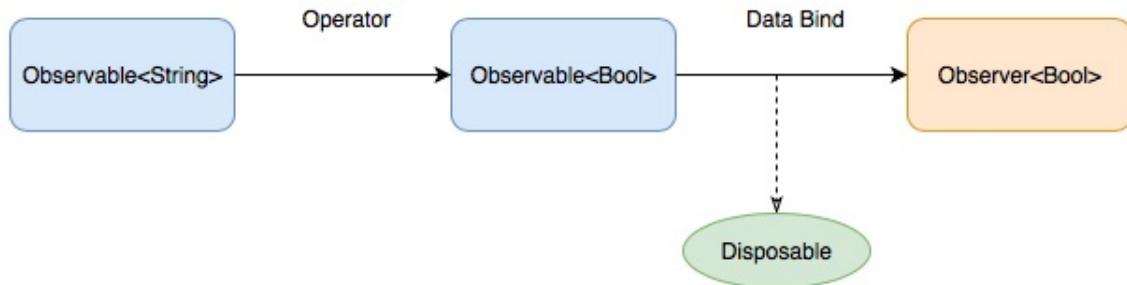
```
let image: Observable<UIImage> = ...
image.bind(to: imageView.rx.image)
```

第一段代码我们非常熟悉，它就是将一个单独的图片设置到 `imageView` 上。

第二段代码则是将一个图片序列“同步”到 `imageView` 上。这个序列里面的图片可以是异步产生的。这里定义的 `image` 就是上图中蓝色部分（可监听序列），`imageView.rx.image` 就是上图中橙色部分（观察者）。而这种“同步机制”就是数据绑定（订阅）。

RxSwift 核心

这一章主要介绍 **RxSwift** 的核心内容：



- **Observable** - 产生事件
- **Observer** - 响应事件
- **Operator** - 创建变化组合事件
- **Disposable** - 管理绑定（订阅）的生命周期
- **Schedulers** - 线程队列调配

```

// Observable<String>
let text = usernameOutlet.rx.text.orEmpty.asObservable()

// Observable<Bool>
let passwordValid = text
    // Operator
    .map { $0.characters.count >= minimalUsernameLength }

// Observer<Bool>
let observer = passwordValidOutlet.rx.isHidden

// Disposable
let disposable = passwordValid
    // Scheduler 用于控制任务在那个线程队列运行
    .subscribeOn(MainScheduler.instance)
    .observeOn(MainScheduler.instance)
    .bind(to: observer)

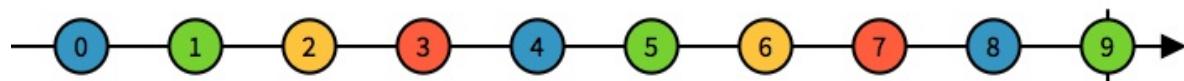
...

// 取消绑定，你可以在退出页面时取消绑定
disposable.dispose()
  
```

下面几节会详细介绍这几个组件的功能和用法。

提示：这一章主要介绍一些偏理论方面的知识。你如果觉得阅读起来比较乏味的话，可以先快速地浏览一遍，了解 **RxSwift** 的核心组件大概有哪些内容。待以后遇到实际问题时，在回来查询。你可以直接跳到 [更多例子](#) 章节，去了解如何应用 **RxSwift**。

Observable - 可监听序列



所有的事物都是序列

之前我们提到，`Observable` 可以用于描述元素异步产生的序列。这样我们生活中许多事物都可以通过它来表示，例如：

- `Observable<Double>` 温度

你可以将温度看作是一个序列，然后监测这个温度值，最后对这个值做出响应。例如：当室温高于 33 度时，打开空调降温。



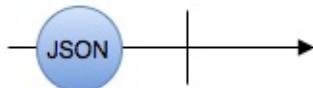
- `Observable<OnePieceEpisode>` 《海贼王》动漫

你也可以把《海贼王》的动漫看作是一个序列。然后当《海贼王》更新一集时，我们就立即观看这一集。



- `Observable<JSON>` JSON

你可以把网络请求的返回的 JSON 看作是一个序列。然后当取到 JSON 时，将它打印出来。



- `Observable<Void>` 任务回调

你可以把任务回调看作是一个序列。当任务结束后，提示用户任务已完成。

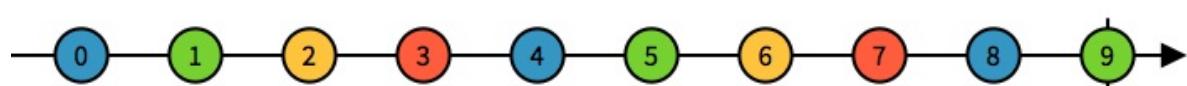


如何创建序列

现在我们已经可以把生活中的许多事物看作是一个序列了。那么我们要怎么创建这些序列呢？

实际上，框架已经帮我们创建好了许多常用的序列。例如：`button` 的点击，`textField` 的当前文本，`switch` 的开关状态，`slider` 的当前数值等等。

另外，有一些自定义的序列是需要我们自己创建的。这里介绍一下创建序列最基本的方法，例如，我们创建一个 `[0, 1, ... 8, 9]` 的序列：



```
let numbers: Observable<Int> = Observable.create { observer -> Disposable in
    observer.onNext(0)
    observer.onNext(1)
    observer.onNext(2)
    observer.onNext(3)
    observer.onNext(4)
    observer.onNext(5)
    observer.onNext(6)
    observer.onNext(7)
    observer.onNext(8)
    observer.onNext(9)
    observer.onCompleted()
}

return Disposables.create()
}
```

创建序列最直接的方法就是调用 `Observable.create`，然后在构建函数里面描述元素的产生过程。
`observer.onNext(0)` 就代表产生了一个元素，他的值是 `0`。后面又产生了 9 个元素分别是 `1, 2, ... 8, 9`。最后，用 `observer.onCompleted()` 表示元素已经全部产生，没有更多元素了。

你可以用这种方式来封装功能组件，例如，闭包回调：



```
typealias JSON = Any
```

```

let json: Observable<JSON> = Observable.create { (observer) -> Disposable in

    let task = URLSession.shared.dataTask(with: ...) { data, _, error in

        guard error == nil else {
            observer.onError(error!)
            return
        }

        guard let data = data,
              let jsonObject = try? JSONSerialization.jsonObject(with: data, options:
.mutableLeaves)
        else {
            observer.onError(DataError.cantParseJSON)
            return
        }

        observer.onNext(jsonObject)
        observer.onCompleted()
    }

    task.resume()

    return Disposables.create { task.cancel() }
}

```

在闭包回调中，如果任务失败，就调用 `observer.onError(error!)`。如果获取到目标元素，就调用 `observer.onNext(jsonObject)`。由于我们的这个序列只有一个元素，所以在成功获取到元素后，就直接调用 `observer.onCompleted()` 来表示任务结束。最后 `Disposables.create { task.cancel() }` 说明如果数据绑定被清除（订阅被取消）的话，就取消网络请求。

这样一来我们就将传统的闭包回调转换成序列了。然后可以用 `subscribe` 方法来响应这个请求的结果：

```

json
    .subscribe(onNext: { json in
        print("取得 json 成功: \(json)")
    }, onError: { error in
        print("取得 json 失败 Error: \(error.localizedDescription)")
    }, onCompleted: {
        print("取得 json 任务成功完成")
    })
    .disposed(by: disposeBag)

```

这里 `subscribe` 后面的 `onNext`，`onError`，`onCompleted` 分别响应我们创建 `json` 时，构建函数里面的 `onNext`，`onError`，`onCompleted` 事件。我们称这些事件为 **Event**：

Event - 事件

```
public enum Event<Element> {
    case next(Element)
    case error(Swift.Error)
    case completed
}
```

- `next` - 序列产生了一个新的元素
- `error` - 创建序列时产生了一个错误，导致序列终止
- `completed` - 序列的所有元素都已经成功产生，整个序列已经完成

你可以合理的利用这些 `Event` 来实现业务逻辑。

决策树

现在我们知道如何用最基本的方法创建序列。你还可参考 [决策树](#) 来选择其他的方式创建序列。

特征序列

我们都知道 **Swift** 是一个强类型语言，而强类型语言相对于弱类型语言的一个优点是更加严谨。我们可以通过类型来判断出，实例有哪些特征。同样的在 **RxSwift** 里面 `Observable` 也存在一些特征序列，这些特征序列可以帮助我们更准确的描述序列。并且它们还可以给我们提供语法糖，让我们能够用更加优雅的方式书写代码，他们分别是：

- [Single](#)
- [Completable](#)
- [Maybe](#)
- [Driver](#)
- [Signal](#)
- [ControlEvent](#)

提示：由于可被观察的序列 (**Observable**) 名字过长，很多时候会增加阅读难度，所以笔者在必要时会将它简写为：序列。

Single

Single 是 `Observable` 的另外一个版本。不像 `Observable` 可以发出多个元素，它要么只能发出一个元素，要么产生一个 `error` 事件。

- 发出一个元素，或一个 `error` 事件
- 不会共享附加作用

一个比较常见的例子就是执行 HTTP 请求，然后返回一个应答或错误。不过你也可以用 **Single** 来描述任何只有一个元素的序列。

如何创建 Single

创建 **Single** 和创建 **Observable** 非常相似：

```
func getRepo(_ repo: String) -> Single<[String: Any]> {
    return Single<[String: Any]>.create { single in
        let url = URL(string: "https://api.github.com/repos/\(repo)")!
        let task = URLSession.shared.dataTask(with: url) {
            data, _, error in

            if let error = error {
                single(.error(error))
                return
            }

            guard let data = data,
                  let json = try? JSONSerialization.jsonObject(with: data, options: .mutableLeaves),
                  let result = json as? [String: Any] else {
                single(.error(DataError.cantParseJSON))
                return
            }

            single(.success(result))
        }

        task.resume()

        return Disposables.create { task.cancel() }
    }
}
```

之后，你可以这样使用 **Single**：

```
getRepo("ReactiveX/RxSwift")
```

```
.subscribe(onSuccess: { json in
    print("JSON: ", json)
}, onError: { error in
    print("Error: ", error)
})
.disposed(by: disposeBag)
```

订阅提供一个 `SingleEvent` 的枚举：

```
public enum SingleEvent<Element> {
    case success(Element)
    case error(Swift.Error)
}
```

- `success` - 产生一个单独的元素
- `error` - 产生一个错误

你同样可以对 `Observable` 调用 `.asSingle()` 方法，将它转换为 **Single**。

Completable

Completable 是 `Observable` 的另外一个版本。不像 `Observable` 可以发出多个元素，它要么只能产生一个 `completed` 事件，要么产生一个 `error` 事件。

- 发出零个元素
- 发出一个 `completed` 事件或者一个 `error` 事件
- 不会共享附加作用

Completable 适用于那种你只关心任务是否完成，而不需要在意任务返回值的情况。它和 `Observable<Void>` 有点相似。

如何创建 Completable

创建 **Completable** 和创建 **Observable** 非常相似：

```
func cacheLocally() -> Completable {
    return Completable.create { completable in
        // Store some data locally
        ...

        guard success else {
            completable(.error(CacheError.failedCaching))
            return Disposables.create {}
        }

        completable(.completed)
        return Disposables.create {}
    }
}
```

之后，你可以这样使用 **Completable**：

```
cacheLocally()
    .subscribe(onCompleted: {
        print("Completed with no error")
    }, onError: { error in
        print("Completed with an error: \(error.localizedDescription)")
    })
    .disposed(by: disposeBag)
```

订阅提供一个 `CompletableEvent` 的枚举：

```
public enum CompletableEvent {
    case error(Swift.Error)
```

```
    case completed  
}
```

- completed - 产生完成事件
- error - 产生一个错误

Maybe

Maybe 是 `Observable` 的另外一个版本。它介于 `Single` 和 `Completable` 之间，它要么只能发出一个元素，要么产生一个 `completed` 事件，要么产生一个 `error` 事件。

- 发出一个元素或者一个 `completed` 事件或者一个 `error` 事件
- 不会共享附加作用

如果你遇到那种可能需要发出一个元素，又可能不需要发出时，就可以使用 **Maybe**。

如何创建 Maybe

创建 **Maybe** 和创建 **Observable** 非常相似：

```
func generateString() -> Maybe<String> {
    return Maybe<String>.create { maybe in
        maybe(.success("RxSwift"))

        // OR

        maybe(.completed)

        // OR

        maybe(.error(error))

    }
}
```

之后，你可以这样使用 **Maybe**：

```
generateString()
    .subscribe(onSuccess: { element in
        print("Completed with element \(element)")
    }, onError: { error in
        print("Completed with an error \(error.localizedDescription)")
    }, onCompleted: {
        print("Completed with no element")
    })
    .disposed(by: disposeBag)
```

你同样可以对 `Observable` 调用 `.asMaybe()` 方法，将它转换为 **Maybe**。

Maybe

Driver

Driver（司机?）是一个精心准备的特征序列。它主要是为了简化 UI 层的代码。不过如果你遇到的序列具有以下特征，你也可以使用它：

- 不会产生 `error` 事件
- 一定在 `MainScheduler` 监听（主线程监听）
- 共享附加作用

这些都是驱动 UI 的序列所具有的特征。

为什么要使用 Driver？

我们举个例子来说明一下，为什么要使用 **Driver**。

这是文档简介页的例子：

```
let results = query.rx.text
    .throttle(0.3, scheduler: MainScheduler.instance)
    .flatMapLatest { query in
        fetchAutoCompleteItems(query)
    }

results
    .map { "($0.count)" }
    .bind(to: resultCount.rx.text)
    .disposed(by: disposeBag)

results
    .bind(to: resultsTableView.rx.items(cellIdentifier: "Cell")) {
        (_, result, cell) in
        cell.textLabel?.text = "(result)"
    }
    .disposed(by: disposeBag)
```

这段代码的主要目的是：

- 取出用户输入稳定后的内容
- 向服务器请求一组结果
- 将返回的结果绑定到两个 UI 元素上：`tableView` 和 显示结果数量的 `label`

那么这里存在什么问题？

- 如果 `fetchAutoCompleteItems` 的序列产生了一个错误（网络请求失败），这个错误将取消所有绑定，当用户输入一个新的关键字时，是无法发起新的网络请求。
- 如果 `fetchAutoCompleteItems` 在后台返回序列，那么刷新页面也会在后台进行，这样就会出现异常崩溃。

- 返回的结果被绑定到两个 UI 元素上。那就意味着，每次用户输入一个新的关键字时，就会分别为两个 UI 元素发起 HTTP 请求，这并不是我们想要的结果。

一个更好的方案是这样的：

```

let results = query.rx.text
    .throttle(0.3, scheduler: MainScheduler.instance)
    .flatMapLatest { query in
        fetchAutoCompleteItems(query)
            .observeOn(MainScheduler.instance) // 结果在主线程返回
            .catchErrorJustReturn([])
    }
    .share(replay: 1) // HTTP 请求是被共享的

results
    .map { "\($0.count)" }
    .bind(to: resultCount.rx.text)
    .disposed(by: disposeBag)

results
    .bind(to: resultsTableView.rx.items(cellIdentifier: "Cell")) {
        (_, result, cell) in
        cell.textLabel?.text = "\(result)"
    }
    .disposed(by: disposeBag)

```

在一个大型系统内，要确保每一步不被遗漏是一件不太容易的事情。所以更好的选择是合理运用编译器和特征序列来确保这些必备条件都已经满足。

以下是使用 **Driver** 优化后的代码：

```

let results = query.rx.text.asDriver() // 将普通序列转换为 Driver
    .throttle(0.3, scheduler: MainScheduler.instance)
    .flatMapLatest { query in
        fetchAutoCompleteItems(query)
            .asDriver(onErrorJustReturn: []) // 仅仅提供发生错误时的备选返回值
    }

results
    .map { "\($0.count)" }
    .drive(resultCount.rx.text) // 这里改用 `drive` 而不是 `bindTo`、
    .disposed(by: disposeBag) // 这样可以确保必备条件都已经满足了

results
    .drive(resultsTableView.rx.items(cellIdentifier: "Cell")) {
        (_, result, cell) in
        cell.textLabel?.text = "\(result)"
    }
    .disposed(by: disposeBag)

```

首先第一个 `asDriver` 方法将 `ControlProperty` 转换为 `Driver`

然后第二个变化是：

```
.asDriver(onErrorJustReturn: [])
```

任何可监听序列都可以被转换为 `Driver`，只要他满足 3 个条件：

- 不会产生 `error` 事件
- 一定在 `MainScheduler` 监听（主线程监听）
- [共享附加作用](#)

那么要如何确定条件都被满足？通过 Rx 操作符来进行转换。`asDriver(onErrorJustReturn: [])` 相当于以下代码：

```
let safeSequence = xs
    .observeOn(MainScheduler.instance)          // 主线程监听
    .catchErrorJustReturn(onErrorJustReturn) // 无法产生错误
    .share(replay: 1, scope: .whileConnected)// 共享附加作用
return Driver(raw: safeSequence)           // 封装
```

最后使用 `drive` 而不是 `bindTo`

`drive` 方法只能被 `Driver` 调用。这意味着，如果你发现代码所存在 `drive`，那么这个序列不会产生错误事件并且一定在主线程监听。这样你可以安全的绑定 UI 元素。

Signal

Signal 和 **Driver** 相似，唯一的区别是，**Driver** 会对新观察者回放（重新发送）上一个元素，而 **Signal** 不会对新观察者回放上一个元素。

他有如下特性：

- 不会产生 `error` 事件
- 一定在 `MainScheduler` 监听（主线程监听）
- [共享附加作用](#)

现在，我们来看看以下代码是否合理：

```
let textField: UITextField = ...
let nameLabel: UILabel = ...
let nameSizeLabel: UILabel = ...

let state: Driver<String?> = textField.rx.text.asDriver()

let observer = nameLabel.rx.text
state.drive(observer)

// ... 假设以下代码是在用户输入姓名后运行

let newObserver = nameSizeLabel.rx.text
state.map { $0?.count.description }.drive(newObserver)
```

这个例子只是将用户输入的姓名绑定到对应的标签上。当用户输入姓名后，我们创建了一个新的观察者，用于订阅姓名的字数。那么问题来了，订阅时，展示字数的标签会立即更新吗？

嗯、、、因为 **Driver** 会对新观察者回放上一个元素（当前姓名），所以这里是会更新的。在对他进行订阅时，标签的默认文本会被刷新。这是合理的。

那如果我们用 **Driver** 来描述点击事件呢，这样合理吗？

```
let button: UIButton = ...
let showAlert: (String) -> Void = ...

let event: Driver<Void> = button.rx.tap.asDriver()

let observer: () -> Void = { showAlert("弹出提示框1") }
event.drive(onNext: observer)

// ... 假设以下代码是在用户点击 button 后运行

let newObserver: () -> Void = { showAlert("弹出提示框2") }
event.drive(onNext: newObserver)
```

当用户点击一个按钮后，我们创建一个新的观察者，来响应点击事件。此时会发生什么？[Driver](#) 会把上一次的点击事件回放给新观察者。所以，这里的 `newObserver` 在订阅时，就会接受到上次的点击事件，然后弹出提示框。这似乎不太合理。

因此像这类型的事件序列，用 [Driver](#) 建模就不合适。于是我们就引入了 **Signal**：

```
...
let event: Signal<Void> = button.rx.tap.asSignal()

let observer: () -> Void = { showAlert("弹出提示框1") }
event.emit(onNext: observer)

// ... 假设以下代码是在用户点击 button 后运行

let newObserver: () -> Void = { showAlert("弹出提示框2") }
event.emit(onNext: newObserver)
```

在同样的场景中，**Signal** 不会把上一次的点击事件回放给新观察者，而只会将订阅后产生的点击事件，发布给新观察者。这正是我们所需要的。

结论

一般情况下状态序列我们会选用 [Driver](#) 这个类型，事件序列我们会选用 **Signal** 这个类型。

参考

- [Driver](#)
- [map](#)
- [共享附加作用](#)

ControlEvent

ControlEvent 专门用于描述 UI 控件所产生的事件，它具有以下特征：

- 不会产生 `error` 事件
- 一定在 `MainScheduler` 订阅（主线程订阅）
- 一定在 `MainScheduler` 监听（主线程监听）
- **共享附加作用**

Observer - 观察者



观察者 是用来监听事件，然后它需要这个事件做出响应。例如：弹出提示框就是观察者，它对点击按钮这个事件做出响应。

响应事件的都是观察者

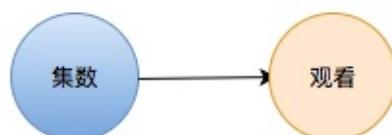
在 [Observable](#) 章节，我们举了个几个例子来介绍什么是可监听序列。那么我们还是用这几个例子来解释一下什么是观察者：

- 当室温高于 33 度时，打开空调降温



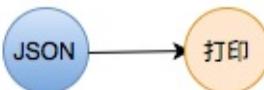
打开空调降温就是观察者 `Observer<Double>`。

- 当《海贼王》更新一集时，我们就立即观看这一集



观看这一集就是观察者 `Observer<OnePieceEpisode>`。

- 当取到 JSON 时，将它打印出来



将它打印出来就是观察者 `Observer<JSON>`

- 当任务结束后，提示用户任务已完成



提示用户任务已完成就是观察者 `Observer<Void>`

如何创建观察者

现在我们已经知道**观察者**主要是做什么的了。那么我们要怎么创建它们呢？

和 **Observable** 一样，框架已经帮我们创建好了许多常用的**观察者**。例如：`view` 是否隐藏，`button` 是否可点击，`label` 的当前文本，`imageView` 的当前图片等等。

另外，有一些自定义的**观察者**是需要我们自己创建的。这里介绍一下创建**观察者**最基本的方法，例如，我们创建一个弹出提示框的**观察者**：



```
tap.subscribe(onNext: { [weak self] in
    self?.showAlert()
}, onError: { error in
    print("发生错误: \(error.localizedDescription)")
}, onCompleted: {
    print("任务完成")
})
```

创建**观察者**最直接的方法就是在 `Observable` 的 `subscribe` 方法后面描述，事件发生时，需要如何做出响应。而**观察者**就是由后面的 `onNext`，`onError`，`onCompleted` 的这些闭包构建出来的。

以上是创建**观察者**最常见的方法。当然你还可以通过其他的方式来创建**观察者**，可以参考一下 [AnyObserver](#) 和 [Binder](#)。

特征观察者

和 **Observable** 一样，**观察者**也存**特征观察者**，例如：

- [Binder](#)

AnyObserver

AnyObserver 可以用来描述任意一种观察者。

例如：

打印网络请求结果：

```
URLSession.shared.rx.data(request: URLRequest(url: url))
    .subscribe(onNext: { data in
        print("Data Task Success with count: \(data.count)")
    }, onError: { error in
        print("Data Task Error: \(error)")
    })
    .disposed(by: disposeBag)
```

可以看作是：

```
let observer: AnyObserver<Data> = AnyObserver { (event) in
    switch event {
        case .next(let data):
            print("Data Task Success with count: \(data.count)")
        case .error(let error):
            print("Data Task Error: \(error)")
        default:
            break
    }
}

URLSession.shared.rx.data(request: URLRequest(url: url))
    .subscribe(observer)
    .disposed(by: disposeBag)
```

用户名提示语是否隐藏：

```
usernameValid
    .bind(to: usernameValidOutlet.rx.isHidden)
    .disposed(by: disposeBag)
```

可以看作是：

```
let observer: AnyObserver<Bool> = AnyObserver { [weak self] (event) in
```

```
switch event {
    case .next(let isHidden):
        self?.usernameValidOutlet.isHidden = isHidden
    default:
        break
}
}

usernameValid
    .bind(to: observer)
    .disposed(by: disposeBag)
```

下一节将介绍 [Binder](#) 以及 `usernameValidOutlet.rx.isHidden` 的由来。

Binder

Binder 主要有以下两个特征:

- 不会处理错误事件
- 确保绑定都是在给定 `Scheduler` 上执行（默认 `MainScheduler`）

一旦产生错误事件，在调试环境下将执行 `fatalError`，在发布环境下将打印错误信息。

示例

在介绍 `AnyObserver` 时，我们举了这样一个例子：

```
let observer: AnyObserver<Bool> = AnyObserver { [weak self] (event) in
    switch event {
        case .next(let isHidden):
            self?.usernameValidOutlet.isHidden = isHidden
        default:
            break
    }
}

usernameValid
    .bind(to: observer)
    .disposed(by: disposeBag)
```

由于这个观察者是一个 **UI 观察者**，所以它在响应事件时，只会处理 `next` 事件，并且更新 **UI** 的操作需要在主线程上执行。

因此一个更好的方案就是使用 **Binder**:

```
let observer: Binder<Bool> = Binder(usernameValidOutlet) { (view, isHidden) in
    view.isHidden = isHidden
}

usernameValid
    .bind(to: observer)
    .disposed(by: disposeBag)
```

Binder 可以只处理 `next` 事件，并且保证响应 `next` 事件的代码一定会在给定 `Scheduler` 上执行，这里采用默认的 `MainScheduler`。

复用

由于页面是否隐藏是一个常用的观察者，所以应该让所有的 `UIView` 都提供这种观察者：

```
extension Reactive where Base: UIView {
    public var isHidden: Binder<Bool> {
        return Binder(self.base) { view, hidden in
            view.isHidden = hidden
        }
    }
}
```

```
usernameValid
    .bind(to: usernameValidOutlet.rx.isHidden)
    .disposed(by: disposeBag)
```

这样你不必为每个 **UI** 控件单独创建该观察者。这就是 `usernameValidOutlet.rx.isHidden` 的由来，许多 **UI 观察者** 都是这样创建的：

- 按钮是否可点击 `button.rx.isEnabled` :

```
extension Reactive where Base: UIControl {
    public var isEnabled: Binder<Bool> {
        return Binder(self.base) { control, value in
            control.isEnabled = value
        }
    }
}
```

- `label` 的当前文本 `label.rx.text` :

```
extension Reactive where Base: UILabel {
    public var text: Binder<String?> {
        return Binder(self.base) { label, text in
            label.text = text
        }
    }
}
```

你也可以用这种方式来创建自定义的 **UI 观察者**。

Observable & Observer 既是可监听序列也是观察者



在我们所遇到的事物中，有一部分非常特别。它们既是可监听序列也是观察者。

例如： `textField` 的当前文本。它可以看成是由用户输入，而产生的一个文本序列。也可以是由外部文本序列，来控制当前显示内容的观察者：

```
// 作为可监听序列
let observable = textField.rx.text
observable.subscribe(onNext: { text in show(text: text) })
```

```
// 作为观察者
let observer = textField.rx.text
let text: Observable<String?> = ...
text.bind(to: observer)
```

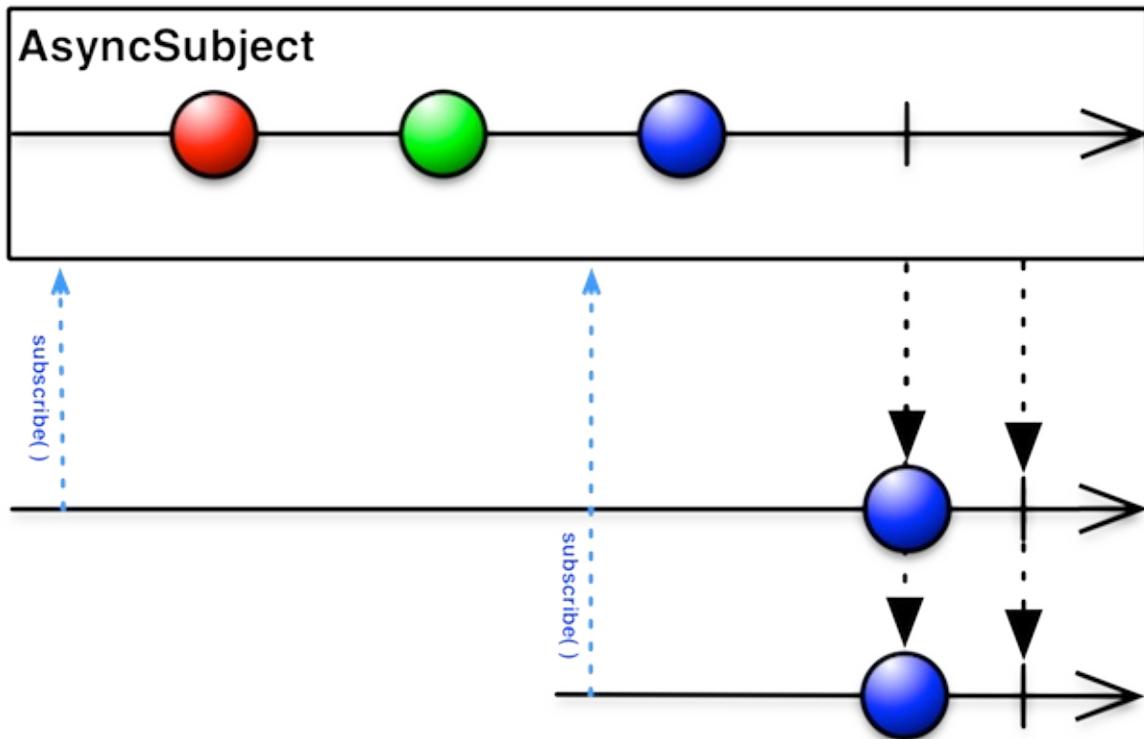
有许多 UI 控件都存在这种特性，例如：`switch` 的开关状态，`segmentedControl` 的选中索引号，`datePicker` 的选中日期等等。

参考

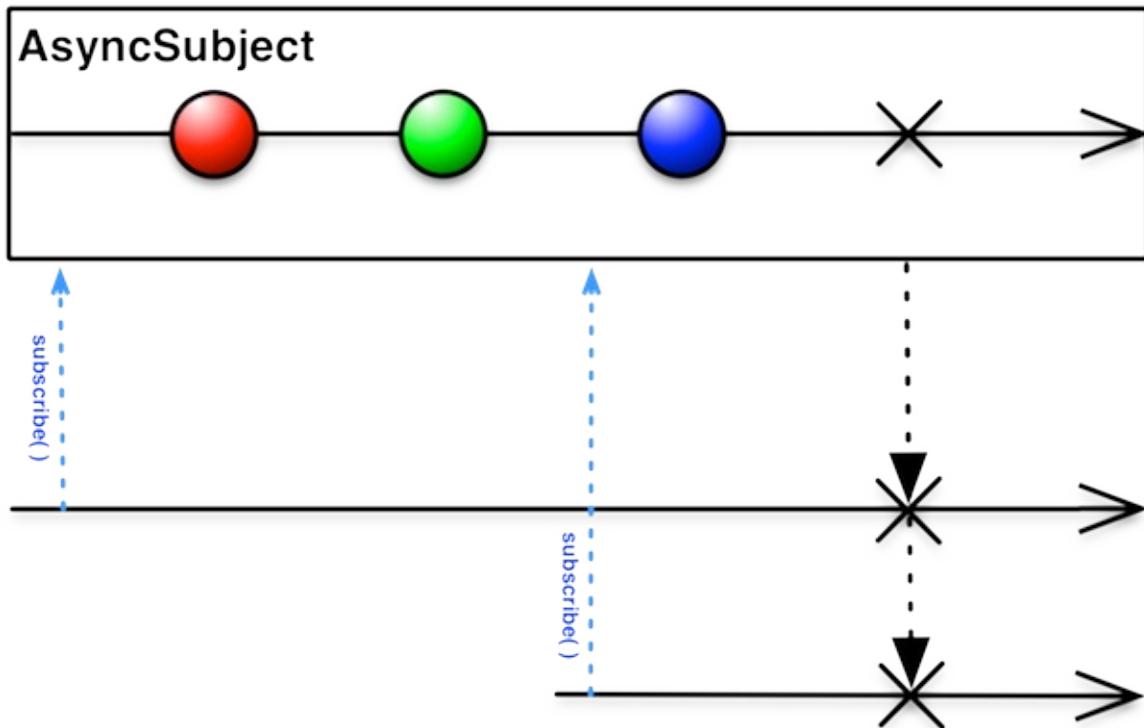
另外，框架里面定义了一些辅助类型，它们既是可监听序列也是观察者。如果你能合适的应用这些辅助类型，它们就可以帮助你更准确的描述事物的特征：

- [AsyncSubject](#)
- [PublishSubject](#)
- [ReplaySubject](#)
- [BehaviorSubject](#)
- [ControlProperty](#)

AsyncSubject



AsyncSubject 将在源 Observable 产生完成事件后，发出最后一个元素（仅仅只有最后一个元素），如果源 Observable 没有发出任何元素，只有一个完成事件。那 **AsyncSubject** 也只有一个完成事件。



它会对随后的观察者发出最终元素。如果源 `Observable` 因为产生了一个 `error` 事件而中止,
AsyncSubject 就不会发出任何元素，而是将这个 `error` 事件发送出来。

演示

```
let disposeBag = DisposeBag()
let subject = AsyncSubject<String>()

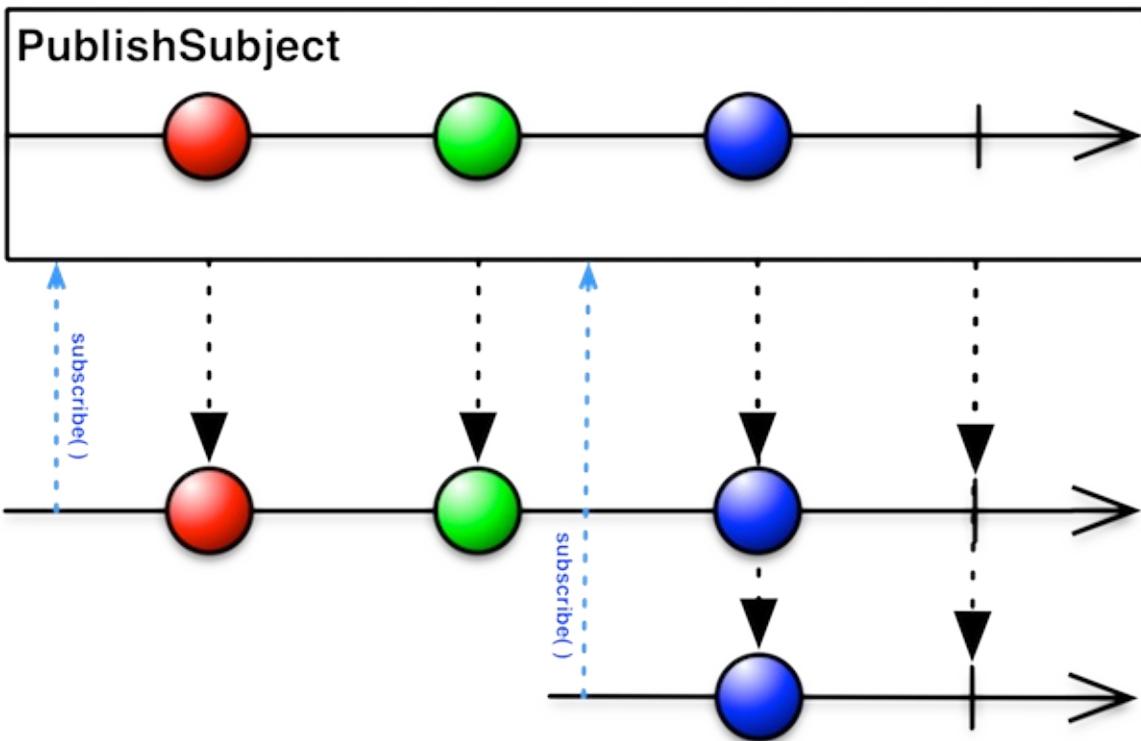
subject
    .subscribe { print("Subscription: 1 Event:", $0) }
    .disposed(by: disposeBag)

subject.onNext(" ")
subject.onNext(" ")
subject.onNext(" ")
subject.onCompleted()
```

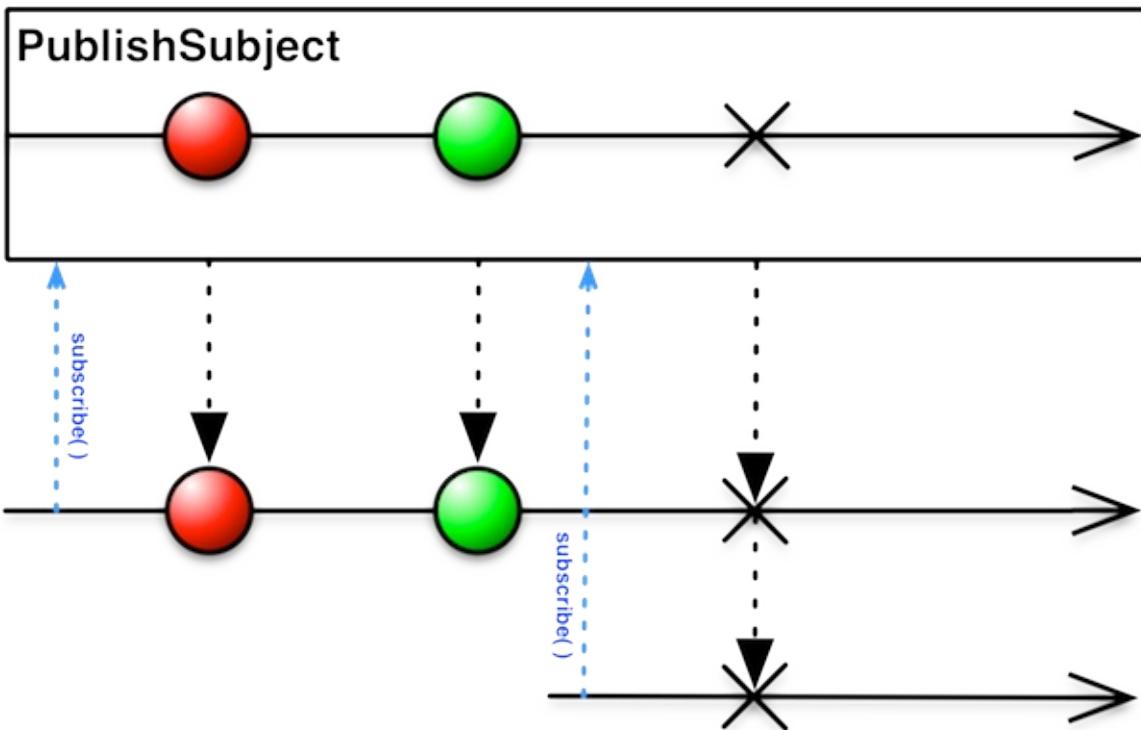
输出结果：

```
Subscription: 1 Event: next( )
Subscription: 1 Event: completed
```

PublishSubject



PublishSubject 将对观察者发送订阅后产生的元素，而在订阅前发出的元素将不会发送给观察者。如果你希望观察者接收到所有的元素，你可以通过使用 `Observable` 的 `create` 方法来创建 `Observable`，或者使用 [ReplaySubject](#)。



如果源 `Observable` 因为产生了一个 `error` 事件而中止, `PublishSubject` 就不会发出任何元素, 而是将这个 `error` 事件发送出来。

演示

```
let disposeBag = DisposeBag()
let subject = PublishSubject<String>()

subject
    .subscribe { print("Subscription: 1 Event:", $0) }
    .disposed(by: disposeBag)

subject.onNext(" ")
subject.onNext("")

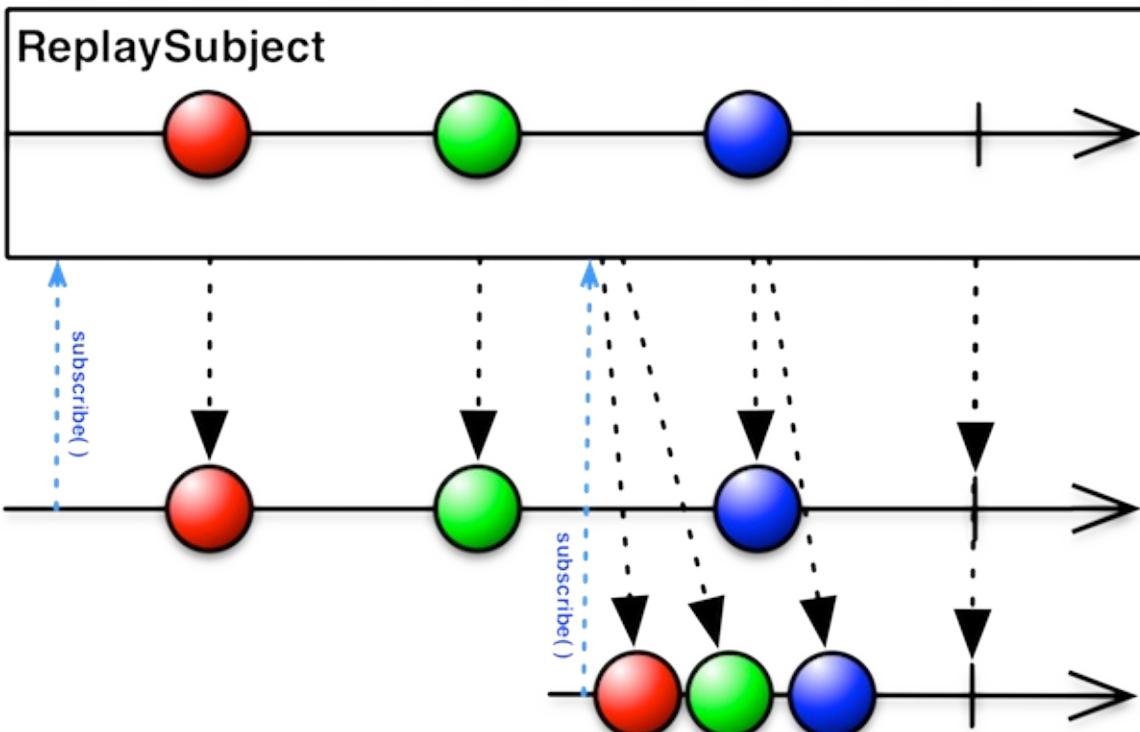
subject
    .subscribe { print("Subscription: 2 Event:", $0) }
    .disposed(by: disposeBag)

subject.onNext("A")
subject.onNext("B")
```

输出结果:

```
Subscription: 1 Event: next( )
Subscription: 1 Event: next( )
Subscription: 1 Event: next(A)
Subscription: 2 Event: next(A)
Subscription: 1 Event: next(B)
Subscription: 2 Event: next(B)
```

ReplaySubject



ReplaySubject 将对观察者发送全部的元素，无论观察者是何时进行订阅的。

这里存在多个版本的 **ReplaySubject**，有的只会将最新的 n 个元素发送给观察者，有的只会将限制时间段内最新的元素发送给观察者。

如果把 **ReplaySubject** 当作观察者来使用，注意不要在多个线程调用 `onNext`，`onError` 或 `onCompleted`。这样会导致无序调用，将造成意想不到的结果。

演示

```
let disposeBag = DisposeBag()
let subject = ReplaySubject<String>.create(bufferSize: 1)

subject
    .subscribe { print("Subscription: 1 Event:", $0) }
    .disposed(by: disposeBag)

subject.onNext(" ")
subject.onNext(" ")

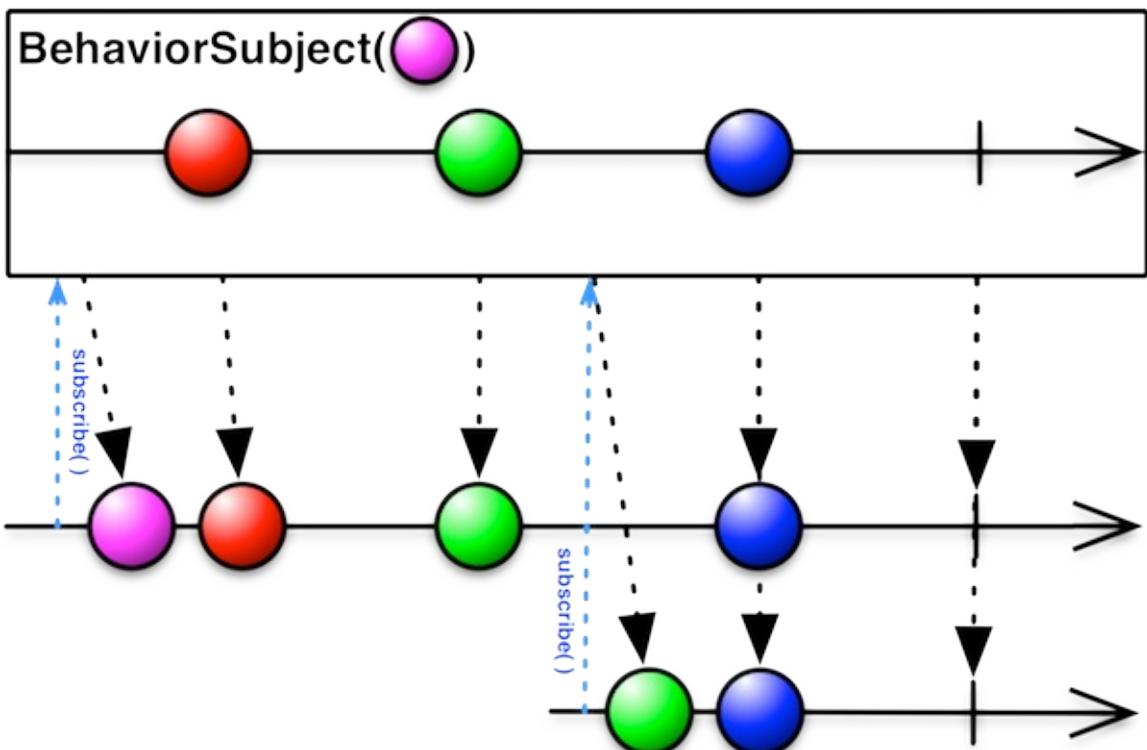
subject
    .subscribe { print("Subscription: 2 Event:", $0) }
    .disposed(by: disposeBag)
```

```
subject.onNext("A")
subject.onNext("B")
```

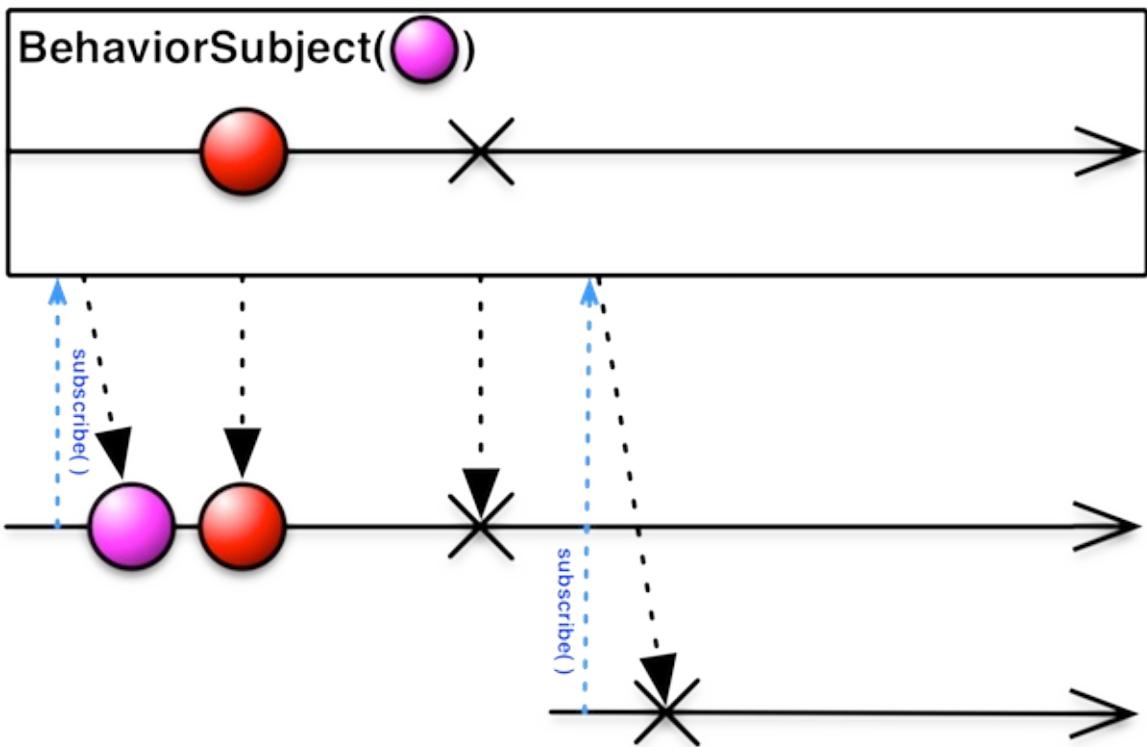
输出结果：

```
Subscription: 1 Event: next()
Subscription: 1 Event: next()
Subscription: 2 Event: next()
Subscription: 1 Event: next(A)
Subscription: 2 Event: next(A)
Subscription: 1 Event: next(B)
Subscription: 2 Event: next(B)
```

BehaviorSubject



当观察者对 **BehaviorSubject** 进行订阅时，它会将源 `Observable` 中最新的元素发送出来（如果不存在最新的元素，就发出默认元素）。然后将随后产生的元素发送出来。



如果源 `Observable` 因为产生了一个 `error` 事件而中止，**BehaviorSubject** 就不会发出任何元素，而是将这个 `error` 事件发送出来。

演示

```
let disposeBag = DisposeBag()
let subject = BehaviorSubject(value: " ")

subject
    .subscribe { print("Subscription: 1 Event:", $0) }
    .disposed(by: disposeBag)

subject.onNext(" ")
subject.onNext(" ")

subject
    .subscribe { print("Subscription: 2 Event:", $0) }
    .disposed(by: disposeBag)

subject.onNext("A")
subject.onNext("B")

subject
    .subscribe { print("Subscription: 3 Event:", $0) }
    .disposed(by: disposeBag)

subject.onNext(" ")
subject.onNext(" ")
```

输出结果：

```
Subscription: 1 Event: next( )
Subscription: 1 Event: next( )
Subscription: 1 Event: next( )
Subscription: 2 Event: next( )
Subscription: 1 Event: next(A)
Subscription: 2 Event: next(A)
Subscription: 1 Event: next(B)
Subscription: 2 Event: next(B)
Subscription: 3 Event: next(B)
Subscription: 1 Event: next( )
Subscription: 2 Event: next( )
Subscription: 3 Event: next( )
Subscription: 1 Event: next( )
Subscription: 2 Event: next( )
Subscription: 3 Event: next( )
```


Variable (已弃用)

Variable 是早期添加到 RxSwift 的概念，通过“setting”和“getting”，他可以帮助我们从原先命令式的思维方式，过渡到响应式的思维方式。

但这只是我们一厢情愿的想法。许多开发者滥用 **Variable**，来构建 **重度命令式** 系统，而不是 Rx 的 **声明式** 系统。这对于新手很常见，并且他们无法意识到，这是代码的坏味道。所以在 RxSwift 4.x 中 **Variable** 被轻度弃用，仅仅给出一个运行时警告。

在 RxSwift 5.x 中，他被[官方的正式的弃用了](#)，并且在需要时，推荐使用 `BehaviorRelay` 或者 `BehaviorSubject`。

ControlProperty

ControlProperty 专门用于描述 UI 控件属性的，它具有以下特征：

- 不会产生 `error` 事件
- 一定在 `MainScheduler` 订阅（主线程订阅）
- 一定在 `MainScheduler` 监听（主线程监听）
- **共享附加作用**

Operator - 操作符

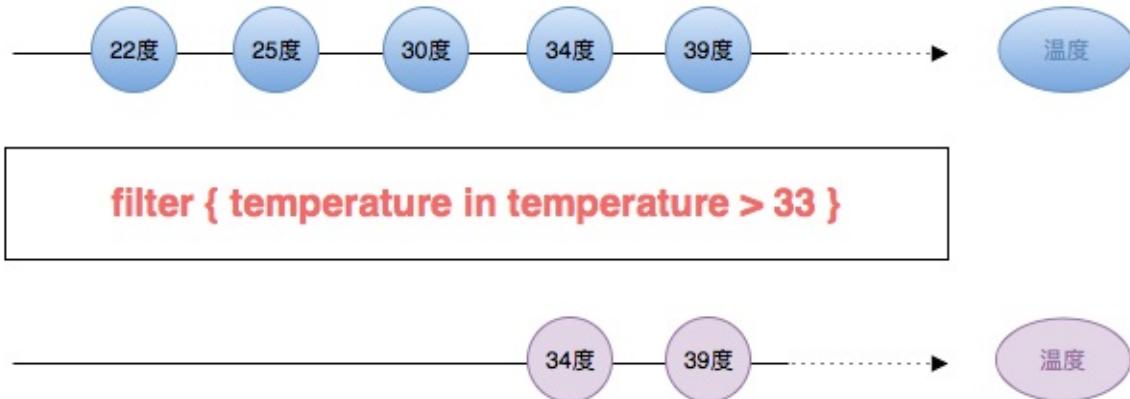


操作符可以帮助大家创建新的序列，或者变化组合原有的序列，从而生成一个新的序列。

我们之前在[输入验证](#)例子中就多次运用到操作符。例如，通过 `map` 方法将输入的用户名，转换为用户名是否有效。然后用这个转化后来的序列来控制红色提示语是否隐藏。我们还通过 `combineLatest` 方法，将用户名是否有效和密码是否有效合并成两者是否同时有效。然后用这个合成后来的序列来控制按钮是否可点击。

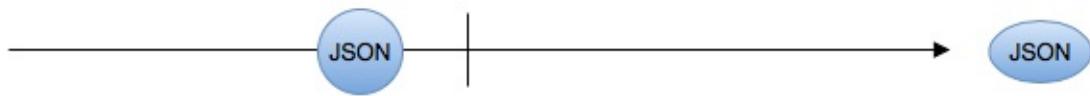
这里 `map` 和 `combineLatest` 都是操作符，它们可以帮助我们构建所需要的序列。现在，我们再来看几个例子：

filter - 过滤



你可以用 `filter` 创建一个新的序列。这个序列只发出温度大于 33 度的元素。

map - 转换

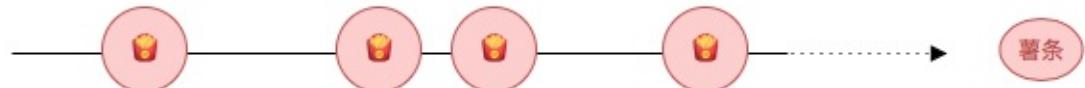
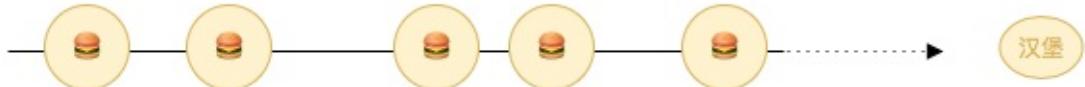


map(Model.init)

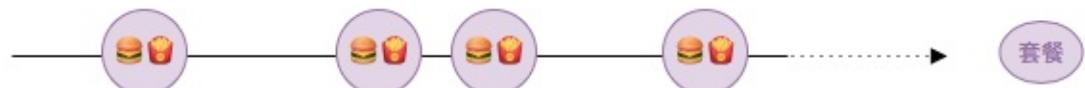


你可以用 `map` 创建一个新的序列。这个序列将原有的 **JSON** 转换成 **Model**。这种转换实际上就是解析 **JSON**。

zip - 配对



zip(hamburg, frenchFries)



你可以用 `zip` 来合成一个新的序列。这个序列将汉堡序列的元素和薯条序列的元素配对后，生成一个新的套餐序列。

如何使用操作符

使用操作符是非常容易的。你可以直接调用实例方法，或者静态方法：

- 温度过滤

```
// 温度
let rxTemperature: Observable<Double> = ...

// filter 操作符
```

```
rxTemperature.filter { temperature in temperature > 33 }
    .subscribe(onNext: { temperature in
        print("高温: \(temperature)度")
    })
    .disposed(by: disposeBag)
```

- 解析 JSON

```
// JSON
let json: Observable<JSON> = ...

// map 操作符
json.map(Model.init)
    .subscribe(onNext: { model in
        print("取得 Model: \(model)")
    })
    .disposed(by: disposeBag)
```

- 合成套餐

```
// 汉堡
let rxHamburg: Observable<Hamburg> = ...
// 薯条
let rxFrenchFries: Observable<FrenchFries> = ...

// zip 操作符
Observable.zip(rxHamburg, rxFrenchFries)
    .subscribe(onNext: { (hamburg, frenchFries) in
        print("取得汉堡: \(hamburg) 和薯条: \(frenchFries)")
    })
    .disposed(by: disposeBag)
```

决策树

Rx 提供了充分的**操作符**来帮我们创建序列。当然如果内置操作符无法满足你的需求时，你还可以创建自定义的操作符。

如果你不确定该如何选择操作符，可以参考 [决策树](#)。它会引导你找出合适的操作符。

操作符列表

26个英文字母我都认识，可是连成一个句子我就不怎么认得了...

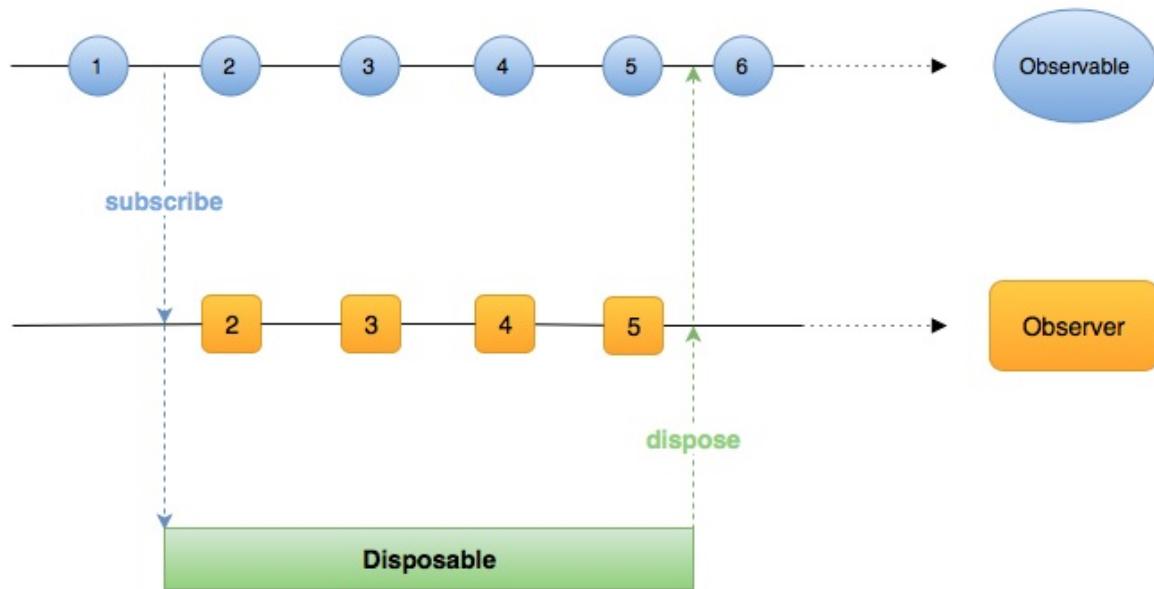
这里提供一个操作符列表，它们就好比是**26个英文字母**。你如果要将它们的作用全部都发挥出来，是需要学习如何将它们连成一个句子的：

- [amb](#)
- [buffer](#)

- catchError
- combineLatest
- concat
- concatMap
- connect
- create
- debounce
- debug
- deferred
- delay
- delaySubscription
- dematerialize
- distinctUntilChanged
- do
- elementAt
- empty
- error
- filter
- flatMap
- flatMapLatest
- from
- groupBy
- ignoreElements
- interval
- just
- map
- merge
- materialize
- never
- observeOn
- publish
- reduce
- refCount
- repeatElement
- replay
- retry
- sample
- scan
- shareReplay
- single
- skip
- skipUntil
- skipWhile
- startWith

- [subscribeOn](#)
- [take](#)
- [takeLast](#)
- [takeUntil](#)
- [takeWhile](#)
- [timeout](#)
- [timer](#)
- [using](#)
- [window](#)
- [withLatestFrom](#)
- [zip](#)

Disposable - 可被清除的资源



通常来说，一个序列如果发出了 `error` 或者 `completed` 事件，那么所有内部资源都会被释放。如果你需要提前释放这些资源或取消订阅的话，那么你可以对返回的 可被清除的资源（**Disposable**）调用 `dispose` 方法：

```
var disposable: Disposable?

override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

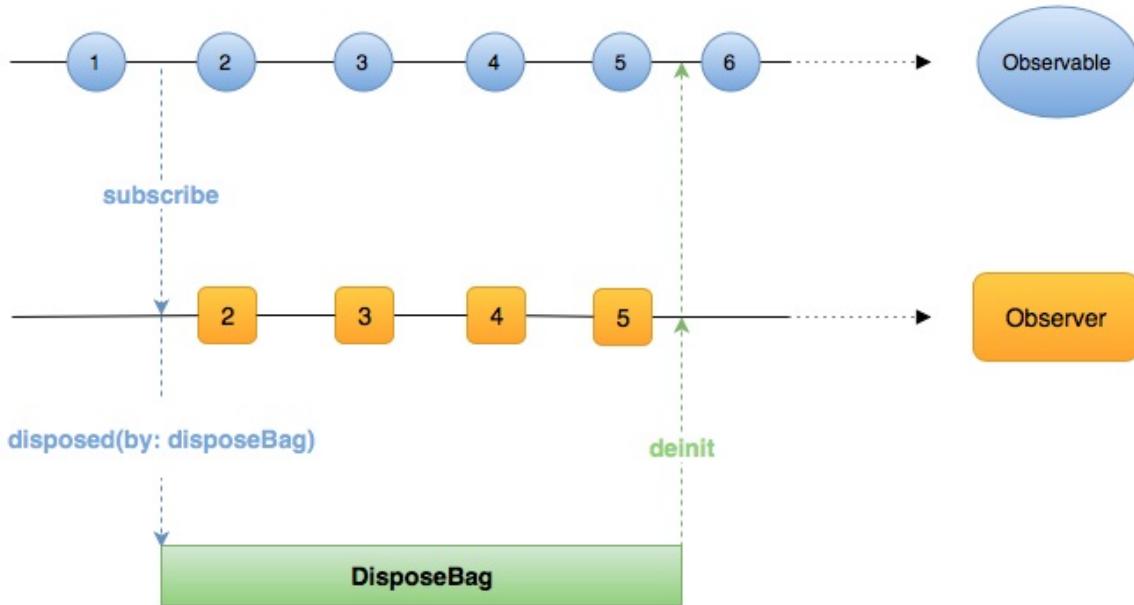
    self.disposable = textField.rx.text.orEmpty
        .subscribe(onNext: { text in print(text) })
}

override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)

    self.disposable?.dispose()
}
```

调用 `dispose` 方法后，订阅将被取消，并且内部资源都会被释放。通常情况下，你是不需要手动调用 `dispose` 方法的，这里只是做个演示而已。我们推荐使用 清除包（**DisposeBag**）或者 **takeUntil** 操作符来管理订阅的生命周期。

DisposeBag - 清除包



因为我们用的是 **Swift**，所以我们更习惯于使用 **ARC** 来管理内存。那么我们能不能用 **ARC** 来管理订阅的生命周期了。答案是肯定了，你可以用 **清除包（DisposeBag）** 来实现这种订阅管理机制：

```
var disposeBag = DisposeBag()

override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

    textField.rx.text.orEmpty
        .subscribe(onNext: { text in print(text) })
        .disposed(by: self.disposeBag)
}

override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)

    self.disposeBag = DisposeBag()
}
```

当 **清除包** 被释放的时候，**清除包** 内部所有 **可被清除的资源（Disposable）** 都将被清除。在[输入验证](#)中我们也多次看到 **清除包** 的身影：

```
var disposeBag = DisposeBag() // 来自父类 ViewController

override func viewDidLoad() {
    super.viewDidLoad()

    ...

    usernameValid
        .bind(to: passwordOutlet.rx.isEnabled)
```

```

    .disposed(by: disposeBag)

    usernameValid
        .bind(to: usernameValidOutlet.rx.isHidden)
        .disposed(by: disposeBag)

    passwordValid
        .bind(to: passwordValidOutlet.rx.isHidden)
        .disposed(by: disposeBag)

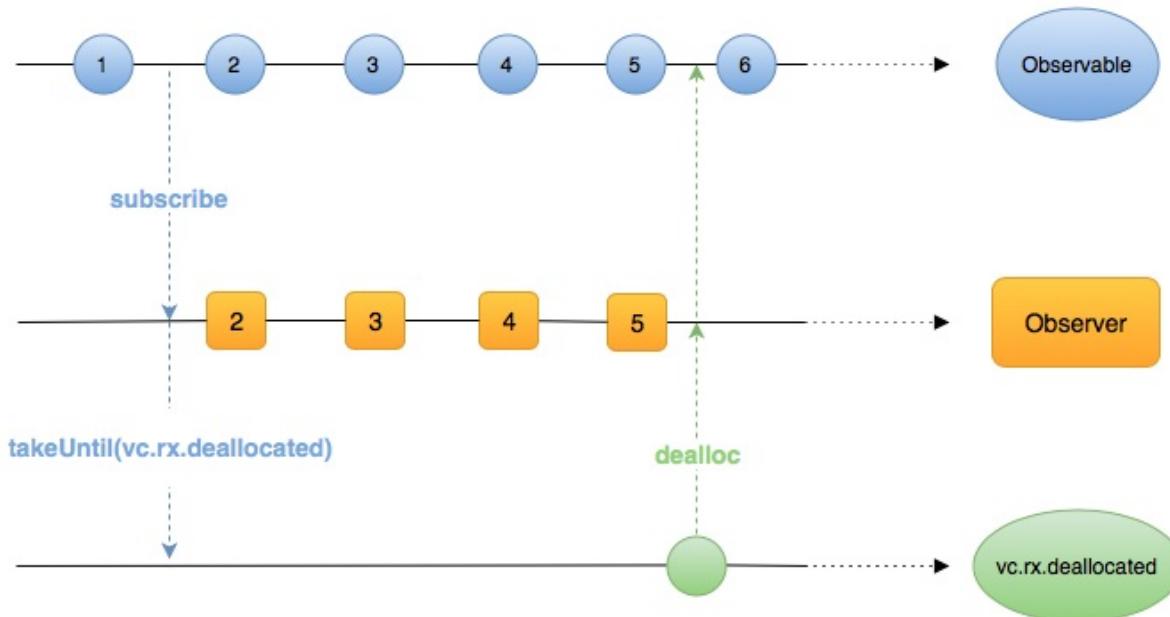
    everythingValid
        .bind(to: doSomethingOutlet.rx.isEnabled)
        .disposed(by: disposeBag)

    doSomethingOutlet.rx.tap
        .subscribe(onNext: { [weak self] in self?.showAlert() })
        .disposed(by: disposeBag)
}

```

这个例子中 `disposeBag` 和 `ViewController` 具有相同的生命周期。当退出页面时，`ViewController` 就被释放，`disposeBag` 也跟着被释放了，那么这里的 5 次绑定（订阅）也就被取消了。这正是我们所需要的。

takeUntil



另外一种实现自动取消订阅的方法就是使用 `takeUntil` 操作符，上面那个[输入验证](#)的演示代码也可以通过使用 `takeUntil` 来实现：

```

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

```

```
...
    ...
    _ = usernameValid
        .takeUntil(self.rx.deallocated)
        .bind(to: passwordOutlet.rx.isEnabled)

    _ = usernameValid
        .takeUntil(self.rx.deallocated)
        .bind(to: usernameValidOutlet.rx.isHidden)

    _ = passwordValid
        .takeUntil(self.rx.deallocated)
        .bind(to: passwordValidOutlet.rx.isHidden)

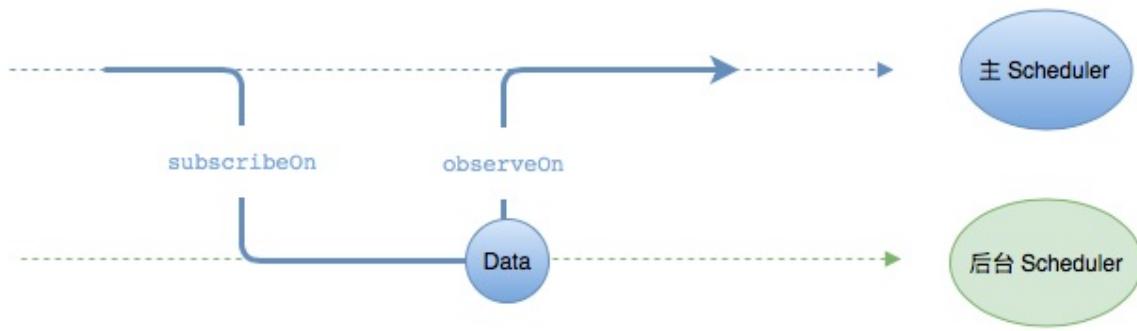
    _ = everythingValid
        .takeUntil(self.rx.deallocated)
        .bind(to: doSomethingOutlet.rx.isEnabled)

    _ = doSomethingOutlet.rx.tap
        .takeUntil(self.rx.deallocated)
        .subscribe(onNext: { [weak self] in self?.showAlert() })
}
}
```

这将使得订阅一直持续到控制器的 **dealloc** 事件产生为止。

注意⚠：这里配图中所使用的 `Observable` 都是“热” `Observable`，它可以帮助我们理解订阅的生命周期。如果你想要了解“冷热” `Observable` 之间的区别，可以参考官方文档 [Hot and Cold Observables](#)。

Schedulers - 调度器



Schedulers 是 Rx 实现多线程的核心模块，它主要用于控制任务在哪个线程或队列运行。

如果你曾经使用过 [GCD](#)， 那你对以下代码应该不会陌生：

```
// 后台取得数据，主线程处理结果
DispatchQueue.global(qos: .userInitiated).async {
    let data = try? Data(contentsOf: url)
    DispatchQueue.main.async {
        self.data = data
    }
}
```

如果用 **RxSwift** 来实现，大致是这样的：

```
let rxData: Observable<Data> = ...

rxData
    .subscribeOn(ConcurrentDispatchQueueScheduler(qos: .userInitiated))
    .observeOn(MainScheduler.instance)
    .subscribe(onNext: { [weak self] data in
        self?.data = data
    })
    .disposed(by: disposeBag)
```

使用 [subscribeOn](#)

我们用 [subscribeOn](#) 来决定数据序列的构建函数在哪个 **Scheduler** 上运行。以上例子中，由于获取 `Data` 需要花很长的时间，所以用 [subscribeOn](#) 切换到 **后台 Scheduler** 来获取 `Data`。这样可以避免主线程被阻塞。

使用 [observeOn](#)

我们用 `observeOn` 来决定在哪个 **Scheduler** 监听这个数据序列。以上例子中，通过使用 `observeOn` 方法切换到主线程来监听并且处理结果。

一个比较典型的例子就是，在后台发起网络请求，然后解析数据，最后在主线程刷新页面。你就可以先用 `subscribeOn` 切到后台去发送请求并解析数据，最后用 `observeOn` 切换到主线程更新页面。

MainScheduler

MainScheduler 代表主线程。如果你需要执行一些和 UI 相关的任务，就需要切换到该 Scheduler 运行。

SerialDispatchQueueScheduler

SerialDispatchQueueScheduler 抽象了串行 `DispatchQueue`。如果你需要执行一些串行任务，可以切换到这个 Scheduler 运行。

ConcurrentDispatchQueueScheduler

ConcurrentDispatchQueueScheduler 抽象了并行 `DispatchQueue`。如果你需要执行一些并发任务，可以切换到这个 Scheduler 运行。

OperationQueueScheduler

OperationQueueScheduler 抽象了 `NSOperationQueue`。

它具备 `NSOperationQueue` 的一些特点，例如，你可以通过设置 `maxConcurrentOperationCount`，来控制同时执行并发任务的最大数量。

Error Handling - 错误处理

一旦序列里面出产出了一个 `error` 事件，整个序列将被终止。[RxSwift](#) 主要有两种错误处理机制：

- `retry` - 重试
- `catch` - 恢复

retry - 重试

[retry](#) 可以让序列在发生错误后重试：

```
// 请求 JSON 失败时，立即重试，  
// 重试 3 次后仍然失败，就将错误抛出  
  
let rxJson: Observable<JSON> = ...  
  
rxJson  
    .retry(3)  
    .subscribe(onNext: { json in  
        print("取得 JSON 成功: \(json)")  
    }, onError: { error in  
        print("取得 JSON 失败: \(error)")  
    })  
    .disposed(by: disposeBag)
```

以上的代码非常直接 `retry(3)` 就是当发生错误时，就进行重试操作，并且最多重试 3 次。

retryWhen

如果我们需要在发生错误时，经过一段延时后重试，那可以这样实现：

```
// 请求 JSON 失败时，等待 5 秒后重试，  
  
let retryDelay: Double = 5 // 重试延时 5 秒  
  
rxJson  
    .retryWhen { (rxError: Observable<Error>) -> Observable<Int> in  
        return Observable.timer(retryDelay, scheduler: MainScheduler.instance)  
    }  
    .subscribe(...)  
    .disposed(by: disposeBag)
```

这里我们需要用到 `retryWhen` 操作符，这个操作符主要描述应该在何时重试，并且通过闭包里面返回的 `Observable` 来控制重试的时机：

```
.retryWhen { (rxError: Observable<Error>) -> Observable<Int> in
    ...
}
```

闭包里面的参数是 `Observable<Error>` 也就是所产生错误的序列，然后返回值是一个 `Observable`。当这个返回的 `Observable` 发出一个元素时，就进行重试操作。当它发出一个 `error` 或者 `completed` 事件时，就不会重试，并且将这个事件传递给到后面的观察者。

如果需要加上一个最大重试次数的限制：

```
// 请求 JSON 失败时，等待 5 秒后重试，
// 重试 4 次后仍然失败，就将错误抛出

let maxRetryCount = 4          // 最多重试 4 次
let retryDelay: Double = 5    // 重试延时 5 秒

rxJson
    .retryWhen { (rxError: Observable<Error>) -> Observable<Int> in
        return rxError.flatMapWithIndex { (error, index) -> Observable<Int> in
            guard index < maxRetryCount else {
                return Observable.error(error)
            }
            return Observable<Int>.timer(retryDelay, scheduler: MainScheduler.instance)
        }
    }
    .subscribe(...)
    .disposed(by: disposeBag)
```

我们这里要实现的是，如果重试超过 4 次，就将错误抛出。如果错误在 4 次以内时，就等待 5 秒后重试：

```
...
rxError.flatMapWithIndex { (error, index) -> Observable<Int> in
    guard index < maxRetryCount else {
        return Observable.error(error)
    }
    return Observable<Int>.timer(retryDelay, scheduler: MainScheduler.instance)
}
...
```

我们用 `flatMapWithIndex` 这个操作符，因为它可以给我们提供错误的索引数 `index`。然后用这个索引数判断是否超过最大重试数，如果超过了，就将错误抛出。如果没有超过，就等待 5 秒后重试。

catchError - 恢复

`catchError` 可以在错误产生时，用一个备用元素或者一组备用元素将错误替换掉：

```
searchBar.rx.text.orEmpty
...
.flatMapLatest { query -> Observable<[Repository]> in
...
    return searchGitHub(query)
        .catchErrorJustReturn([])
}
...
.bind(to: ...)
.disposed(by: disposeBag)
```

我们开头的 [Github 搜索](#)就用到了`catchErrorJustReturn`。当错误产生时，就返回一个空数组，于是就会显示一个空列表页。

你也可以使用 `catchError`，当错误产生时，将错误事件替换成一个备选序列：

```
// 先从网络获取数据，如果获取失败了，就从本地缓存获取数据

let rxData: Observable<Data> = ...           // 网络请求的数据
let cahcedData: Observable<Data> = ... // 之前本地缓存的数据

rxData
    .catchError { _ in cahcedData }
    .subscribe(onNext: { date in
        print("获取数据成功: \(date.count)")
    })
    .disposed(by: disposeBag)
```

Result

如果我们只是想给用户错误提示，那要如何操作呢？

以下提供一个最为直接的方案，不过这个方案存在一些问题：

```
// 当用户点击更新按钮时,
// 就立即取出修改后的用户信息。
// 然后发起网络请求，进行更新操作,
// 一旦操作失败就提示用户失败原因

updateUserInfoButton.rx.tap
    .withLatestFrom(rxUserInfo)
    .flatMapLatest { userInfo -> Observable<Void> in
        return update(userInfo)
    }
    .observeOn(MainScheduler.instance)
```

```

.subscribe(onNext: {
    print("用户信息更新成功")
}, onError: { error in
    print("用户信息更新失败: \(error.localizedDescription)")
})
.disposed(by: disposeBag)

```

这样实现是非常直接的。但是一旦网络请求操作失败了，序列就会终止。整个订阅将被取消。如果用户再次点击更新按钮，就无法再次发起网络请求进行更新操作了。

为了解决这个问题，我们需要选择合适的方案来进行错误处理。例如，使用系统自带的枚举 **Result**:

```

public enum Result<Success, Failure> where Failure : Error {
    case success(Success)
    case failure(Failure)
}

```

然后之前的代码需要修改成：

```

updateUserInfoButton.rx.tap
    .withLatestFrom(rxUserInfo)
    .flatMapLatest { userInfo -> Observable<Result<Void, Error>> in
        return update(userInfo)
            .map(Result.success) // 转换成 Result
            .catchError { error in Observable.just(Result.failure(error)) }
    }
    .observeOn(MainScheduler.instance)
    .subscribe(onNext: { result in
        switch result { // 处理 Result
        case .success:
            print("用户信息更新成功")
        case .failure(let error):
            print("用户信息更新失败: \(error.localizedDescription)")
        }
    })
    .disposed(by: disposeBag)

```

这样我们的错误事件被包装成了 `Result.failure(Error)` 元素，就不会终止整个序列。即便网络请求失败了，整个订阅依然存在。如果用户再次点击更新按钮，也是能够发起网络请求进行更新操作的。

另外你也可以使用 `materialize` 操作符来进行错误处理。这里就不详细介绍了，如你想了解如何使用 `materialize` 可以参考这篇文章 [How to handle errors in RxSwift!](#)

如何选择操作符?



下面这个决策树可以帮助你找到需要的操作符。

决策树

我想要创建一个 `Observable`

- 产生特定的一个元素: `just`
 - 经过一段延时: `timer`
- 从一个序列拉取元素: `from`
- 重复的产生某一个元素: `repeatElement`
- 存在自定义逻辑: `create`
- 每次订阅时产生: `deferred`
- 每隔一段时间, 发出一个元素: `interval`
 - 在一段延时后: `timer`
- 一个空序列, 只有一个完成事件: `empty`
- 一个任何事件都没有产生的序列: `never`

我想要创建一个 `Observable` 通过组合其他的 `observables`

- 任意一个 `Observable` 产生了元素, 就发出这个元素: `merge`
- 让这些 `Observables` 一个接一个的发出元素, 当上一个 `Observable` 元素发送完毕后, 下一个 `Observable` 才能开始发出元素: `concat`
- 组合多个 `Observables` 的元素
 - 当每一个 `Observable` 都发出一个新的元素: `zip`
 - 当任意一个 `Observable` 发出一个新的元素: `combineLatest`

我想要转换 `Observable` 的元素后, 再将它们发出来

- 对每个元素直接转换: `map`
- 转换到另一个 `Observable` : `flatMap`
 - 只接收最新的元素转换的 `Observable` 所产生的元素: `flatMapLatest`
 - 每一个元素转换的 `Observable` 按顺序产生元素: `concatMap`
- 基于所有遍历过的元素: `scan`

我想要将产生的每一个元素, 拖延一段时间后再发出: `delay`

我想要将产生的事件封装成元素发送出来

- 将他们封装成 `Event<Element>` : `materialize`
 - 然后解封出来: `dematerialize`

我想要忽略掉所有的 `next` 事件, 只接收 `completed` 和 `error` 事件: `ignoreElements`

我想创建一个新的 `Observable` 在原有的序列前面加入一些元素: `startWith`

我想从 `Observable` 中收集元素, 缓存这些元素之后在发出: `buffer`

我想将 `Observable` 拆分成多个 `Observables` : `window`

- 基于元素的共同特征: `groupBy`

我想只接收 `Observable` 中特定的元素

- 发出唯一的元素: `single`

我想重新从 `Observable` 中发出某些元素

- 通过判定条件过滤出一些元素: `filter`
- 仅仅发出头几个元素: `take`
- 仅仅发出尾部的几个元素: `takeLast`
- 仅仅发出第 n 个元素: `elementAt`
- 跳过头几个元素
 - 跳过头 n 个元素: `skip`
 - 跳过头几个满足判定的元素: `skipWhile`, `skipWhileWithIndex`
 - 跳过某段时间内产生的头几个元素: `skip`
 - 跳过头几个元素直到另一个 `Observable` 发出一个元素: `skipUntil`
- 只取头几个元素
 - 只取头几个满足判定的元素: `takeWhile`, `takeWhileWithIndex`
 - 只取某段时间内产生的头几个元素: `take`
 - 只取头几个元素直到另一个 `Observable` 发出一个元素: `takeUntil`
- 周期性的对 `Observable` 抽样: `sample`
- 发出那些元素, 这些元素产生后的特定的时间内, 没有新的元素产生: `debounce`
- 直到元素的值发生变化, 才发出新的元素: `distinctUntilChanged`
 - 并提供元素是否相等的判定函数: `distinctUntilChanged`
- 在开始发出元素时, 延时后进行订阅: `delaySubscription`

我想要从一些 `Observables` 中, 只取第一个产生元素的 `Observable` : `amb`

我想评估 `Observable` 的全部元素

- 并且对每个元素应用聚合方法, 待所有元素都应用聚合方法后, 发出结果: `reduce`
- 并且对每个元素应用聚合方法, 每次应用聚合方法后, 发出结果: `scan`

我想把 `Observable` 转换为其他的数据结构: `as...`

我想在某个 `Scheduler` 应用操作符: `subscribeOn`

- 在某个 `Scheduler` 监听: `observeOn`

我想要 `Observable` 发生某个事件时, 采取某个行动: `do`

我想要 `Observable` 发出一个 `error` 事件: `error`

- 如果规定时间内没有产生元素: `timeout`

我想要 `Observable` 发生错误时, 优雅的恢复

- 如果规定时间内没有产生元素, 就切换到备选 `Observable : timeout`
- 如果产生错误, 将错误替换成某个元素: `catchErrorJustReturn`
- 如果产生错误, 就切换到备选 `Observable : catchError`
- 如果产生错误, 就重试: `retry`

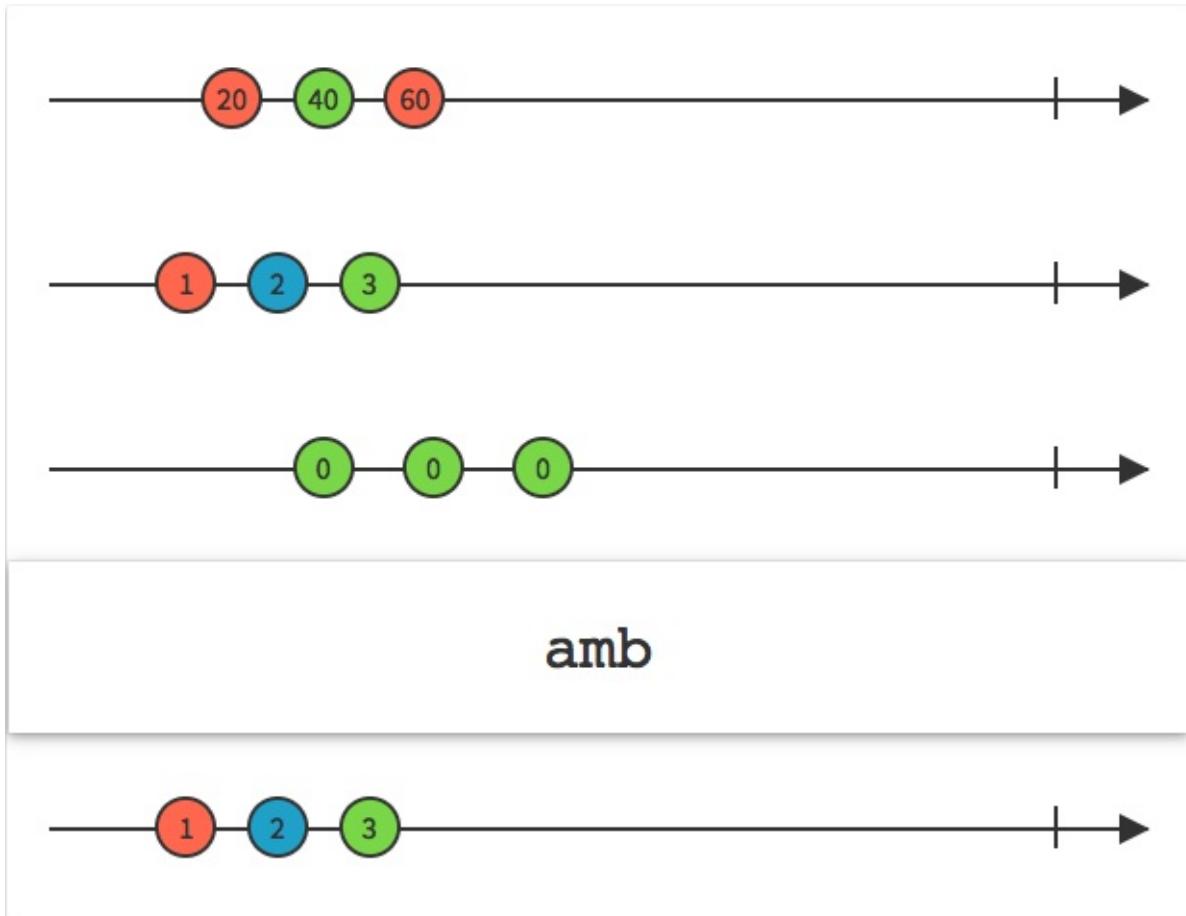
我创建一个 `Disposable` 资源, 使它与 `Observable` 具有相同的寿命: `using`

我创建一个 `Observable`, 直到我通知它可以产生元素后, 才能产生元素: `publish`

- 并且, 就算是在产生元素后订阅, 也要发出全部元素: `replay`
- 并且, 一旦所有观察者取消观察, 他就被释放掉: `refCount`
- 通知它可以产生元素了: `connect`

amb

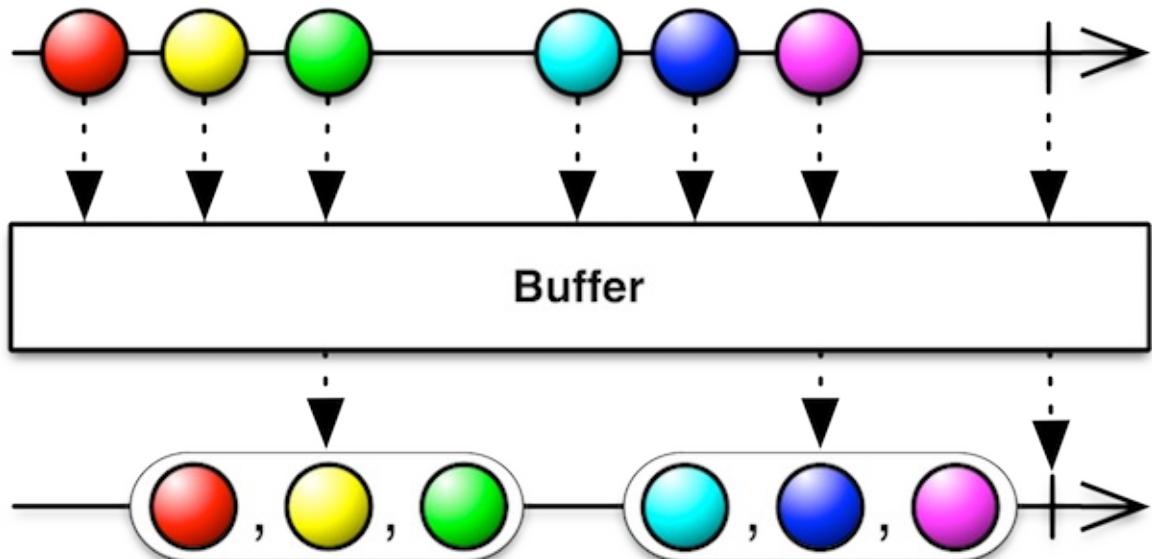
在多个源 `observables` 中， 取第一个发出元素或产生事件的 `observable`， 然后只发出它的元素



当你传入多个 `Observables` 到 `amb` 操作符时，它将取其中一个 `observable`：第一个产生事件的那个 `Observable`，可以是一个 `next`，`error` 或者 `completed` 事件。 `amb` 将忽略掉其他的 `Observables`。

buffer

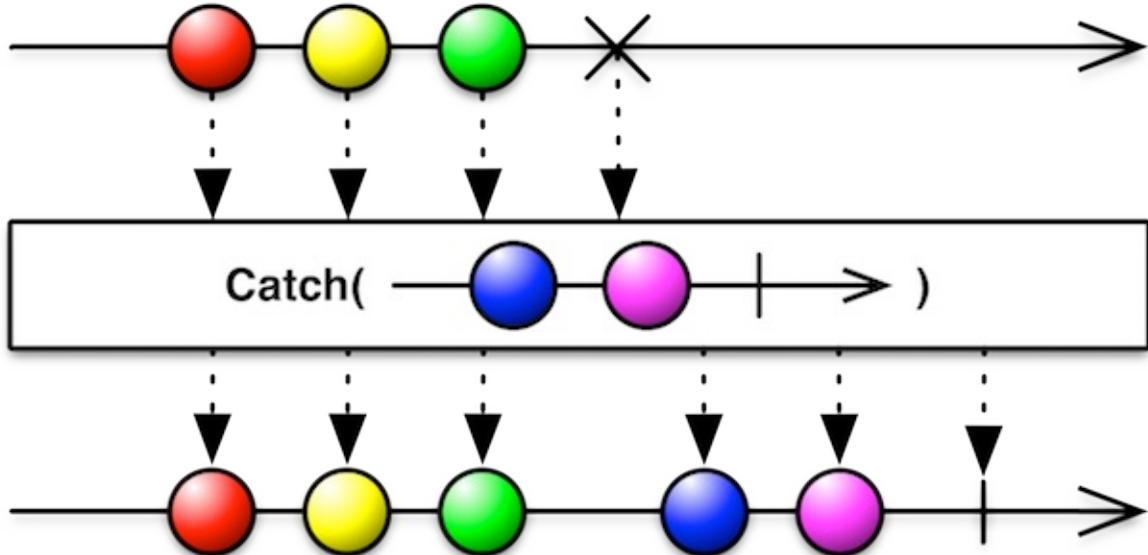
缓存元素，然后将缓存的元素集合，周期性的发出来



`buffer` 操作符将缓存 `Observable` 中发出的新元素，当元素达到某个数量，或者经过了特定的时间，它就会将这个元素集合发送出来。

catchError

从一个错误事件中恢复，将错误事件替换成一个备选序列



catchError 操作符将会拦截一个 `error` 事件，将它替换成其他的元素或者一组元素，然后传递给观察者。这样可以使得 `Observable` 正常结束，或者根本都不需要结束。

这里存在其他版本的 `catchError` 操作符。

演示

```
let disposeBag = DisposeBag()

let sequenceThatFails = PublishSubject<String>()
let recoverySequence = PublishSubject<String>()

sequenceThatFails
    .catchError {
        print("Error:", $0)
        return recoverySequence
    }
    .subscribe { print($0) }
    .disposed(by: disposeBag)

sequenceThatFails.onNext(" ")
sequenceThatFails.onNext(" ")
sequenceThatFails.onNext(" ")
sequenceThatFails.onNext(" ")
sequenceThatFails.onError(TestError.test)
```

```
recoverySequence.onNext(" ")
```

输出结果：

```
next( )  
next( )  
next( )  
next( )  
Error: test  
next( )
```

catchErrorJustReturn

catchErrorJustReturn 操作符会将 `error` 事件替换成其他的一个元素，然后结束该序列。

演示

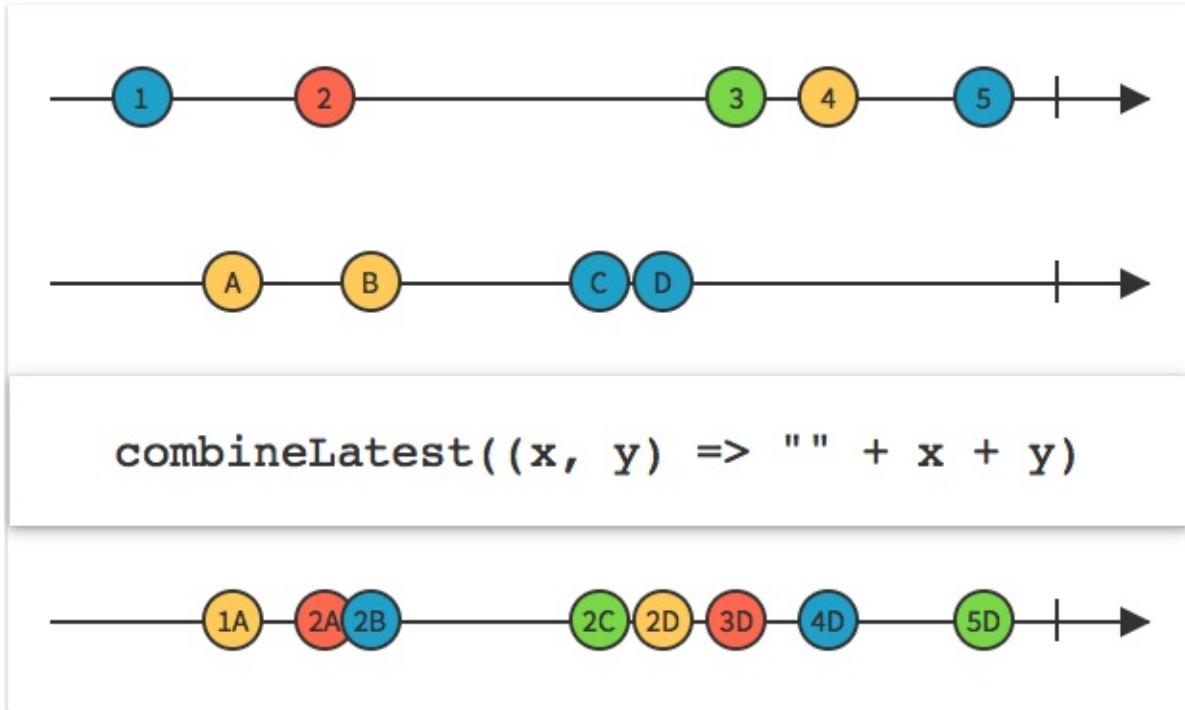
```
let disposeBag = DisposeBag()  
let sequenceThatFails = PublishSubject<String>()  
  
sequenceThatFails  
    .catchErrorJustReturn(" ")  
    .subscribe { print($0) }  
    .disposed(by: disposeBag)  
  
sequenceThatFails.onNext(" ")  
sequenceThatFails.onNext(" ")  
sequenceThatFails.onNext(" ")  
sequenceThatFails.onNext(" ")  
sequenceThatFails.onError(TestError.test)
```

输出结果：

```
next( )  
next( )  
next( )  
next( )  
next( )  
completed
```


combineLatest

当多个 `Observables` 中任何一个发出一个元素，就发出一个元素。这个元素是由这些 `Observables` 中最新的元素，通过一个函数组合起来的



`combineLatest` 操作符将多个 `Observables` 中最新的元素通过一个函数组合起来，然后将这个组合的结果发出来。这些源 `Observables` 中任何一个发出一个元素，他都会发出一个元素（前提是，这些 `Observables` 曾经发出过元素）。

演示

tips: 可与 `zip` 比较学习

```

let disposeBag = DisposeBag()

let first = PublishSubject<String>()
let second = PublishSubject<String>()

Observable.combineLatest(first, second) { $0 + $1 }
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)

first.onNext("1")
second.onNext("A")
first.onNext("2")
second.onNext("B")
  
```

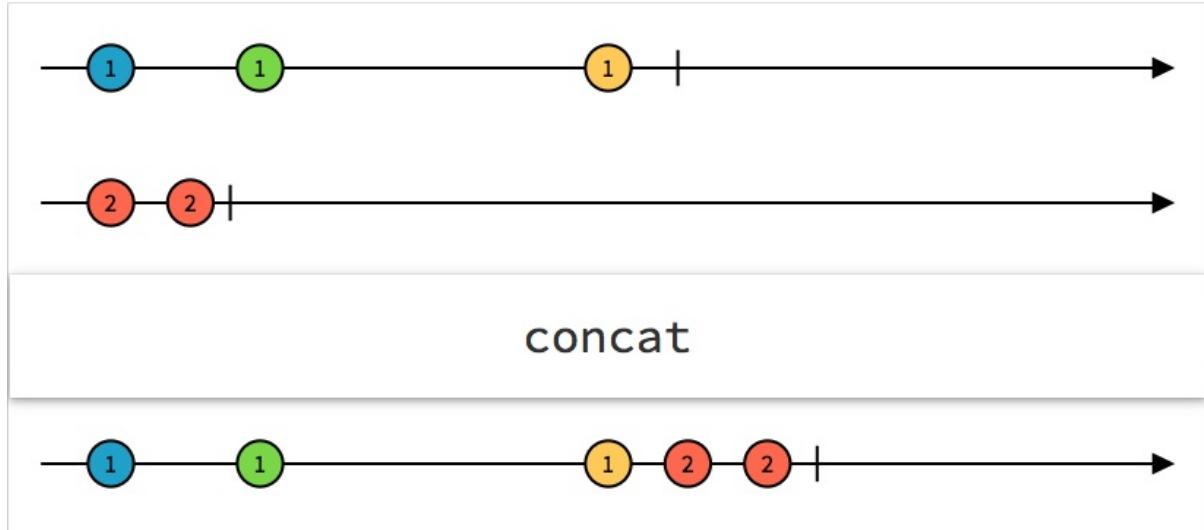
```
second.onNext("C")
second.onNext("D")
first.onNext("3")
first.onNext("4")
```

输出结果：

```
1A
2A
2B
2C
2D
3D
4D
```

concat

让两个或多个 `Observables` 按顺序串连起来



`concat` 操作符将多个 `Observables` 按顺序串联起来，当前一个 `Observable` 元素发送完毕后，后一个 `Observable` 才可以开始发出元素。

`concat` 将等待前一个 `Observable` 产生完成事件后，才对后一个 `Observable` 进行订阅。如果后一个是“热” `Observable`，在它前一个 `Observable` 产生完成事件前，所产生的元素将不会被发送出来。

`startWith` 和它十分相似。但是`startWith`不是在后面添加元素，而是在前面插入元素。

`merge` 和它也是十分相似。`merge`并不是将多个 `Observables` 按顺序串联起来，而是将他们合并到一起，不需要 `Observables` 按先后顺序发出元素。

演示

```
let disposeBag = DisposeBag()

let subject1 = BehaviorSubject(value: " ")
let subject2 = BehaviorSubject(value: " ")

let variable = Variable(subject1)

variable.asObservable()
    .concat()
    .subscribe { print($0) }
    .disposed(by: disposeBag)

subject1.onNext(" ")
```

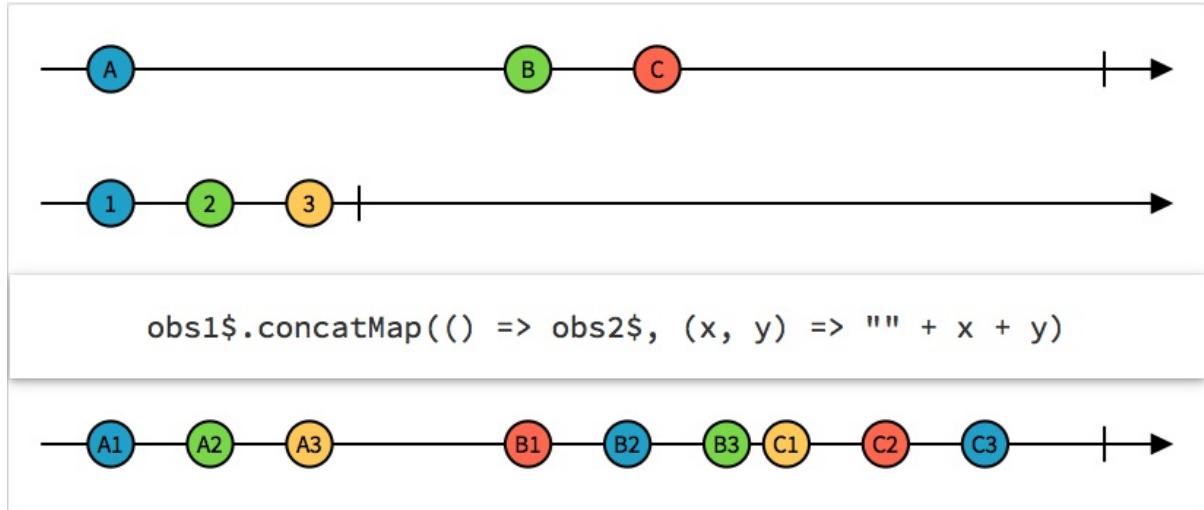
```
subject1.onNext(" ")  
  
variable.value = subject2  
  
subject2.onNext("I would be ignored")  
subject2.onNext(" ")  
  
subject1.onCompleted()  
  
subject2.onNext(" ")
```

输出结果：

```
next()  
next()  
next()  
next()  
next()
```

concatMap

将 `Observable` 的元素转换成其他的 `Observable`，然后将这些 `Observables` 串连起来



`concatMap` 操作符将源 `Observable` 的每一个元素应用一个转换方法，将他们转换成 `Observables`。然后让这些 `Observables` 按顺序的发出元素，当前一个 `Observable` 元素发送完毕后，后一个 `Observable` 才可以开始发出元素。等待前一个 `Observable` 产生完成事件后，才对后一个 `Observable` 进行订阅。

演示

```

let disposeBag = DisposeBag()

let subject1 = BehaviorSubject(value: "")
let subject2 = BehaviorSubject(value: "")

let variable = Variable(subject1)

variable.asObservable()
    .concatMap { $0 }
    .subscribe { print($0) }
    .disposed(by: disposeBag)

subject1.onNext(" ")
subject1.onNext(" ")

variable.value = subject2

subject2.onNext("I would be ignored")
subject2.onNext(" ")

```

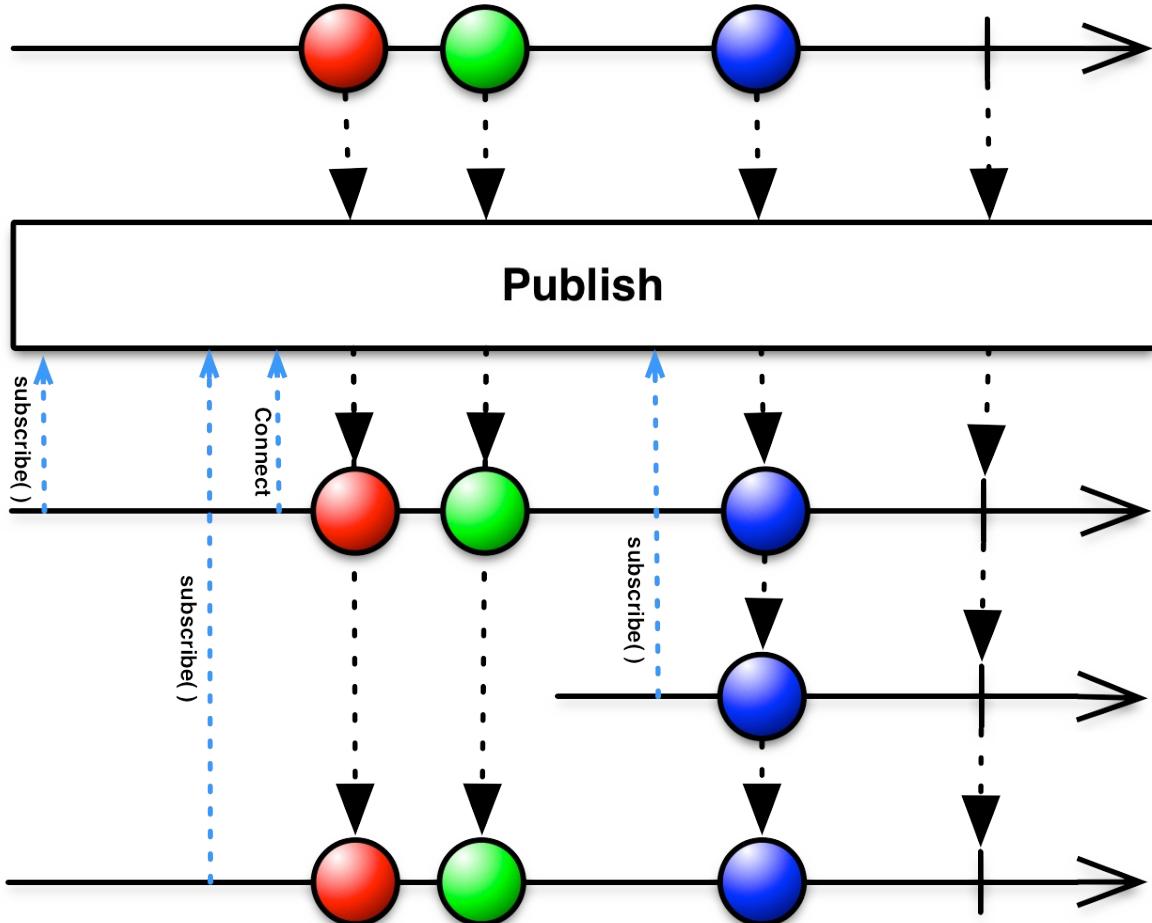
```
subject1.onCompleted()  
subject2.onNext(" ")
```

输出结果：

```
next()  
next()  
next()  
next()  
next()
```

connect

通知 `ConnectableObservable` 可以开始发出元素了



`ConnectableObservable` 和普通的 `Observable` 十分相似，不过在被订阅后不会发出元素，直到 `connect` 操作符被应用为止。这样一来你可以等所有观察者全部订阅完成后，才发出元素。

演示

```
let intSequence = Observable<Int>.interval(1, scheduler: MainScheduler.instance)
    .publish()

_ = intSequence
    .subscribe(onNext: { print("Subscription 1:, Event: \"\($0)\"") })

DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
    _ = intSequence.connect()
}

DispatchQueue.main.asyncAfter(deadline: .now() + 4) {
```

```
_ = intSequence
    .subscribe(onNext: { print("Subscription 2:, Event: \"\($0)\"") })
}

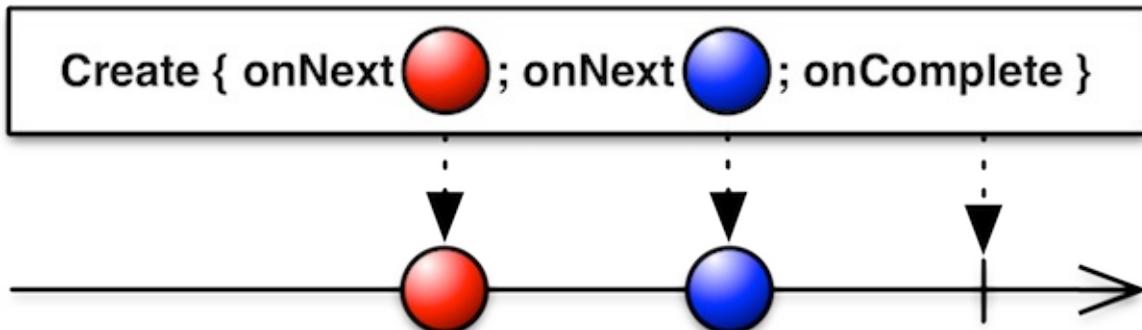
DispatchQueue.main.asyncAfter(deadline: .now() + 6) {
    _ = intSequence
    .subscribe(onNext: { print("Subscription 3:, Event: \"\($0)\"") })
}
```

输出结果：

```
Subscription 1:, Event: 0
Subscription 1:, Event: 1
Subscription 2:, Event: 1
Subscription 1:, Event: 2
Subscription 2:, Event: 2
Subscription 1:, Event: 3
Subscription 2:, Event: 3
Subscription 3:, Event: 3
Subscription 1:, Event: 4
Subscription 2:, Event: 4
Subscription 3:, Event: 4
Subscription 1:, Event: 5
Subscription 2:, Event: 5
Subscription 3:, Event: 5
Subscription 1:, Event: 6
Subscription 2:, Event: 6
Subscription 3:, Event: 6
...
...
```

create

通过一个构建函数完整的创建一个 `Observable`



`create` 操作符将创建一个 `Observable`，你需要提供一个构建函数，在构建函数里面描述事件 (`next` , `error` , `completed`) 的产生过程。

通常情况下一个有限的序列，只会调用一次观察者的 `onCompleted` 或者 `onError` 方法。并且在调用它们后，不会再去调用观察者的其他方法。

演示

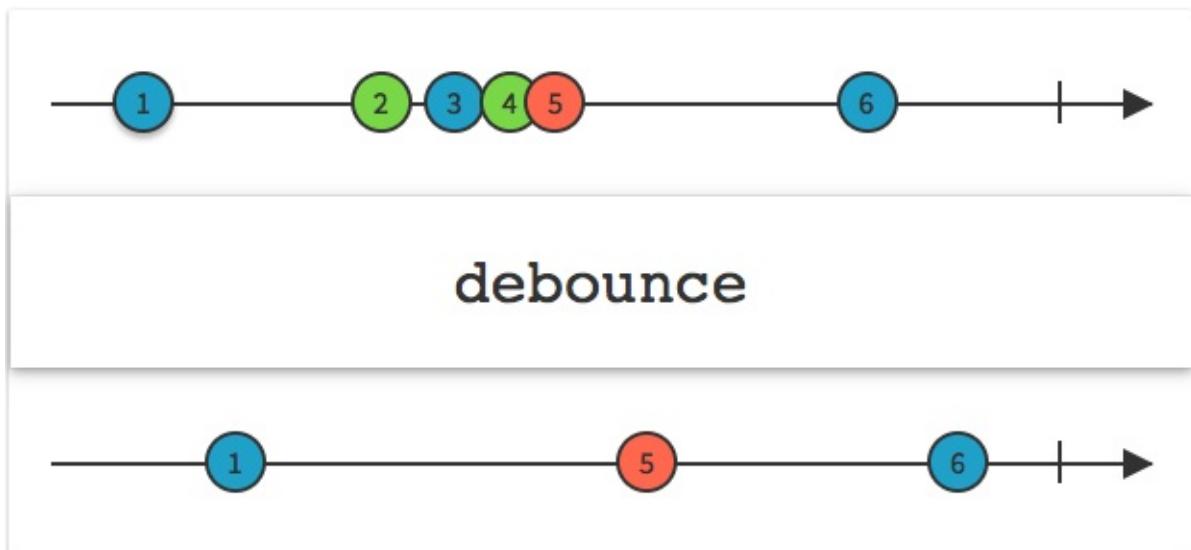
创建一个 `[0, 1, ... 8, 9]` 的序列：

```

let id = Observable<Int>.create { observer in
    observer.onNext(0)
    observer.onNext(1)
    observer.onNext(2)
    observer.onNext(3)
    observer.onNext(4)
    observer.onNext(5)
    observer.onNext(6)
    observer.onNext(7)
    observer.onNext(8)
    observer.onNext(9)
    observer.onCompleted()
    return Disposables.create()
}
  
```

debounce

过滤掉高频产生的元素



debounce 操作符将发出这种元素，在 Observable 产生这种元素后，一段时间内没有新元素产生。

debug

打印所有的订阅，事件以及销毁信息

演示

```
let disposeBag = DisposeBag()

let sequence = Observable<String>.create { observer in
    observer.onNext(" ")
    observer.onNext(" ")
    observer.onCompleted()
    return Disposables.create()
}

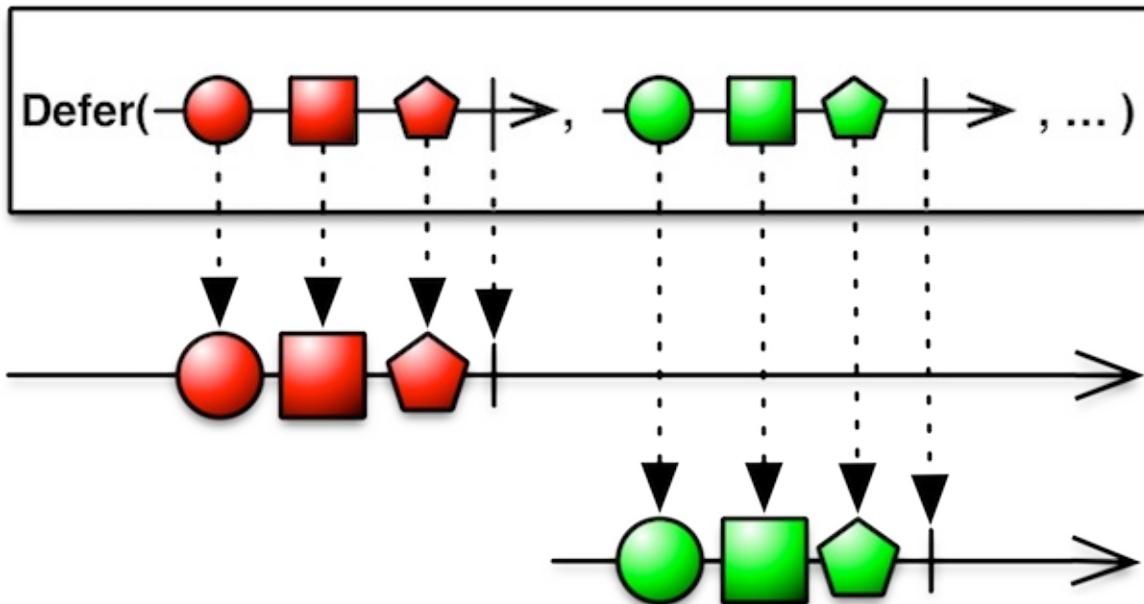
sequence
    .debug("Fruit")
    .subscribe()
    .disposed(by: disposeBag)
```

输出结果：

```
2017-11-06 20:49:43.187: Fruit -> subscribed
2017-11-06 20:49:43.188: Fruit -> Event next( )
2017-11-06 20:49:43.188: Fruit -> Event next( )
2017-11-06 20:49:43.188: Fruit -> Event completed
2017-11-06 20:49:43.189: Fruit -> isDisposed
```

deferred

直到订阅发生，才创建 `Observable`，并且为每位订阅者创建全新的 `Observable`

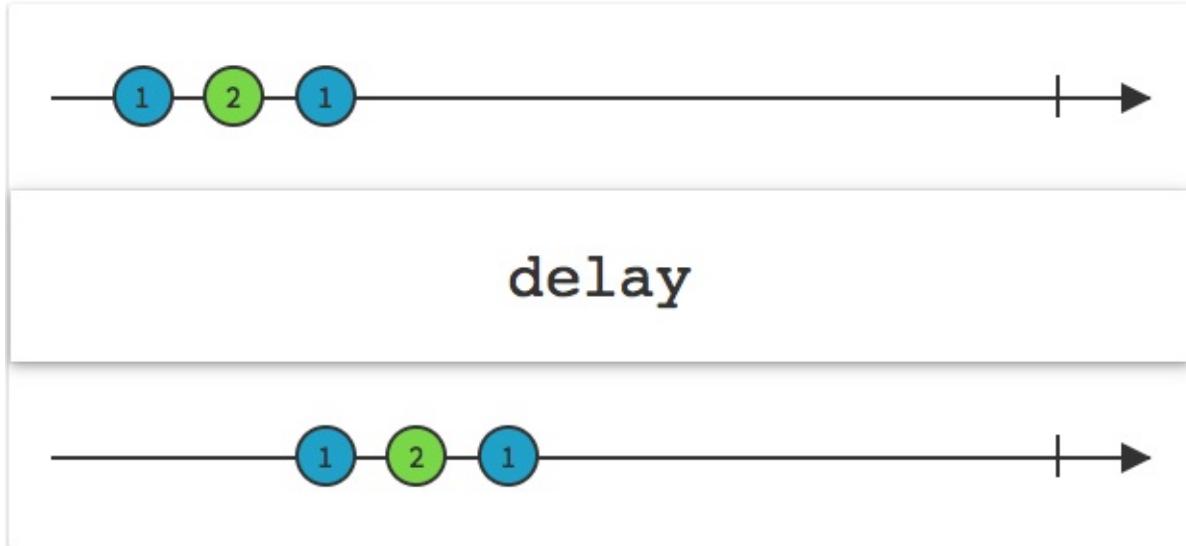


`deferred` 操作符将等待观察者订阅它，才创建一个 `Observable`，它会通过一个构建函数为每一位订阅者创建新的 `Observable`。看上去每位订阅者都是对同一个 `Observable` 产生订阅，实际上它们都获得了独立的序列。

在一些情况下，直到订阅时才创建 `Observable` 是可以保证拿到的数据都是最新的。

delay

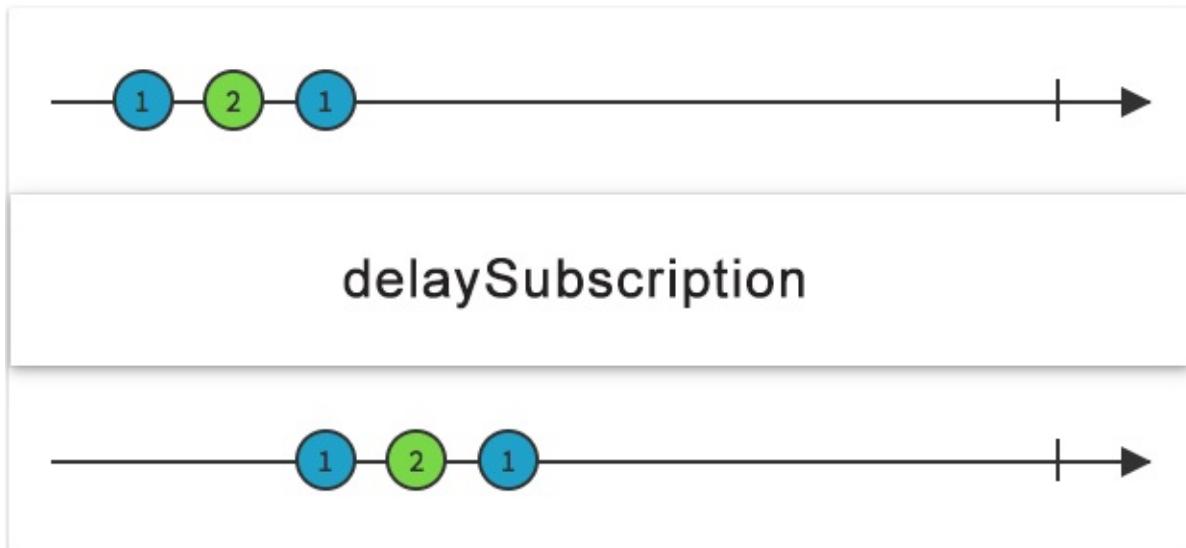
将 `Observable` 的每一个元素拖延一段时间后发出



`delay` 操作符将修改一个 `Observable`，它会将 `Observable` 的所有元素都拖延一段设定好的时间， 然后才将它们发送出来。

delaySubscription

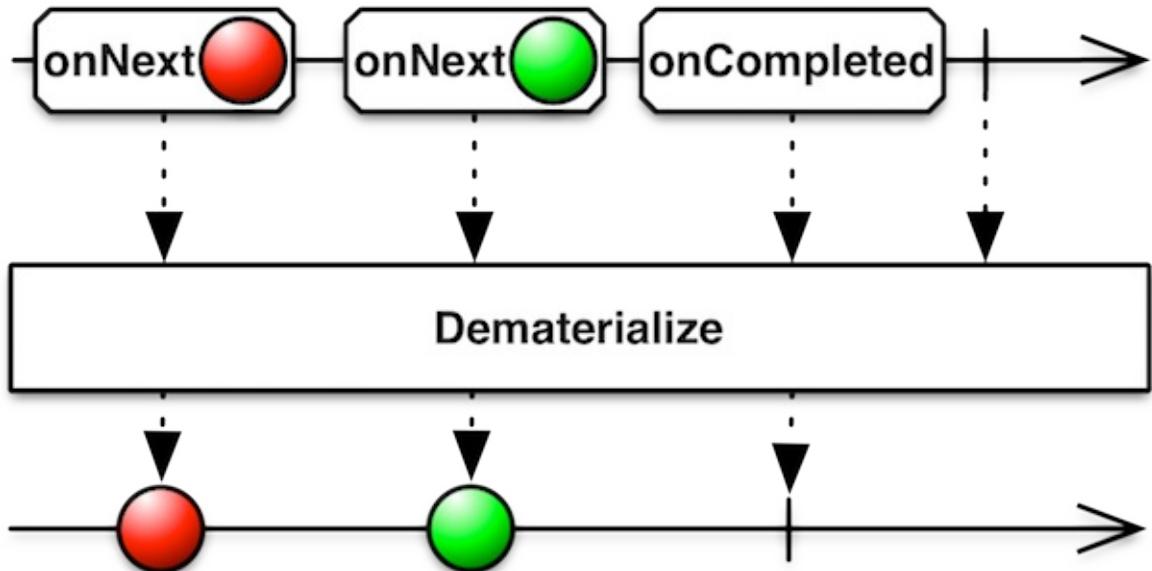
进行延时订阅



delaySubscription 操作符将在经过所设定的时间后，才对 `Observable` 进行订阅操作。

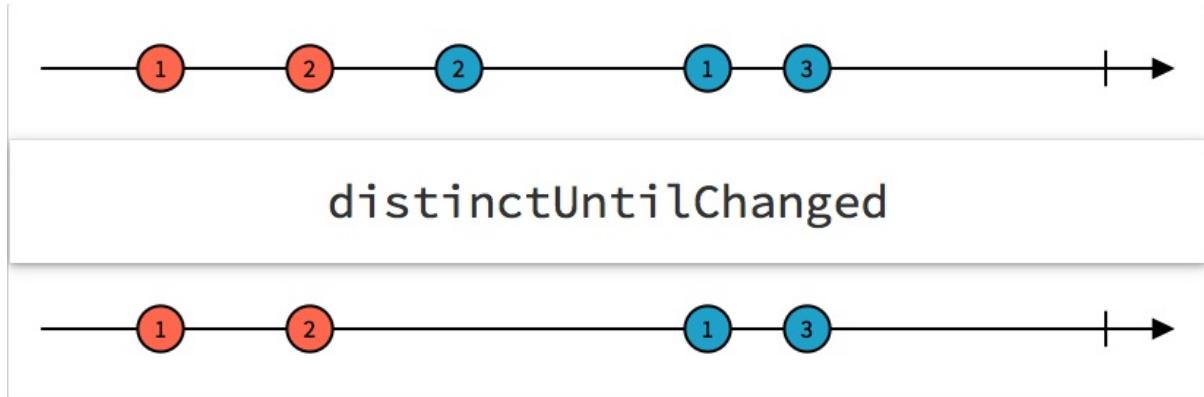
dematerialize

dematerialize 操作符将 materialize 转换后的元素还原



distinctUntilChanged

阻止 `observable` 发出相同的元素



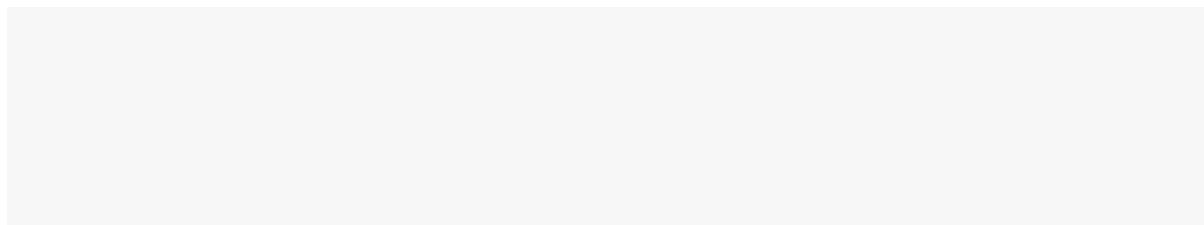
`distinctUntilChanged` 操作符将阻止 `observable` 发出相同的元素。如果后一个元素和前一个元素是相同的，那么这个元素将不会被发出来。如果后一个元素和前一个元素不相同，那么这个元素才会被发出来。

演示

```
let disposeBag = DisposeBag()

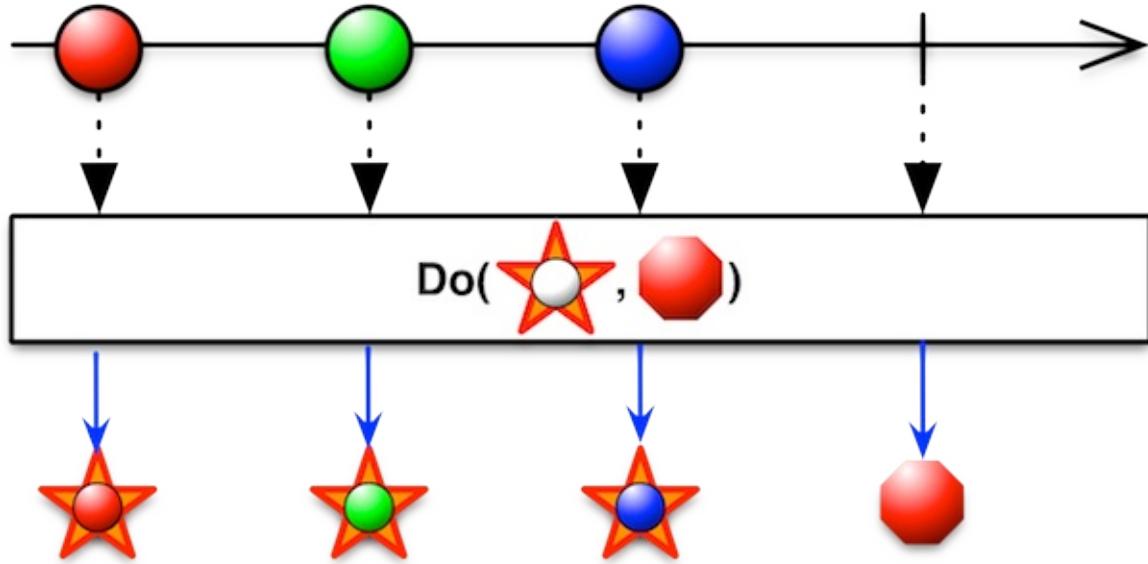
Observable.of(" ", " ", " ", " ", " ")
    .distinctUntilChanged()
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

输出结果：



do

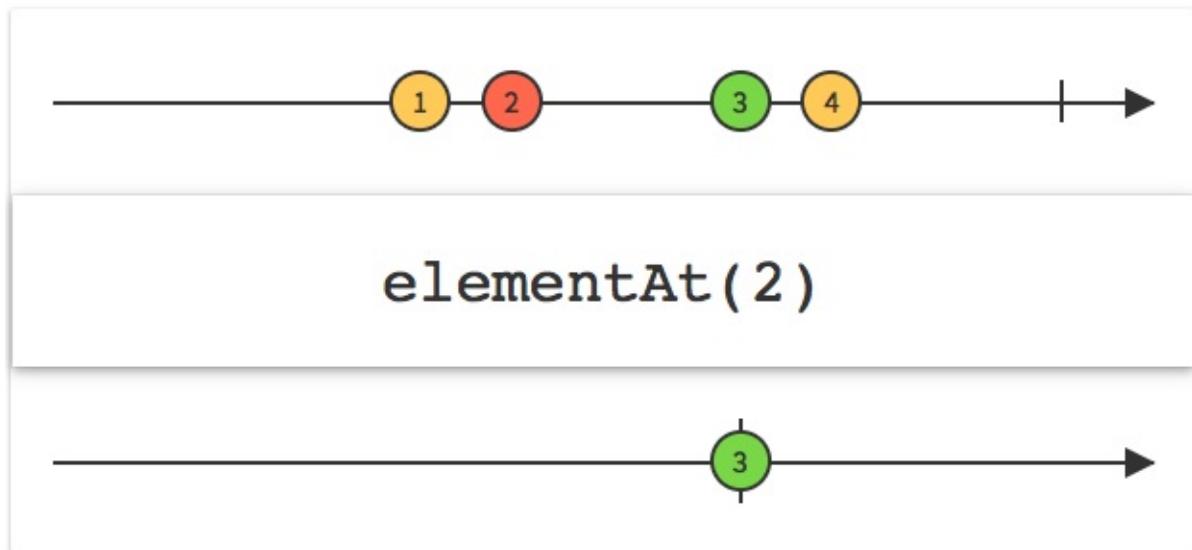
当 `Observable` 产生某些事件时，执行某个操作



当 `Observable` 的某些事件产生时，你可以使用 `do` 操作符来注册一些回调操作。这些回调会被单独调用，它们会和 `Observable` 原本的回调分离。

elementAt

只发出 `Observable` 中的第 n 个元素



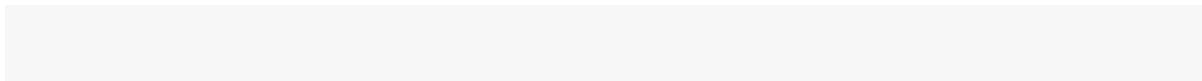
`elementAt` 操作符将拉取 `Observable` 序列中指定索引数的元素，然后将它作为唯一的元素发出。

演示

```
let disposeBag = DisposeBag()

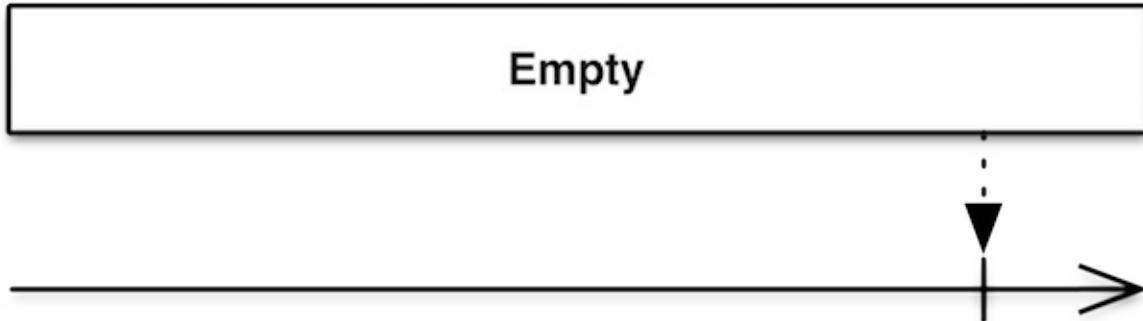
Observable.of(" ", " ", " ", " ")
    .elementAt(3)
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

输出结果：



empty

创建一个空 Observable



empty 操作符将创建一个 Observable，这个 observable 只有一个完成事件。

演示

创建一个空 Observable :

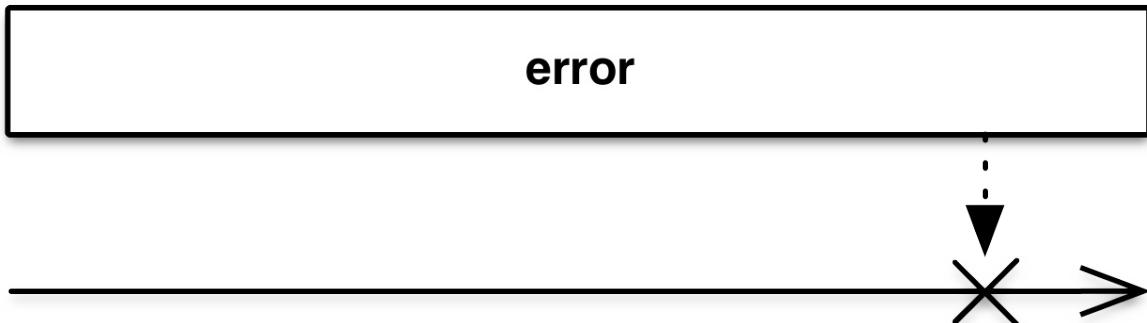
```
let id = Observable<Int>.empty()
```

它相当于：

```
let id = Observable<Int>.create { observer in
    observer.onCompleted()
    return Disposables.create()
}
```

error

创建一个只有 `error` 事件的 `Observable`



`error` 操作符将创建一个 `Observable`，这个 `Observable` 只会产生一个 `error` 事件。

演示

创建一个只有 `error` 事件的 `Observable`：

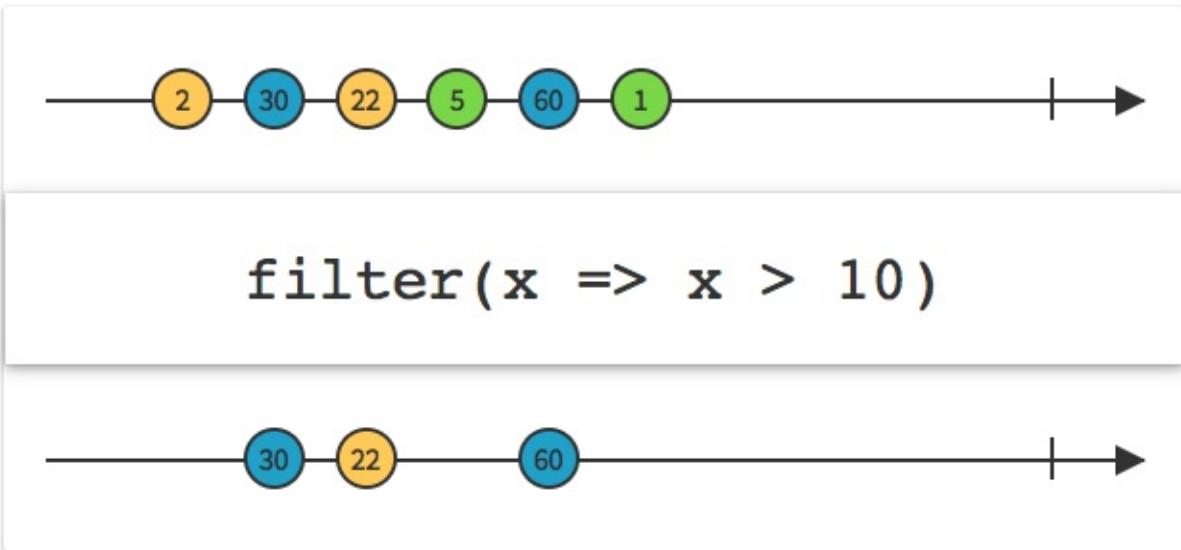
```
let error: Error = ...
let id = Observable<Int>.error(error)
```

它相当于：

```
let error: Error = ...
let id = Observable<Int>.create { observer in
  observer.onError(error)
  return Disposables.create()
}
```

filter

仅仅发出 `observable` 中通过判定的元素



`filter` 操作符将通过你提供的判定方法过滤一个 `observable`。

演示

```
let disposeBag = DisposeBag()

Observable.of(2, 30, 22, 5, 60, 1)
    .filter { $0 > 10 }
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

输出结果：

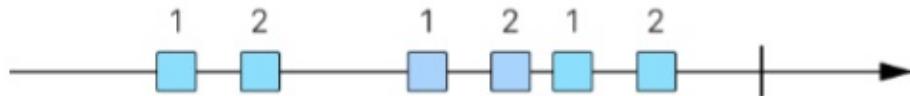
```
30
22
60
```

flatMap

将 `Observable` 的元素转换成其他的 `Observable`，然后将这些 `Observables` 合并



`a.flatMap(b)`



`flatMap` 操作符将源 `Observable` 的每一个元素应用一个转换方法，将他们转换成 `Observables`。然后将这些 `Observables` 的元素合并之后再发送出来。

这个操作符是非常有用的，例如，当 `Observable` 的元素本身拥有其他的 `Observable` 时，你可以将所有子 `Observables` 的元素发送出来。

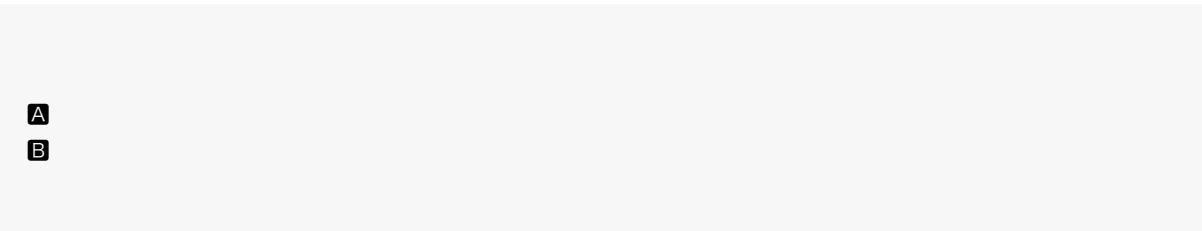
演示

```
let disposeBag = DisposeBag()
let first = BehaviorSubject(value: " ")
let second = BehaviorSubject(value: "A")
let variable = Variable(first)

variable.asObservable()
    .flatMap { $0 }
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)

first.onNext(" ")
variable.value = second
second.onNext("B")
first.onNext(" ")
```

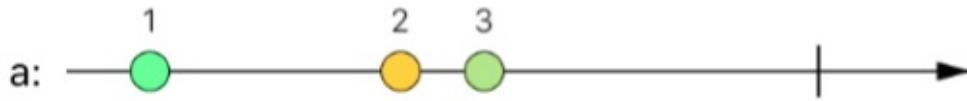
输出结果：



A
B

flatMapLatest

将 `Observable` 的元素转换成其他的 `Observable`，然后取这些 `observables` 中最新的一个



a.flatMapLatest(b)



flatMapLatest 操作符将源 `Observable` 的每一个元素应用一个转换方法，将他们转换成 `Observables`。一旦转换出一个新的 `Observable`，就只发出它的元素，旧的 `Observables` 的元素将被忽略掉。

演示

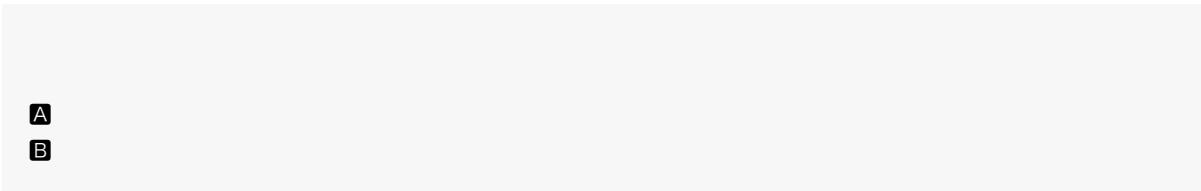
tips: 与 `flatMap` 比较更容易理解

```
let disposeBag = DisposeBag()
let first = BehaviorSubject(value: " ")
let second = BehaviorSubject(value: "A")
let variable = Variable(first)

variable.asObservable()
    .flatMapLatest { $0 }
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)

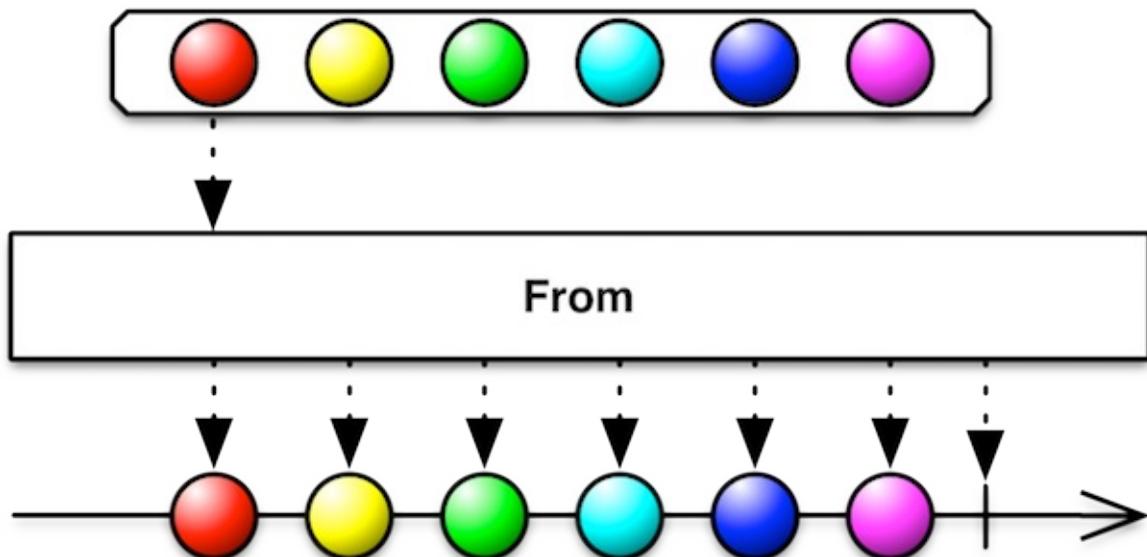
first.onNext(" ")
variable.value = second
second.onNext("B")
first.onNext(" ")
```

输出结果：



from

将其他类型或者数据结构转换为 `Observable`



当你在使用 `Observable` 时，如果能够直接将其他类型转换为 `Observable`，这将是非常省事的。`from` 操作符就提供了这种功能。

演示

将一个数组转换为 `Observable`：

```
let numbers = Observable.from([0, 1, 2])
```

它相当于：

```
let numbers = Observable<Int>.create { observer in
    observer.onNext(0)
    observer.onNext(1)
    observer.onNext(2)
    observer.onCompleted()
    return Disposables.create()
}
```

将一个可选值转换为 `Observable`：

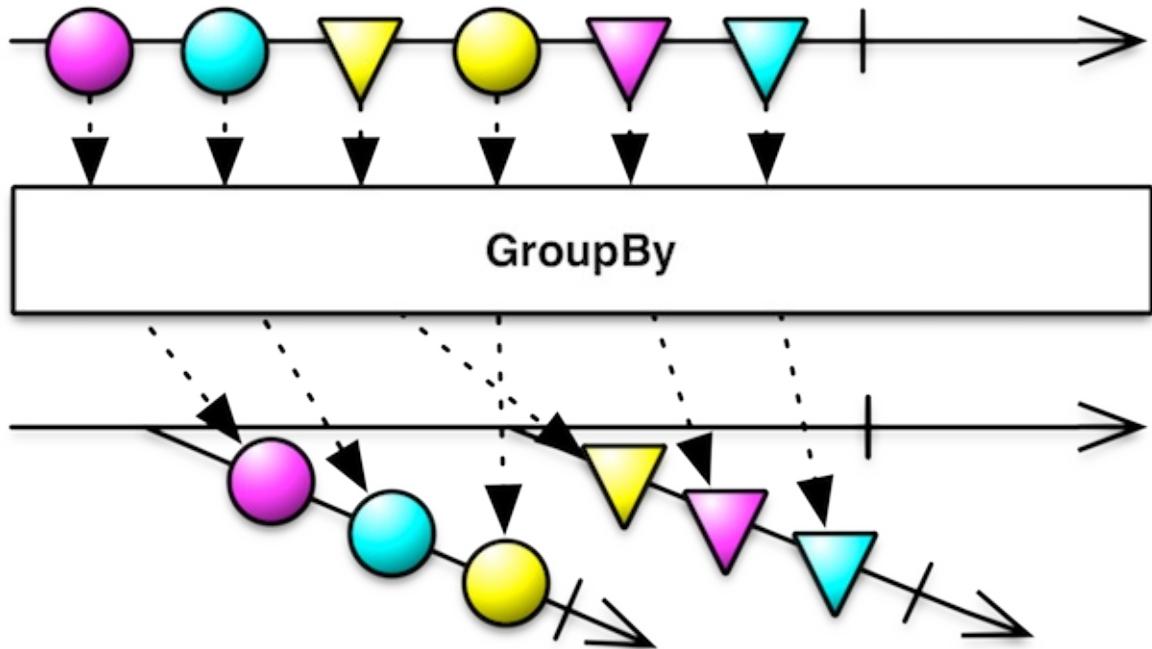
```
let optional: Int? = 1
let value = Observable.from(optional: optional)
```

它相当于：

```
let optional: Int? = 1
let value = Observable<Int>.create { observer in
    if let element = optional {
        observer.onNext(element)
    }
    observer.onCompleted()
    return Disposables.create()
}
```

groupBy

将源 Observable 分解为多个子 Observable , 并且每个子 Observable 将源 Observable 中“相似”的元素发送出来

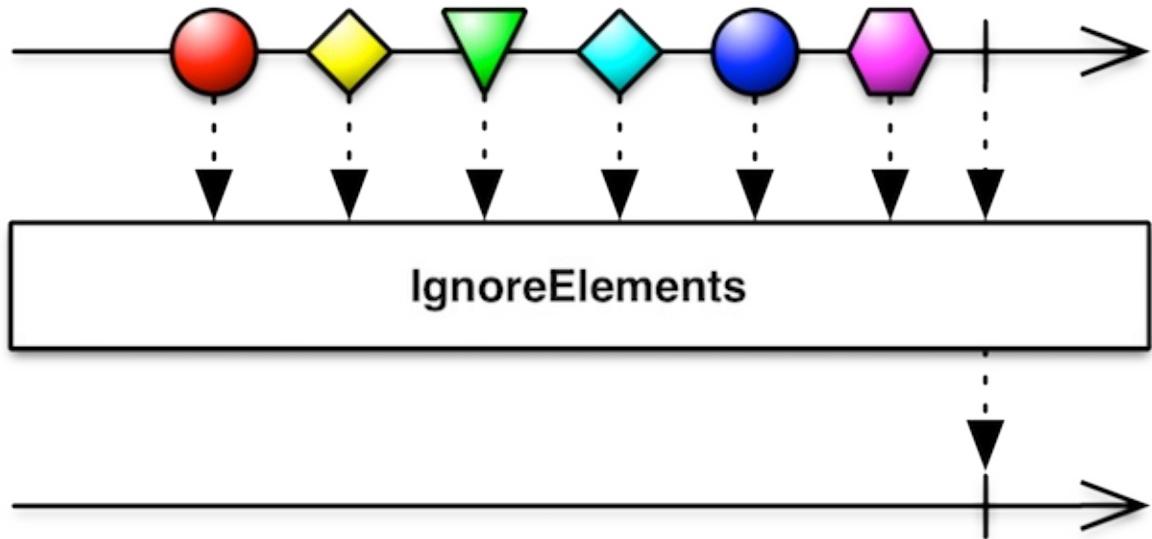


`groupBy` 操作符将源 Observable 分解为多个子 Observable , 然后将这些子 Observable 发送出来。

它会将元素通过某个键进行分组，然后将分组后的元素序列以 observable 的形态发送出来。

ignoreElements

忽略掉所有的元素，只发出 `error` 或 `completed` 事件

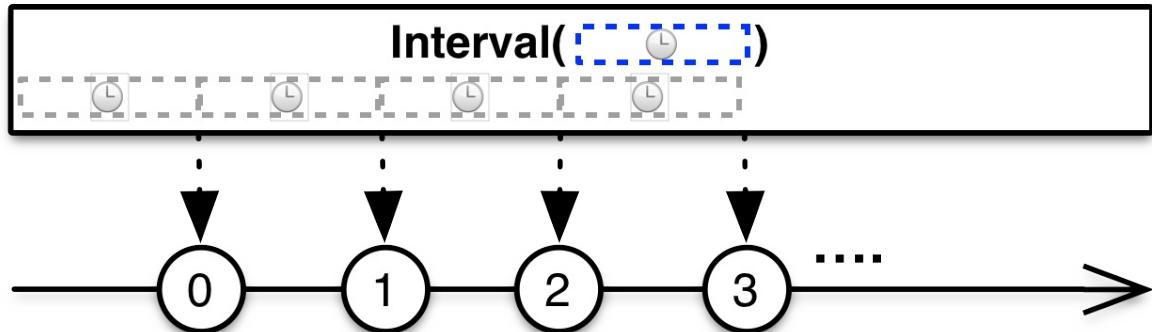


`ignoreElements` 操作符将阻止 `Observable` 发出 `next` 事件，但是允许他发出 `error` 或 `completed` 事件。

如果你并不关心 `Observable` 的任何元素，你只想知道 `observable` 在什么时候终止，那就可以使用 `ignoreElements` 操作符。

interval

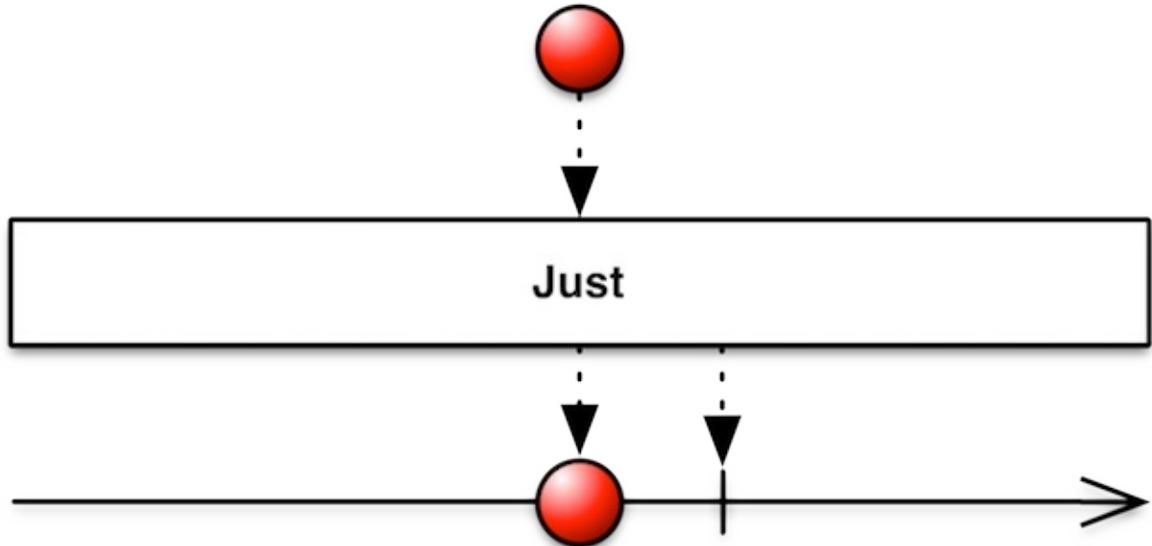
创建一个 `observable` 每隔一段时间，发出一个索引数



`interval` 操作符将创建一个 `Observable`，它每隔一段设定的时间，发出一个索引数的元素。它将发出无数个元素。

just

创建 `Observable` 发出唯一的一个元素



`just` 操作符将某一个元素转换为 `Observable`。

演示

一个序列只有唯一的元素 `0`:

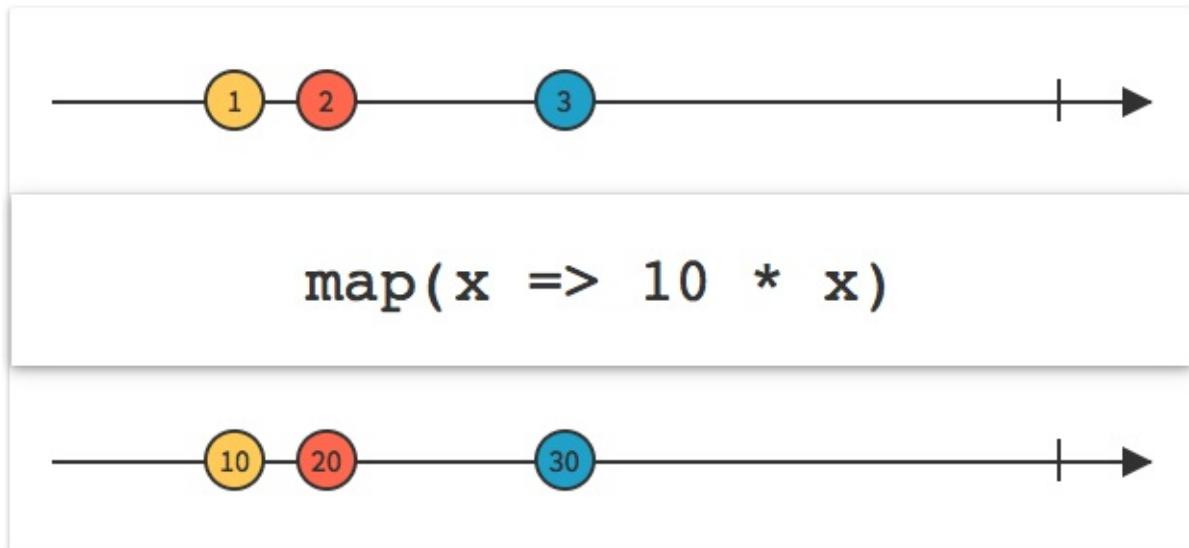
```
let id = Observable.just(0)
```

它相当于:

```
let id = Observable<Int>.create { observer in
    observer.onNext(0)
    observer.onCompleted()
    return Disposables.create()
}
```

map

通过一个转换函数，将 `Observable` 的每个元素转换一遍



`map` 操作符将源 `Observable` 的每个元素应用你提供的转换方法，然后返回含有转换结果的 `Observable`。

演示

```
let disposeBag = DisposeBag()
Observable.of(1, 2, 3)
    .map { $0 * 10 }
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

输出结果：

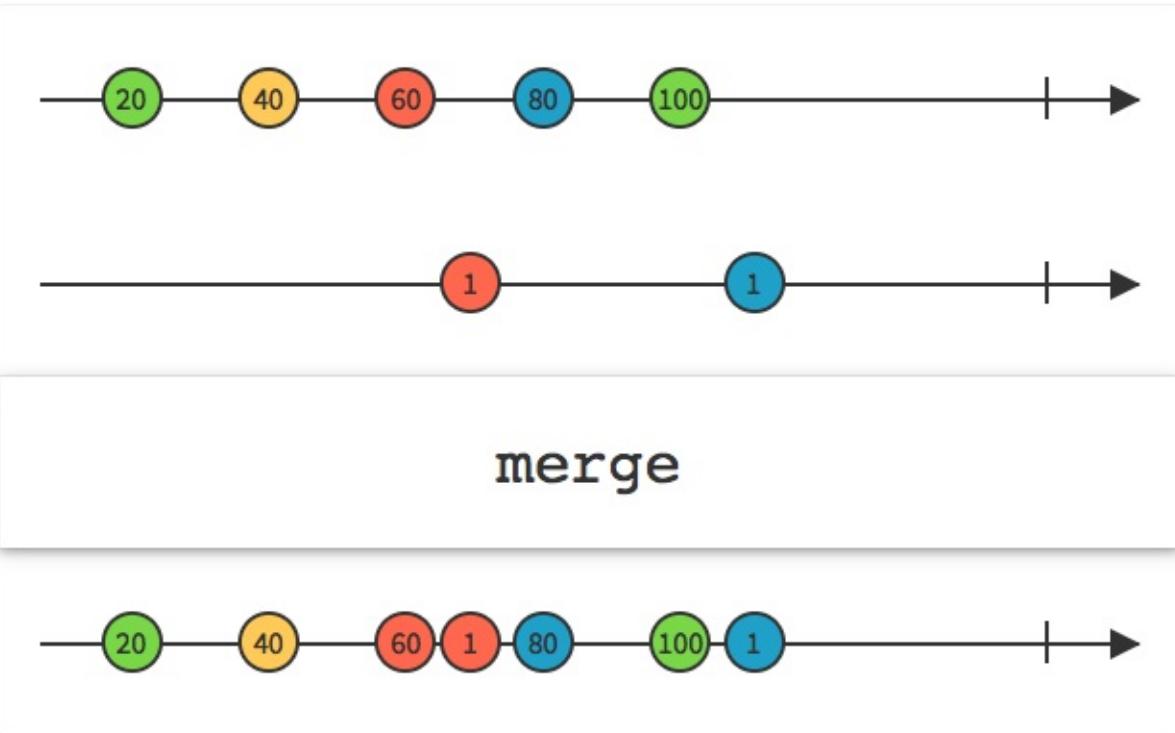
```
10
20
30
```

参考

- [flatMap](#)

merge

将多个 `Observables` 合并成一个



通过使用 `merge` 操作符你可以将多个 `Observables` 合并成一个，当某一个 `Observable` 发出一个元素时，他就将这个元素发出。

如果，某一个 `Observable` 发出一个 `onError` 事件，那么被合并的 `Observable` 也会将它发出，并且立即终止序列。

演示

```
let disposeBag = DisposeBag()

let subject1 = PublishSubject<String>()
let subject2 = PublishSubject<String>()

Observable.of(subject1, subject2)
    .merge()
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)

subject1.onNext("A")

subject1.onNext("B")
```

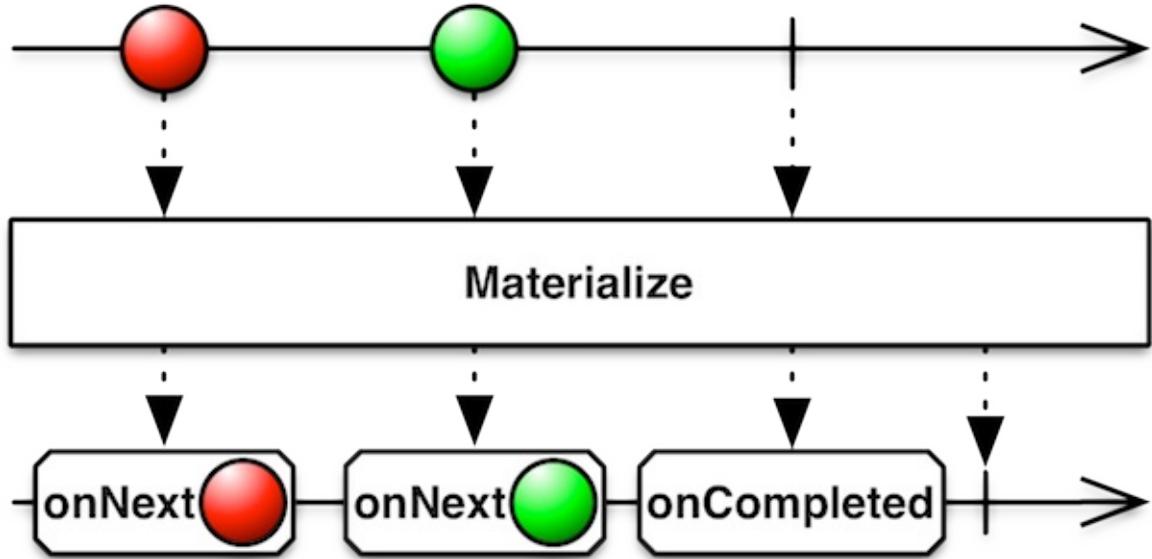
```
subject2.onNext("①")
subject2.onNext("②")
subject1.onNext(" ")
subject2.onNext("③")
```

输出结果：

```
A
B
①
②
③
```

materialize

将序列产生的事件，转换成元素



通常，一个有限的 `Observable` 将产生零个或者多个 `onNext` 事件，然后产生一个 `onCompleted` 或者 `onError` 事件。

`materialize` 操作符将 `Observable` 产生的这些事件全部转换成元素，然后发送出来。

never

创建一个永远不会发出元素的 `Observable`

Never



`never` 操作符将创建一个 `Observable`，这个 `Observable` 不会产生任何事件。

演示

创建一个不会产生任何事件的 `Observable`：

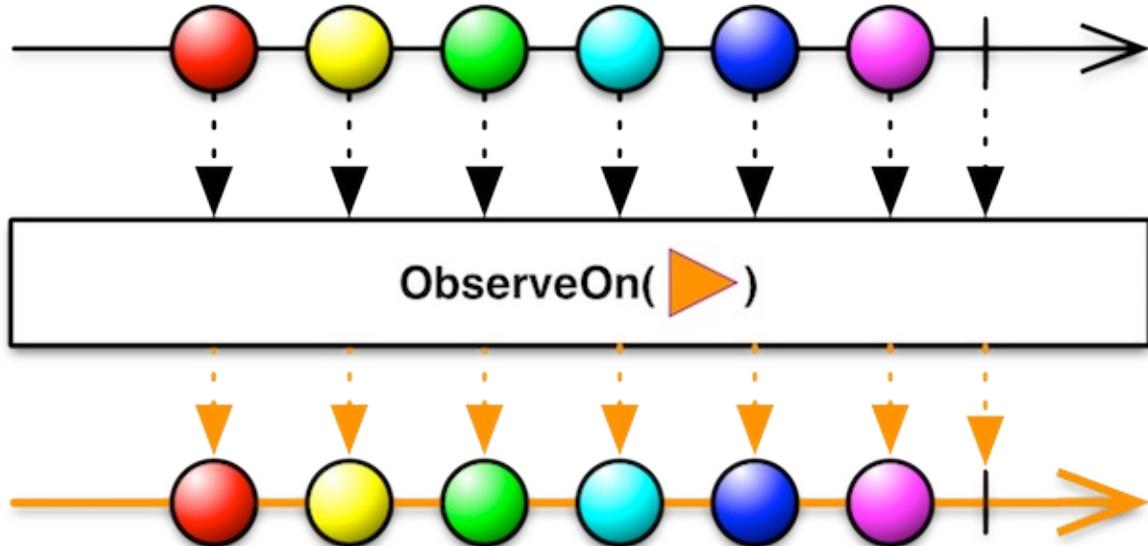
```
let id = Observable<Int>.never()
```

它相当于：

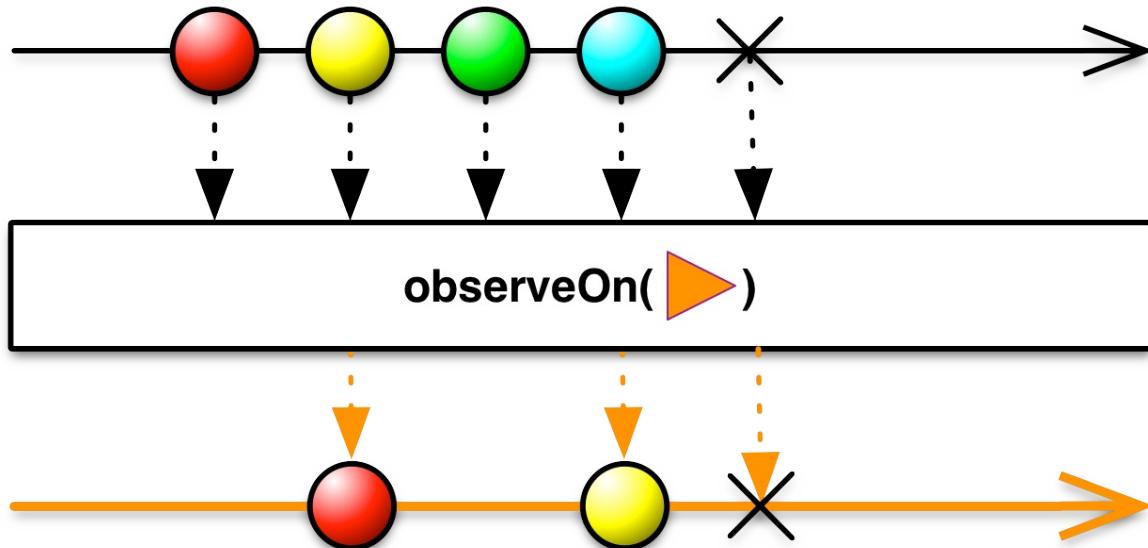
```
let id = Observable<Int>.create { observer in
    return Disposables.create()
}
```

observeOn

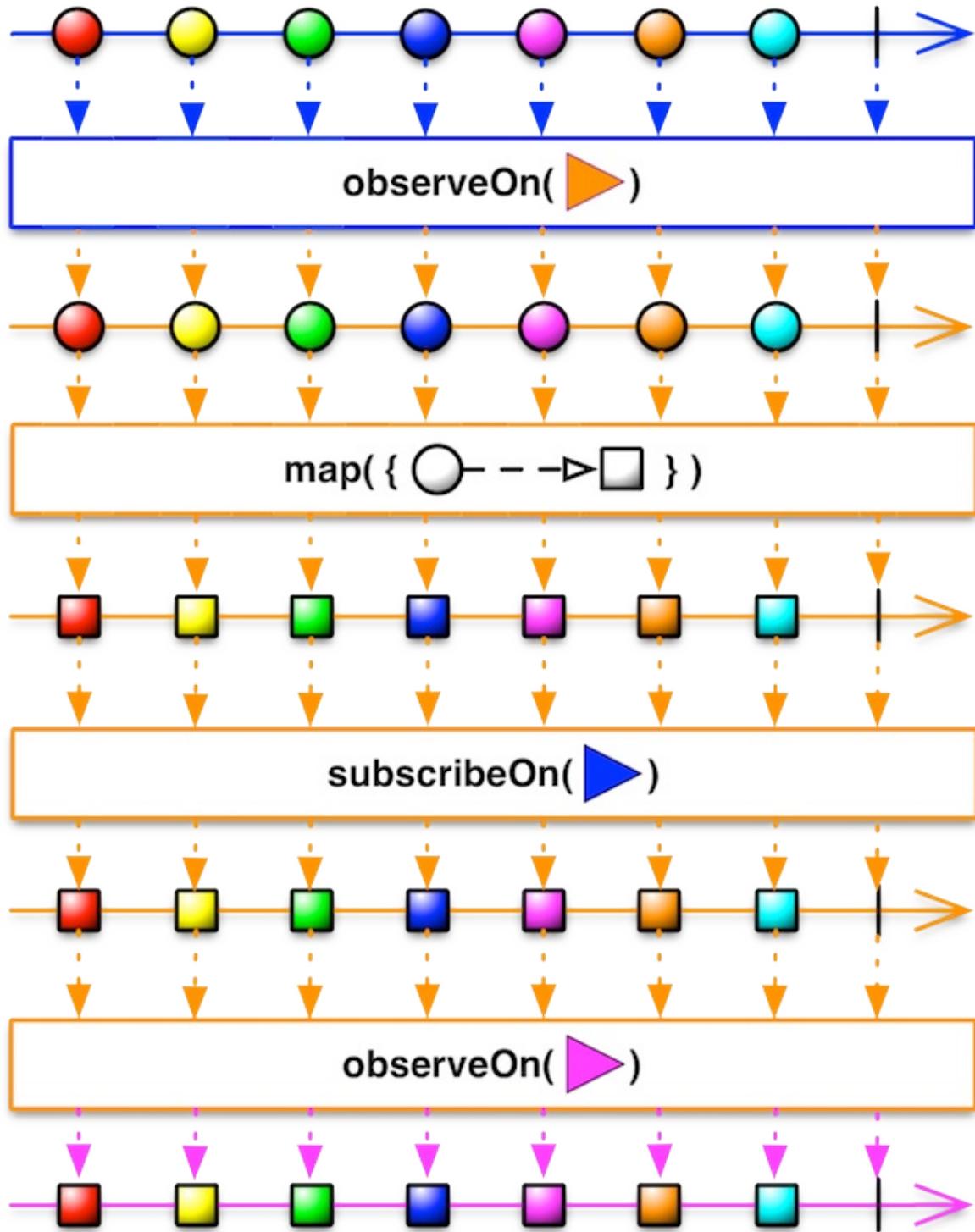
指定 `Observable` 在那个 `Scheduler` 发出通知



ReactiveX 使用 `Scheduler` 来让 `Observable` 支持多线程。你可以使用 `observeOn` 操作符，来指示 `Observable` 在哪个 `Scheduler` 发出通知。



注意⚠：一旦产生了 `onError` 事件，`observeOn` 操作符将立即转发。他不会等待 `onError` 之前的事件全部被收到。这意味着 `onError` 事件可能会跳过一些元素提前发送出去，如上图所示。



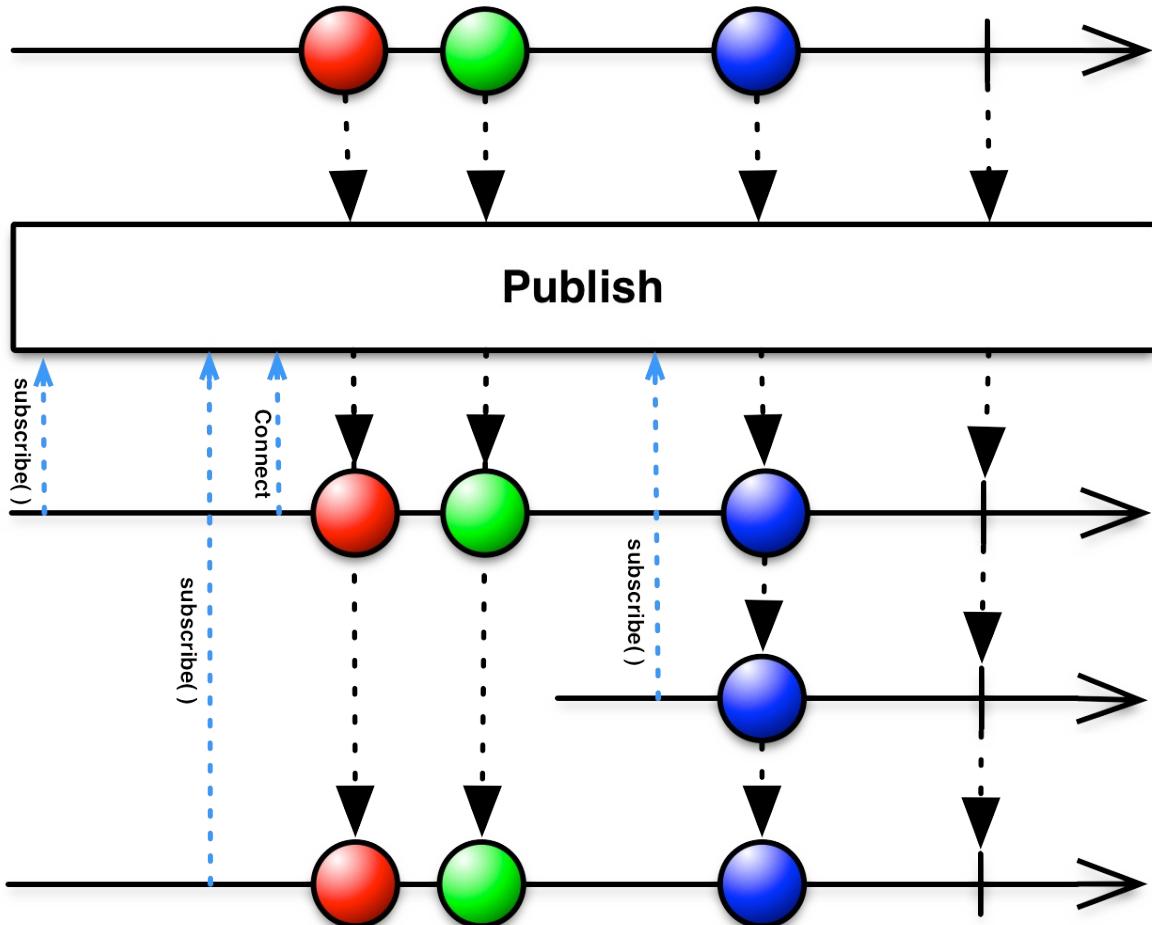
`subscribeOn` 操作符非常相似。它指示 `Observable` 在哪个 `Scheduler` 发出执行。

默认情况下，`Observable` 创建，应用操作符以及发出通知都会在 `Subscribe` 方法调用的 `Scheduler` 执行。`subscribeOn` 操作符将改变这种行为，它会指定一个不同的 `Scheduler` 来让 `Observable` 执行，`observeOn` 操作符将指定一个不同的 `Scheduler` 来让 `observable` 通知观察者。

如上图所示，`subscribeOn` 操作符指定 `Observable` 在那个 `Scheduler` 开始执行，无论它处于链的那个位置。另一方面 `observeOn` 将决定后面的方法在哪个 `Scheduler` 运行。因此，你可能会多次调用 `observeOn` 来决定某些操作符在哪个线程运行。

publish

将 `Observable` 转换为可被连接的 `Observable`



`publish` 会将 `Observable` 转换为可被连接的 `Observable`。可被连接的 `Observable` 和普通的 `Observable` 十分相似，不过在被订阅后不会发出元素，直到 `connect` 操作符被应用为止。这样一来你可以控制 `Observable` 在什么时候开始发出元素。

演示

```
let intSequence = Observable<Int>.interval(1, scheduler: MainScheduler.instance)
    .publish()

_ = intSequence
    .subscribe(onNext: { print("Subscription 1:, Event: \"\($0)\"") })

DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
    _ = intSequence.connect()
}
```

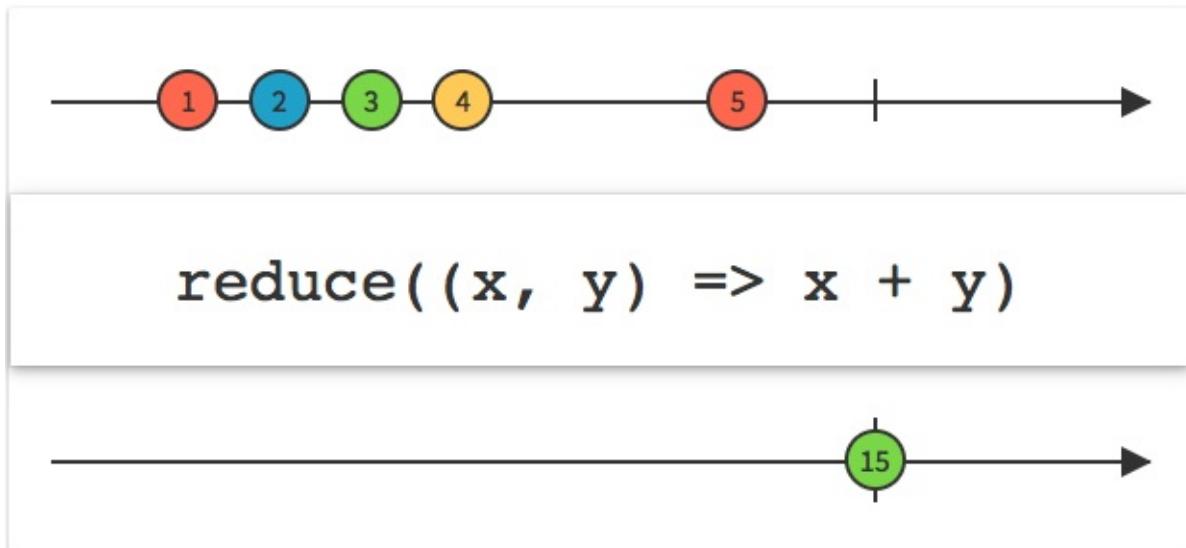
```
DispatchQueue.main.asyncAfter(deadline: .now() + 4) {  
    _ = intSequence  
        .subscribe(onNext: { print("Subscription 2:, Event: \"\($0)\"") })  
}  
  
DispatchQueue.main.asyncAfter(deadline: .now() + 6) {  
    _ = intSequence  
        .subscribe(onNext: { print("Subscription 3:, Event: \"\($0)\"") })  
}
```

输出结果：

```
Subscription 1:, Event: 0  
Subscription 1:, Event: 1  
Subscription 2:, Event: 1  
Subscription 1:, Event: 2  
Subscription 2:, Event: 2  
Subscription 1:, Event: 3  
Subscription 2:, Event: 3  
Subscription 3:, Event: 3  
Subscription 1:, Event: 4  
Subscription 2:, Event: 4  
Subscription 3:, Event: 4  
Subscription 1:, Event: 5  
Subscription 2:, Event: 5  
Subscription 3:, Event: 5  
Subscription 1:, Event: 6  
Subscription 2:, Event: 6  
Subscription 3:, Event: 6  
...
```

reduce

持续的将 Observable 的每一个元素应用一个函数，然后发出最终结果



reduce 操作符将对第一个元素应用一个函数。然后，将结果作为参数填入到第二个元素的应用函数中。以此类推，直到遍历完全部的元素后发出最终结果。

这种操作符在其他地方有时候被称作是 `accumulator`, `aggregate`, `compress`, `fold` 或者 `inject`。

演示

```
let disposeBag = DisposeBag()

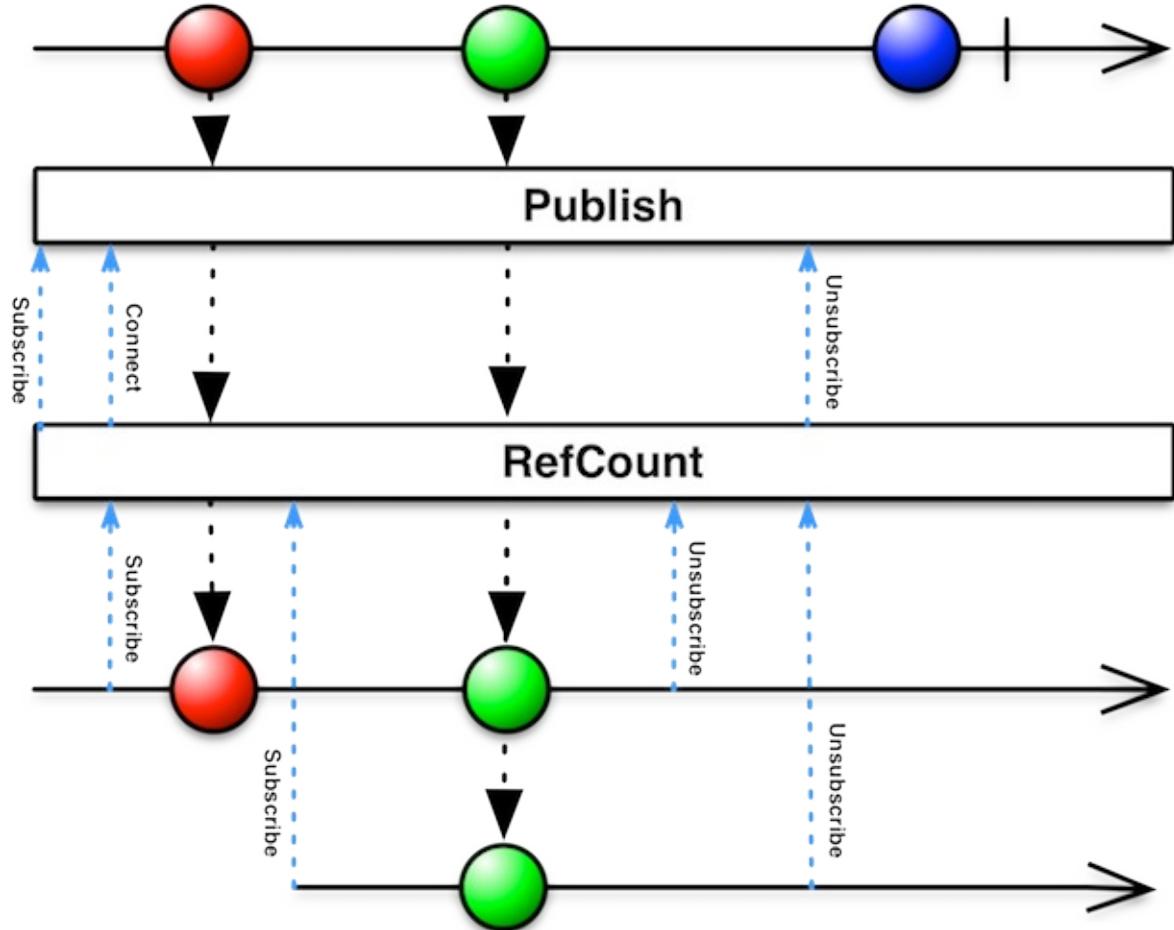
Observable.of(10, 100, 1000)
    .reduce(1, accumulator: +)
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

输出结果：

1111

refCount

将可被连接的 `Observable` 转换为普通 `Observable`

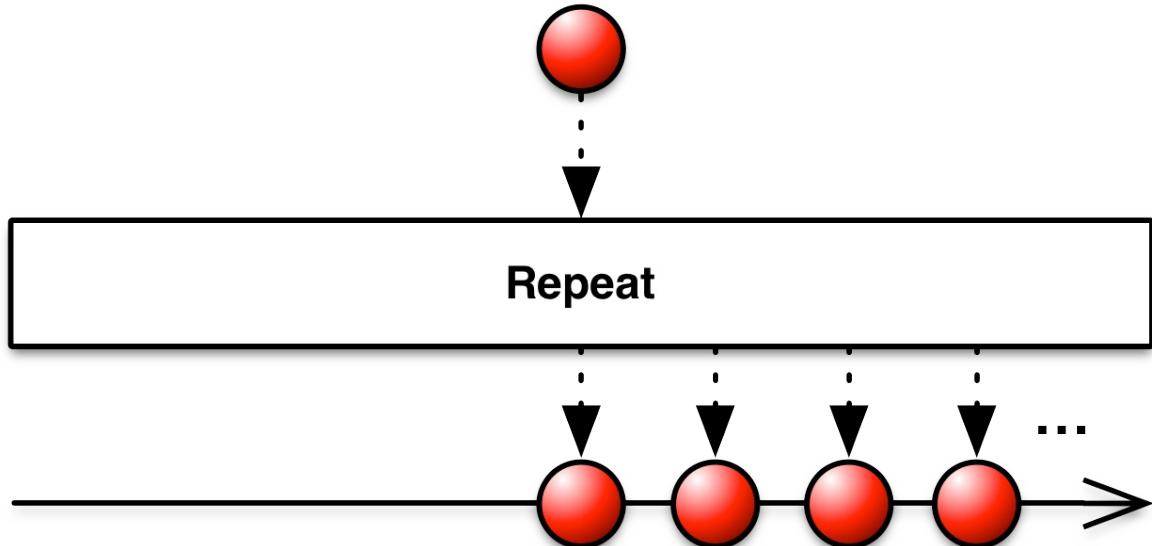


可被连接的 `Observable` 和普通的 `Observable` 十分相似，不过在被订阅后不会发出元素，直到 `connect` 操作符被应用为止。这样一来你可以控制 `Observable` 在什么时候开始发出元素。

`refCount` 操作符将自动连接和断开可被连接的 `observable`。它将可被连接的 `Observable` 转换为普通 `Observable`。当第一个观察者对它订阅时，那么底层的 `Observable` 将被连接。当最后一个观察者离开时，那么底层的 `observable` 将被断开连接。

repeatElement

创建 `Observable` 重复的发出某个元素



`repeatElement` 操作符将创建一个 `Observable`，这个 `Observable` 将无止尽的发出同一个元素。

演示

创建 `Observable` 重复的发出某个元素：

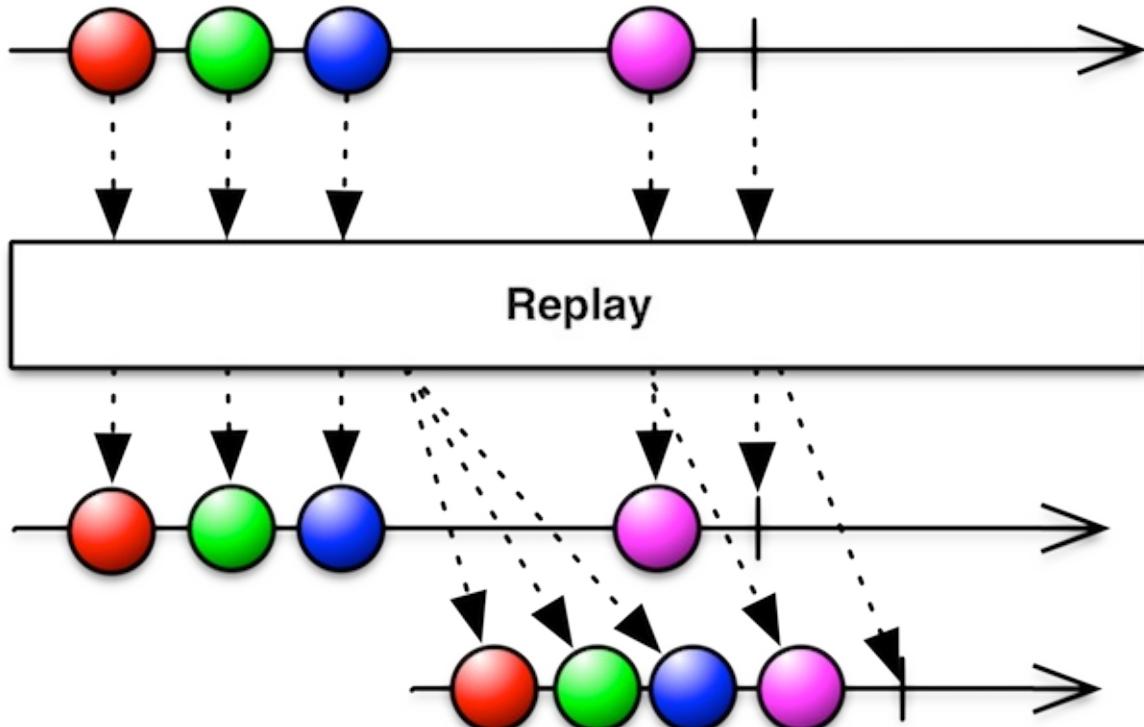
```
let id = Observable.repeatElement(0)
```

它相当于：

```
let id = Observable<Int>.create { observer in
    observer.onNext(0)
    observer.onNext(0)
    observer.onNext(0)
    observer.onNext(0)
    ...
    // 无数次
    return Disposables.create()
}
```

replay

确保观察者接收到同样的序列，即使是在 `Observable` 发出元素后才订阅



可被连接的 `Observable` 和普通的 `Observable` 十分相似，不过在被订阅后不会发出元素，直到 `connect` 操作符被应用为止。这样一来你可以控制 `Observable` 在什么时候开始发出元素。

`replay` 操作符将 `Observable` 转换为可被连接的 `Observable`，并且这个可被连接的 `Observable` 将缓存最新的 n 个元素。当有新的观察者对它进行订阅时，它就把这些被缓存的元素发送给观察者。

演示

```
let intSequence = Observable<Int>.interval(1, scheduler: MainScheduler.instance)
    .replay(5)

_ = intSequence
    .subscribe(onNext: { print("Subscription 1:, Event: \"\($0)\"") })

DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
    _ = intSequence.connect()
}

DispatchQueue.main.asyncAfter(deadline: .now() + 4) {
    _ = intSequence
```

```
.subscribe(onNext: { print("Subscription 2:, Event: \"\($0)\"") })
}

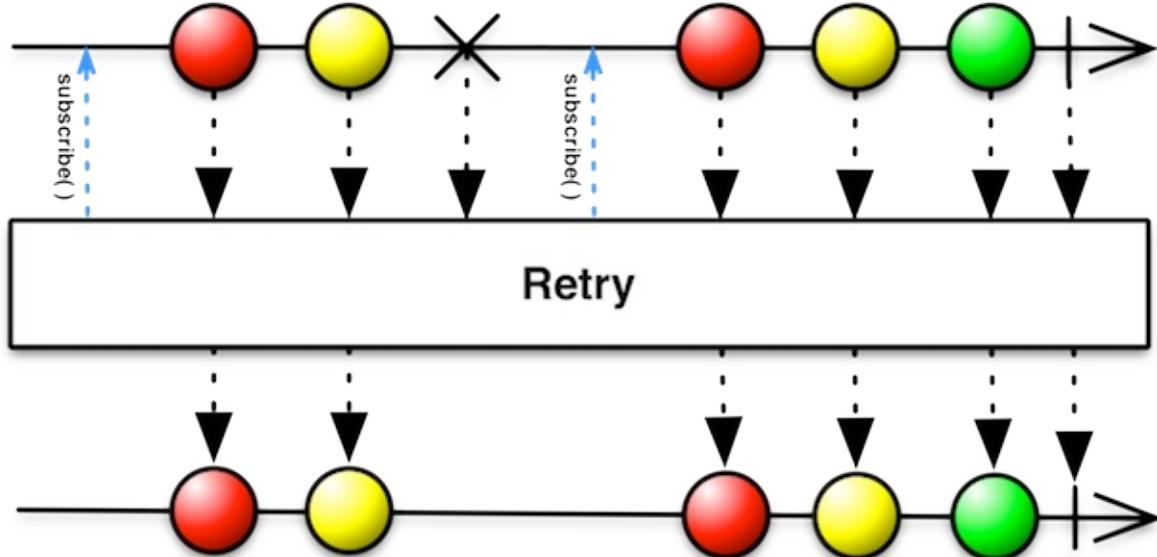
DispatchQueue.main.asyncAfter(deadline: .now() + 8) {
    _ = intSequence
    .subscribe(onNext: { print("Subscription 3:, Event: \"\($0)\"") })
}
```

输出结果：

```
Subscription 1:, Event: 0
Subscription 2:, Event: 0
Subscription 1:, Event: 1
Subscription 2:, Event: 1
Subscription 1:, Event: 2
Subscription 2:, Event: 2
Subscription 1:, Event: 3
Subscription 2:, Event: 3
Subscription 1:, Event: 4
Subscription 2:, Event: 4
Subscription 3:, Event: 0
Subscription 3:, Event: 1
Subscription 3:, Event: 2
Subscription 3:, Event: 3
Subscription 3:, Event: 4
Subscription 1:, Event: 5
Subscription 2:, Event: 5
Subscription 3:, Event: 5
Subscription 1:, Event: 6
Subscription 2:, Event: 6
Subscription 3:, Event: 6
...
...
```

retry

如果源 `Observable` 产生一个错误事件，重新对它进行订阅，希望它不会再次产生错误



`retry` 操作符将不会将 `error` 事件，传递给观察者，然而，它会从新订阅源 `Observable`，给这个 `Observable` 一个重试的机会，让它有机会不产生 `error` 事件。`retry` 总是对观察者发出 `next` 事件，即便源序列产生了一个 `error` 事件，所以这样可能会产生重复的元素（如上图所示）。

演示 1

```
let disposeBag = DisposeBag()
var count = 1

let sequenceThatErrors = Observable<String>.create { observer in
    observer.onNext(" ")
    observer.onNext(" ")
    observer.onNext(" ")

    if count == 1 {
        observer.onError(TestError.test)
        print("Error encountered")
        count += 1
    }

    observer.onNext(" ")
    observer.onNext(" ")
    observer.onNext(" ")
    observer.onCompleted()
}
```

```

    return Disposables.create()
}

sequenceThatErrors
    .retry()
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)

```

输出结果：

```
Error encountered
```

演示 2

```

let disposeBag = DisposeBag()
var count = 1

let sequenceThatErrors = Observable<String>.create { observer in
    observer.onNext(" ")
    observer.onNext(" ")
    observer.onNext(" ")

    if count < 5 {
        observer.onError(TestError.test)
        print("Error encountered")
        count += 1
    }

    observer.onNext(" ")
    observer.onNext(" ")
    observer.onNext(" ")
    observer.onCompleted()

    return Disposables.create()
}

sequenceThatErrors
    .retry(3)
    .subscribe(onNext: { print($0) })

```

```
.disposed(by: disposeBag)
```

输出结果：

```
Error encountered
```

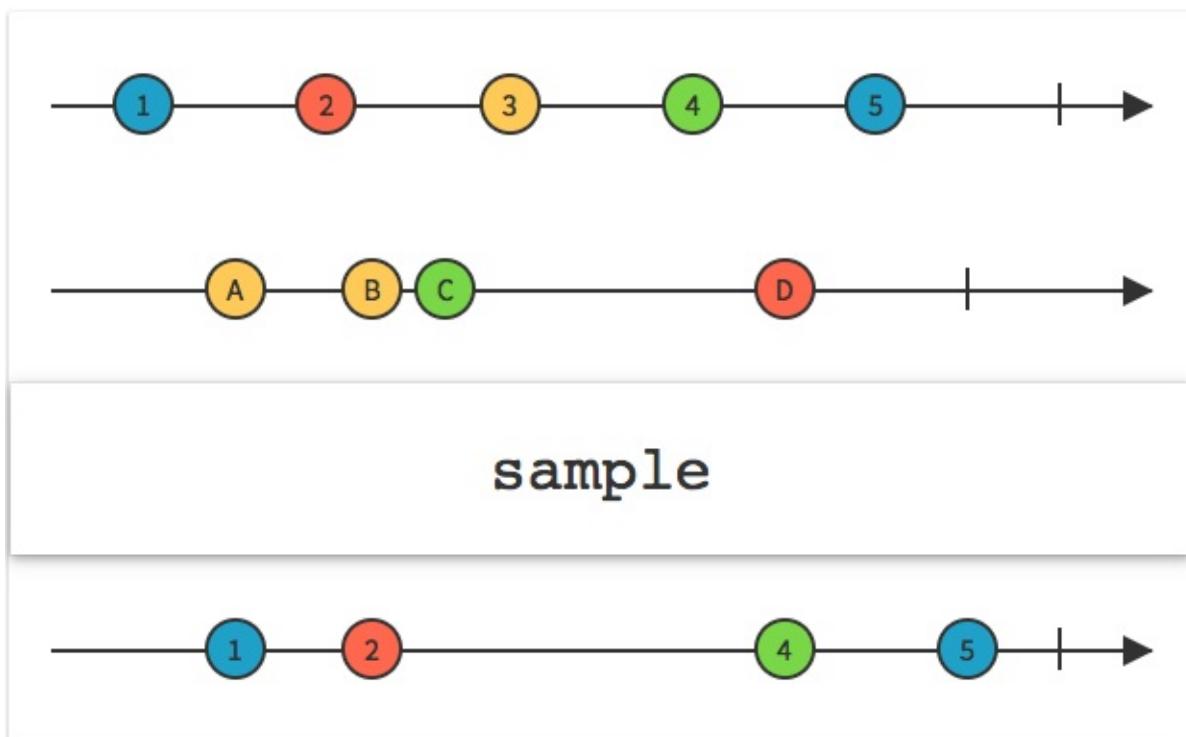
```
Error encountered
```

```
Error encountered
```

```
Unhandled error happened: test  
subscription called from:
```

sample

不定期的对 `Observable` 取样



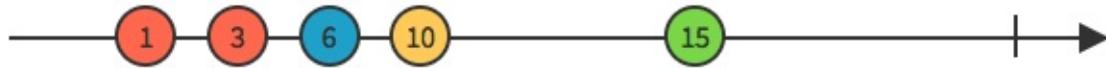
`sample` 操作符将不定期的对源 `Observable` 进行取样操作。通过第二个 `Observable` 来控制取样时机。一旦第二个 `Observable` 发出一个元素，就从源 `observable` 中取出最后产生的元素。

scan

持续的将 `observable` 的每一个元素应用一个函数，然后发出每一次函数返回的结果



`scan((x, y) => x + y)`



`scan` 操作符将对第一个元素应用一个函数，将结果作为第一个元素发出。然后，将结果作为参数填入到第二个元素的应用函数中，创建第二个元素。以此类推，直到遍历完全部的元素。

这种操作符在其他地方有时候被称作是 **accumulator**。

演示

```
let disposeBag = DisposeBag()

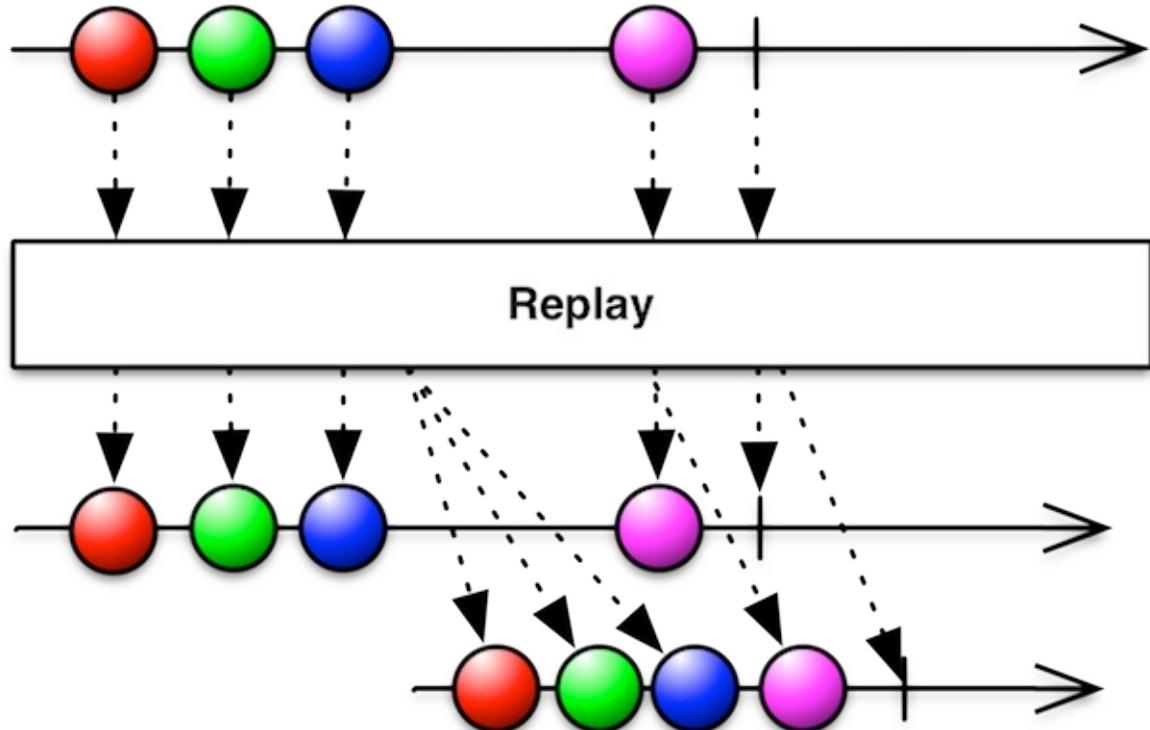
Observable.of(10, 100, 1000)
    .scan(1) { aggregateValue, newValue in
        aggregateValue + newValue
    }
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

输出结果：

```
11
111
1111
```


shareReplay

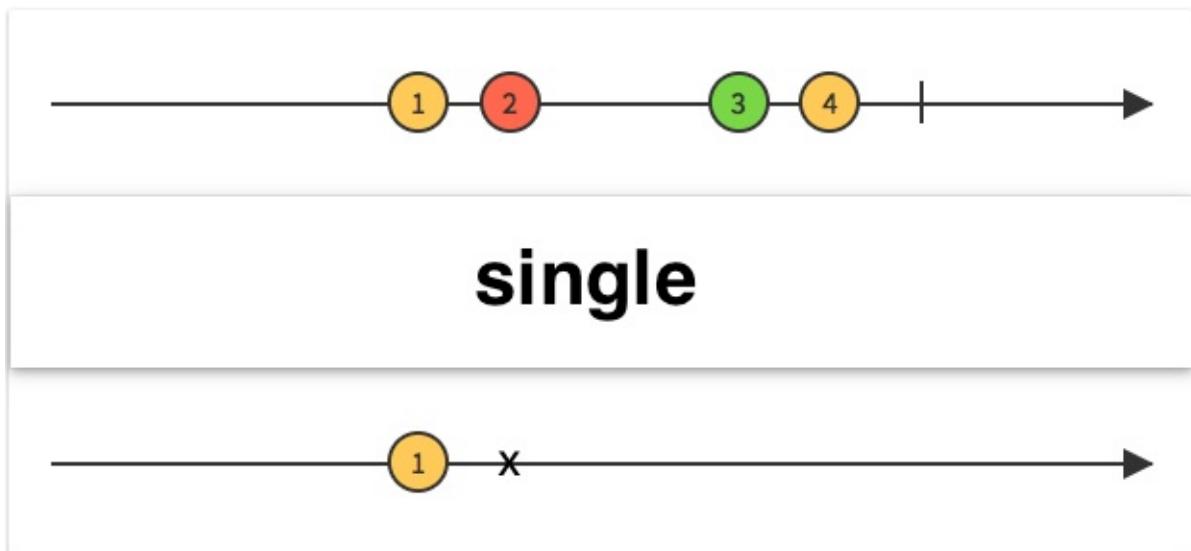
使观察者共享 `Observable`，观察者会立即收到最新的元素，即使这些元素是在订阅前产生的



`shareReplay` 操作符将使得观察者共享 `Observable`，并且缓存最新的 n 个元素，将这些元素直接发送给新的观察者。

single

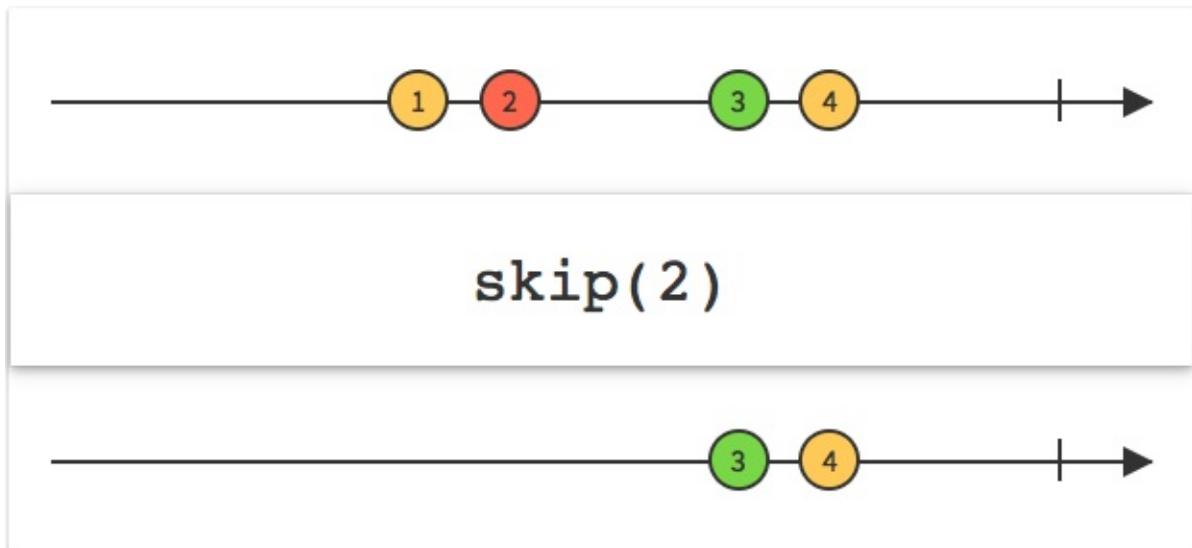
限制 `Observable` 只有一个元素，否出发出一个 `error` 事件



`single` 操作符将限制 `Observable` 只产生一个元素。如果 `Observable` 只有一个元素，它将镜像这个 `Observable`。如果 `Observable` 没有元素或者元素数量大于一，它将产生一个 `error` 事件。

skip

跳过 `Observable` 中头 n 个元素



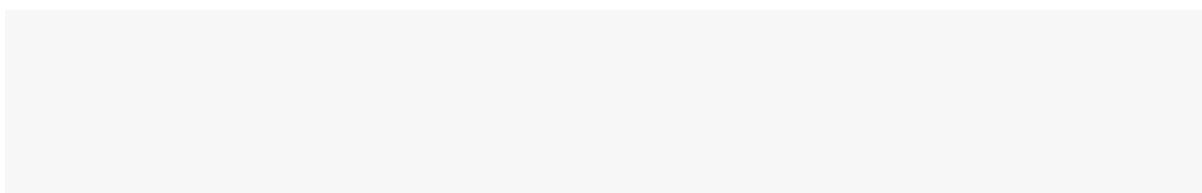
`skip` 操作符可以让你跳过 `Observable` 中头 n 个元素，只关注后面的元素。

演示

```
let disposeBag = DisposeBag()

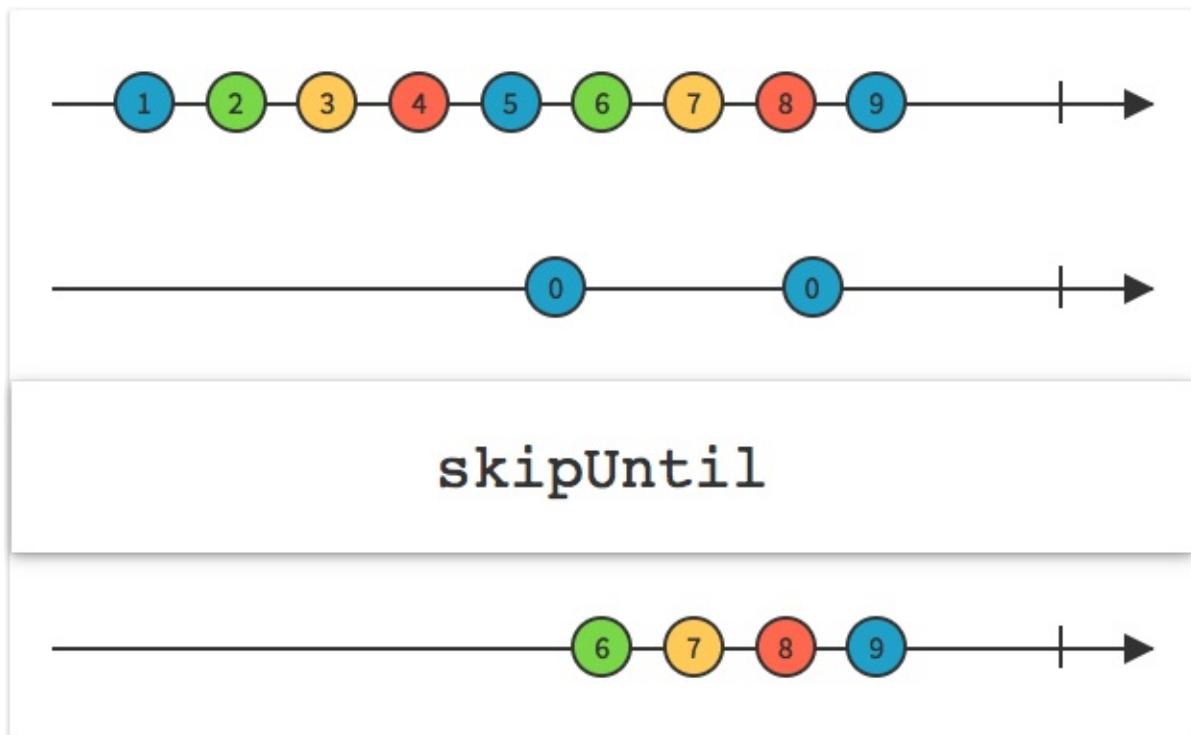
Observable.of(" ", " ", " ", " ")
    .skip(2)
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

输出结果：



skipUntil

跳过 `Observable` 中头几个元素，直到另一个 `observable` 发出一个元素



`skipUntil` 操作符可以让你忽略源 `Observable` 中头几个元素，直到另一个 `observable` 发出一个元素后，它才镜像源 `Observable`。

演示

```
let disposeBag = DisposeBag()

let sourceSequence = PublishSubject<String>()
let referenceSequence = PublishSubject<String>()

sourceSequence
    .skipUntil(referenceSequence)
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)

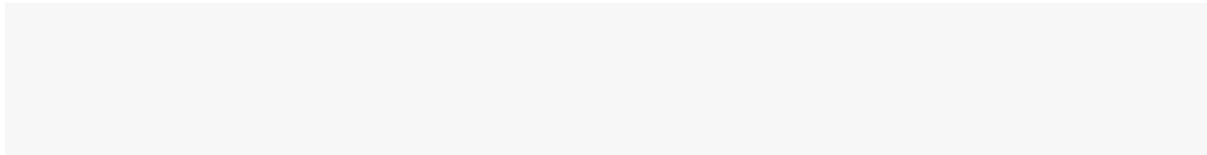
sourceSequence.onNext(" ")
sourceSequence.onNext(" ")
sourceSequence.onNext(" ")

referenceSequence.onNext(" ")

sourceSequence.onNext(" ")
```

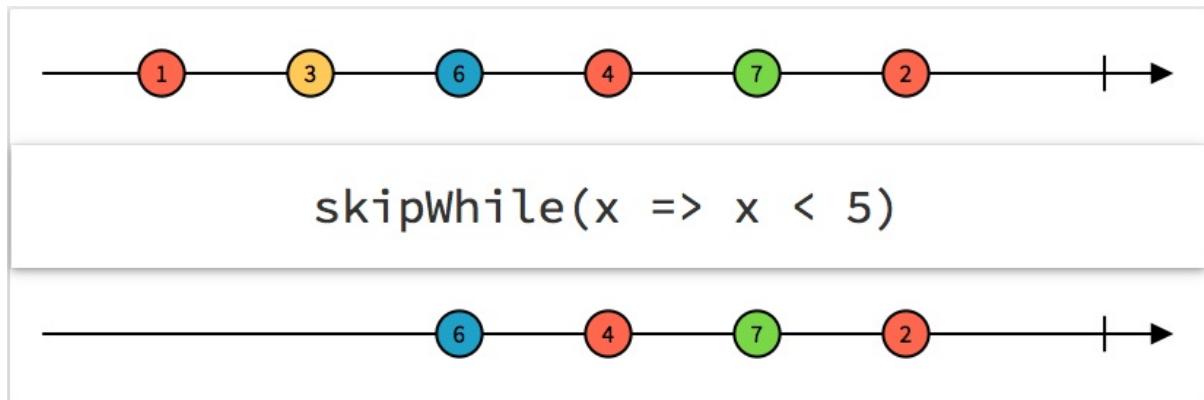
```
sourceSequence.onNext(" ")  
sourceSequence.onNext(" ")
```

输出结果：



skipWhile

跳过 `Observable` 中头几个元素，直到元素的判定为否



`skipWhile` 操作符可以让你忽略源 `Observable` 中头几个元素，直到元素的判定为否后，它才镜像源 `Observable`。

演示

```
let disposeBag = DisposeBag()

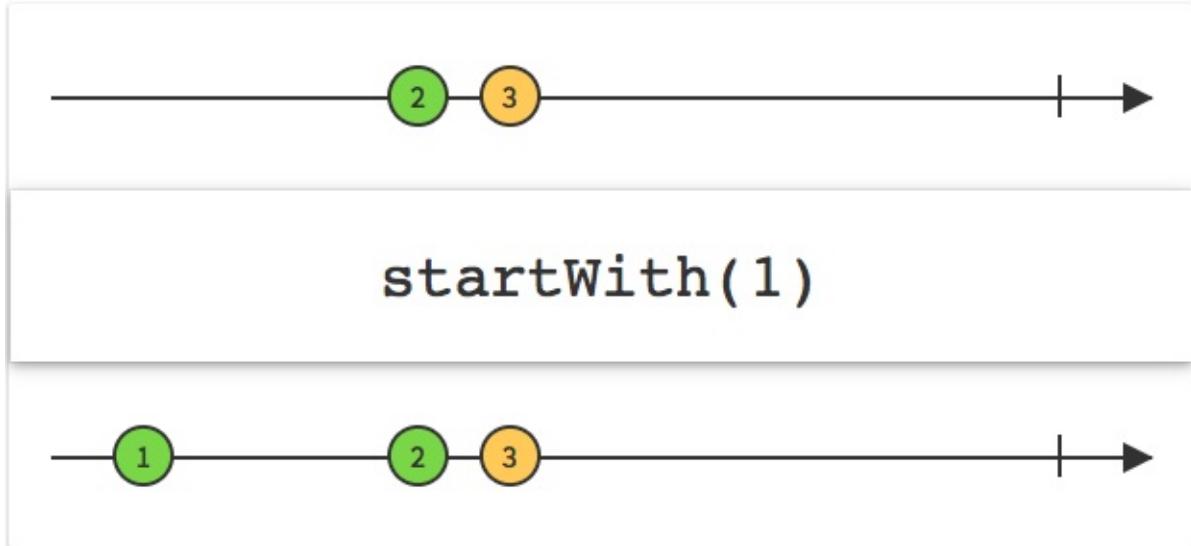
Observable.of(1, 2, 3, 4, 3, 2, 1)
    .skipWhile { $0 < 4 }
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

输出结果：

```
4
3
2
1
```

startsWith

将一些元素插入到序列的头部



`startsWith` 操作符会在 `Observable` 头部插入一些元素。

(如果你想在尾部加入一些元素可以用[concat](#))

演示

```
let disposeBag = DisposeBag()

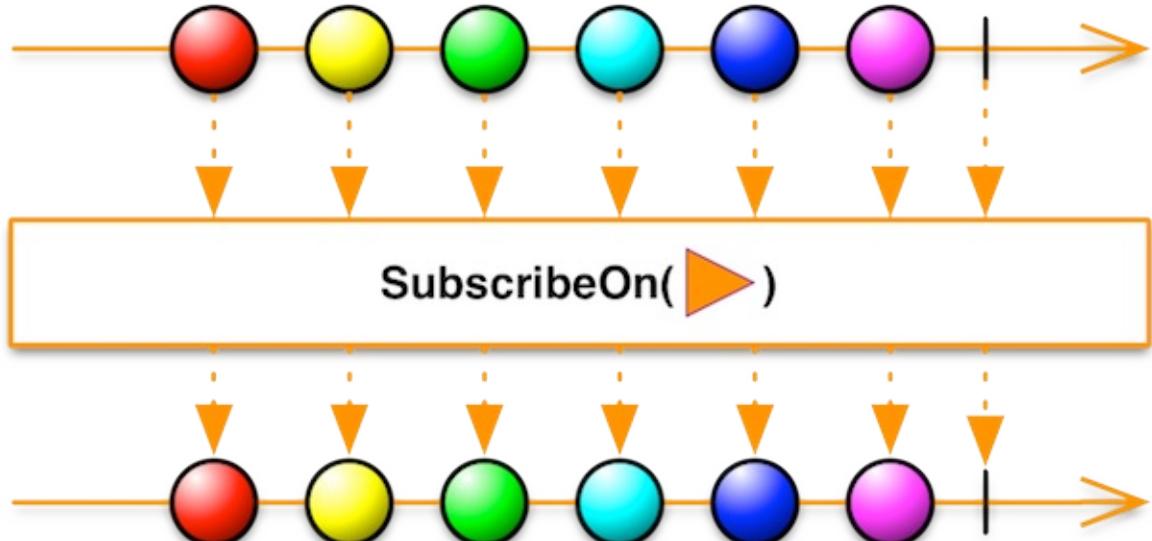
Observable.of(" ", " ", " ", " ")
    .startWith("1")
    .startWith("2")
    .startWith("3", "A", "B")
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

输出结果：

```
3
A
B
2
1
```

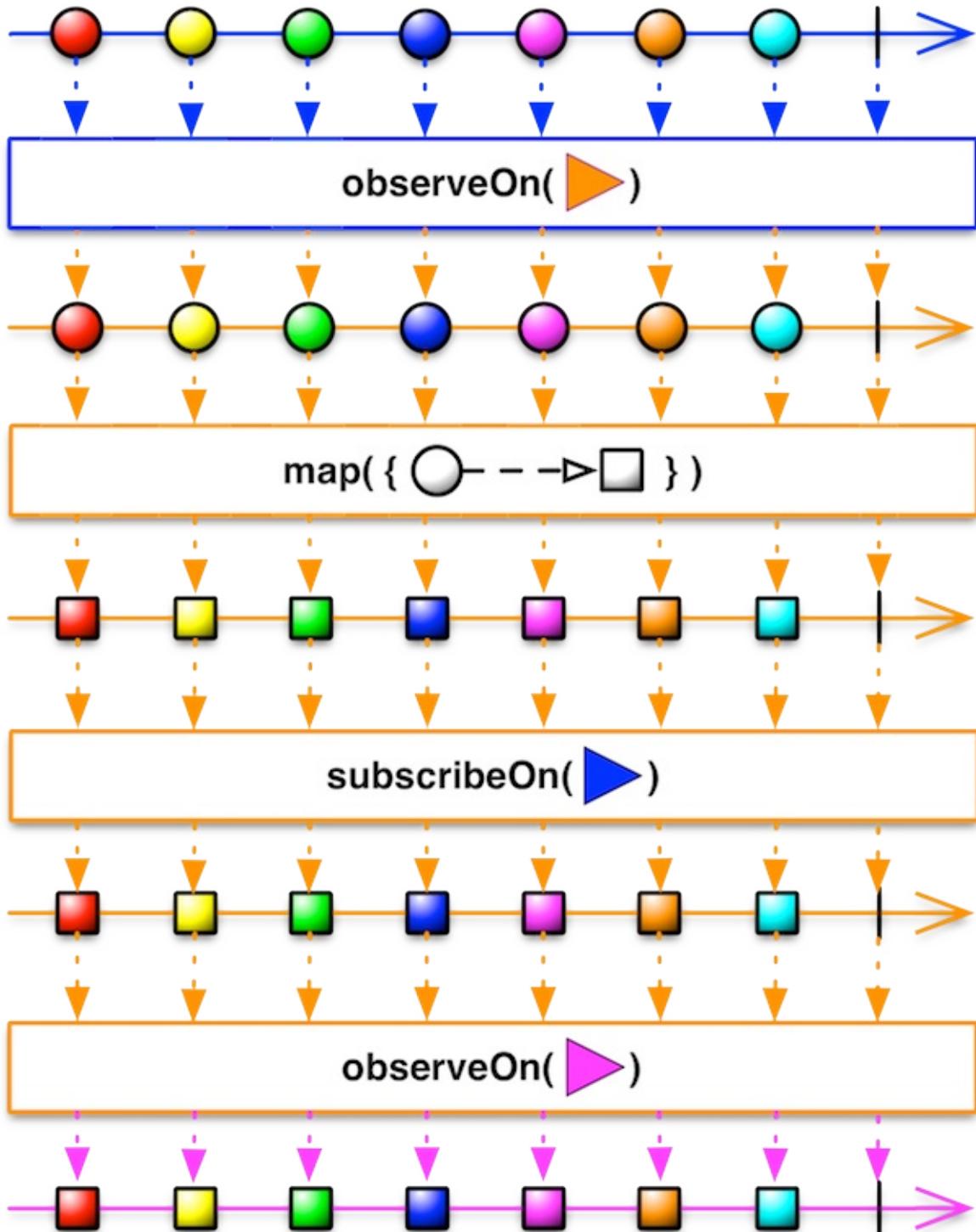

subscribeOn

指定 `Observable` 在那个 `Scheduler` 执行



ReactiveX 使用 `Scheduler` 来让 `Observable` 支持多线程。你可以使用 `subscribeOn` 操作符，来指示 `Observable` 在哪个 `Scheduler` 执行。

`observeOn` 操作符非常相似。它指示 `Observable` 在哪个 `Scheduler` 发出通知。

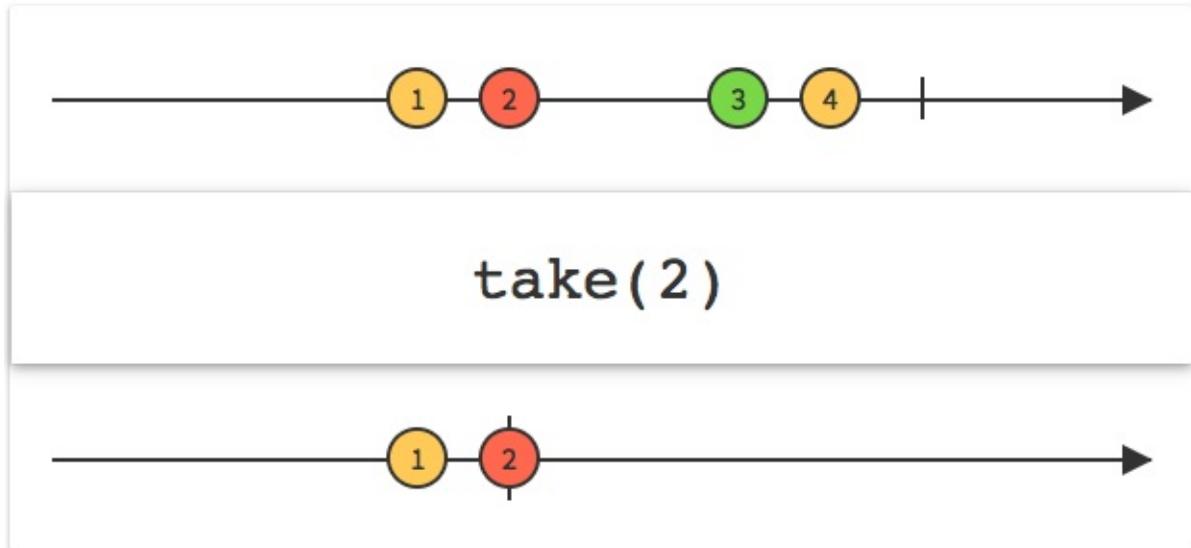


默认情况下，`Observable` 创建，应用操作符以及发出通知都会在 `Subscribe` 方法调用的 `Scheduler` 执行。`subscribeOn` 操作符将改变这种行为，它会指定一个不同的 `Scheduler` 来让 `observable` 执行，`observeOn` 操作符将指定一个不同的 `Scheduler` 来让 `observable` 通知观察者。

如上图所示，`subscribeOn` 操作符指定 `Observable` 在那个 `Scheduler` 开始执行，无论它处于链的那个位置。另一方面 `observeOn` 将决定后面的方法在哪个 `Scheduler` 运行。因此，你可能会多次调用 `observeOn` 来决定某些操作符在哪个线程运行。

take

仅仅从 `observable` 中发出头 n 个元素



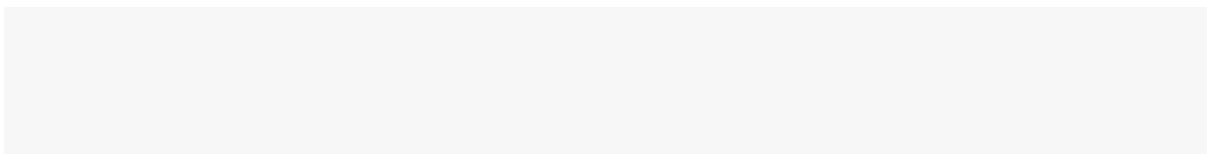
通过 `take` 操作符你可以只发出头 n 个元素。并且忽略掉后面的元素，直接结束序列。

演示

```
let disposeBag = DisposeBag()

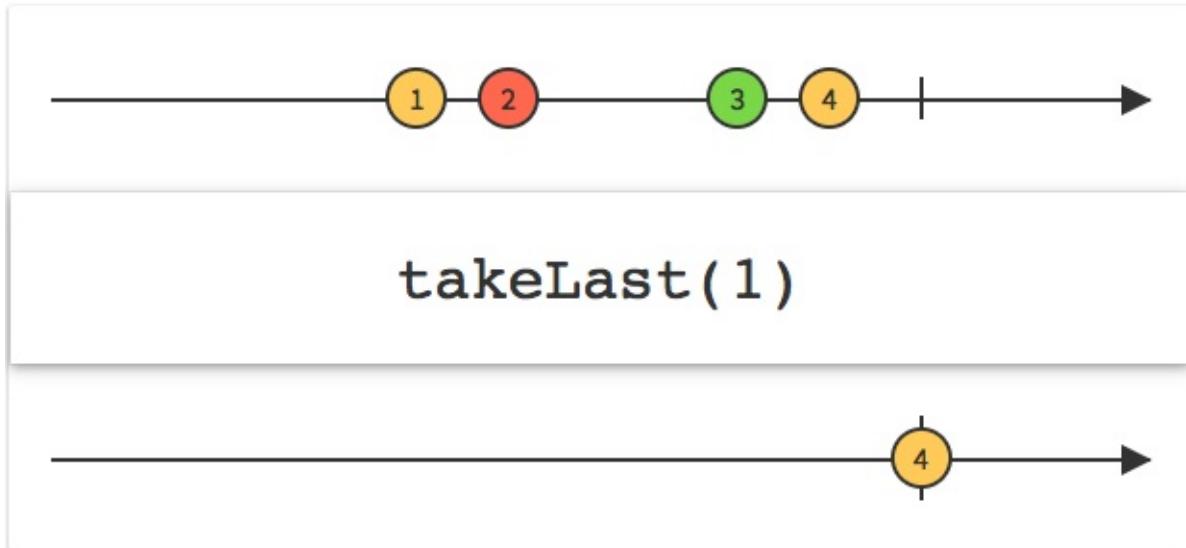
Observable.of(" ", " ", " ", " ")
    .take(3)
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

输出结果：



takeLast

仅仅从 `observable` 中发出尾部 n 个元素



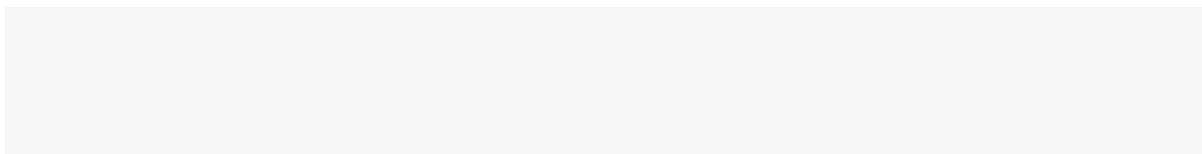
通过 `takeLast` 操作符你可以只发出尾部 n 个元素。并且忽略掉前面的元素。

演示

```
let disposeBag = DisposeBag()

Observable.of(" ", " ", " ", " ")
    .takeLast(3)
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

输出结果：



takeUntil

忽略掉在第二个 `Observable` 产生事件后发出的那部分元素



takeUntil



`takeUntil` 操作符将镜像源 `Observable`，它同时观测第二个 `Observable`。一旦第二个 `Observable` 发出一个元素或者产生一个终止事件，那个镜像的 `Observable` 将立即终止。

演示

```
let disposeBag = DisposeBag()

let sourceSequence = PublishSubject<String>()
let referenceSequence = PublishSubject<String>()

sourceSequence
    .takeUntil(referenceSequence)
    .subscribe { print($0) }
    .disposed(by: disposeBag)

sourceSequence.onNext(" ")
sourceSequence.onNext(" ")
sourceSequence.onNext(" ")

referenceSequence.onNext(" ")
```

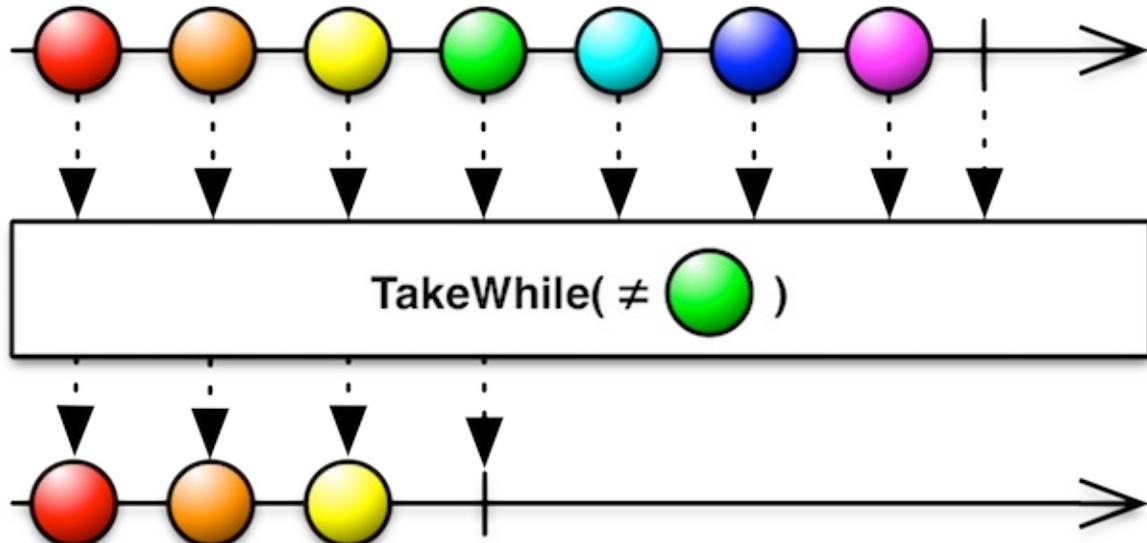
```
sourceSequence.onNext(" ")
sourceSequence.onNext(" ")
sourceSequence.onNext(" ")
```

输出结果：

```
next()
next()
next()
completed
```

takeWhile

镜像一个 `Observable` 直到某个元素的判定为 `false`



`takeWhile` 操作符将镜像源 `Observable` 直到某个元素的判定为 `false`。此时，这个镜像的 `Observable` 将立即终止。

演示

```
let disposeBag = DisposeBag()

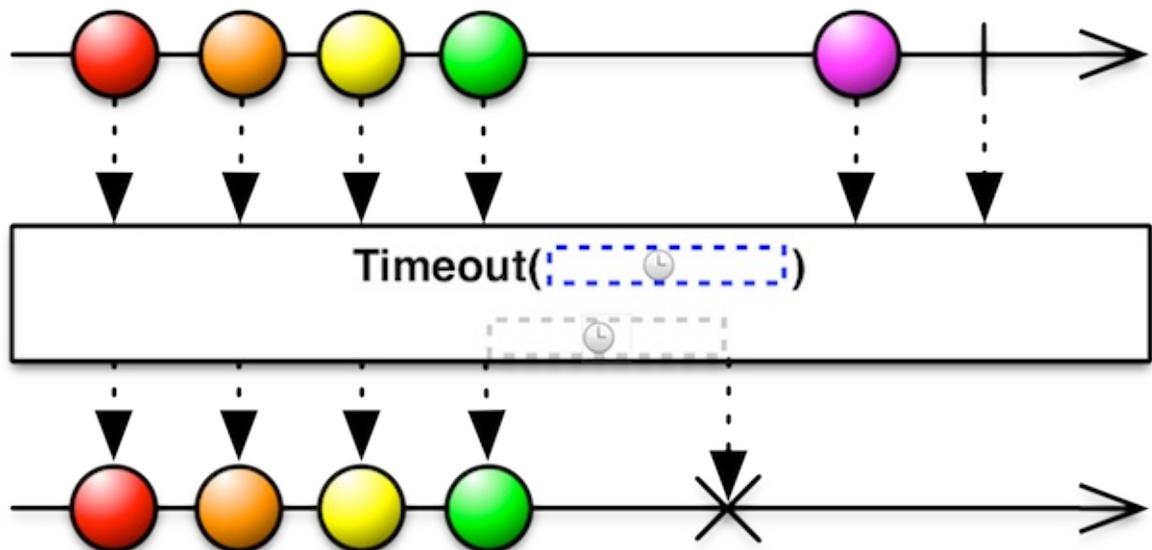
Observable.of(1, 2, 3, 4, 3, 2, 1)
    .takeWhile { $0 < 4 }
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

输出结果：

```
1  
2  
3
```

timeout

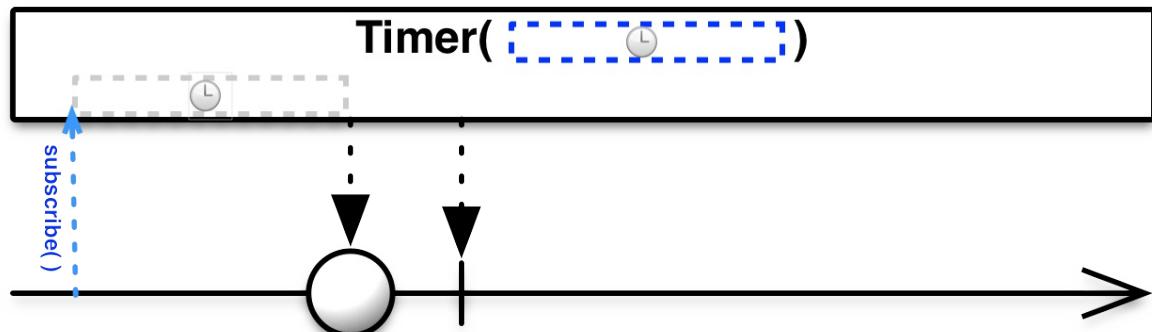
如果源 `Observable` 在规定时间内没有发出任何元素，就产生一个超时的 `error` 事件



如果 `Observable` 在一段时间内没有产生元素，`timeout` 操作符将使它发出一个 `error` 事件。

timer

创建一个 `observable` 在一段延时后，产生唯一的一个元素

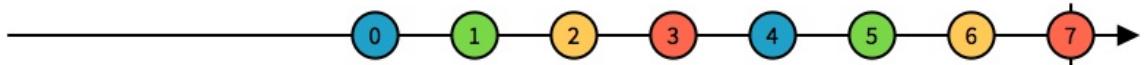


`timer` 操作符将创建一个 `Observable`，它在经过设定的一段时间后，产生唯一的一个元素。

这里存在其他版本的 `timer` 操作符。

timer

`Observable.timer(30, 10)`

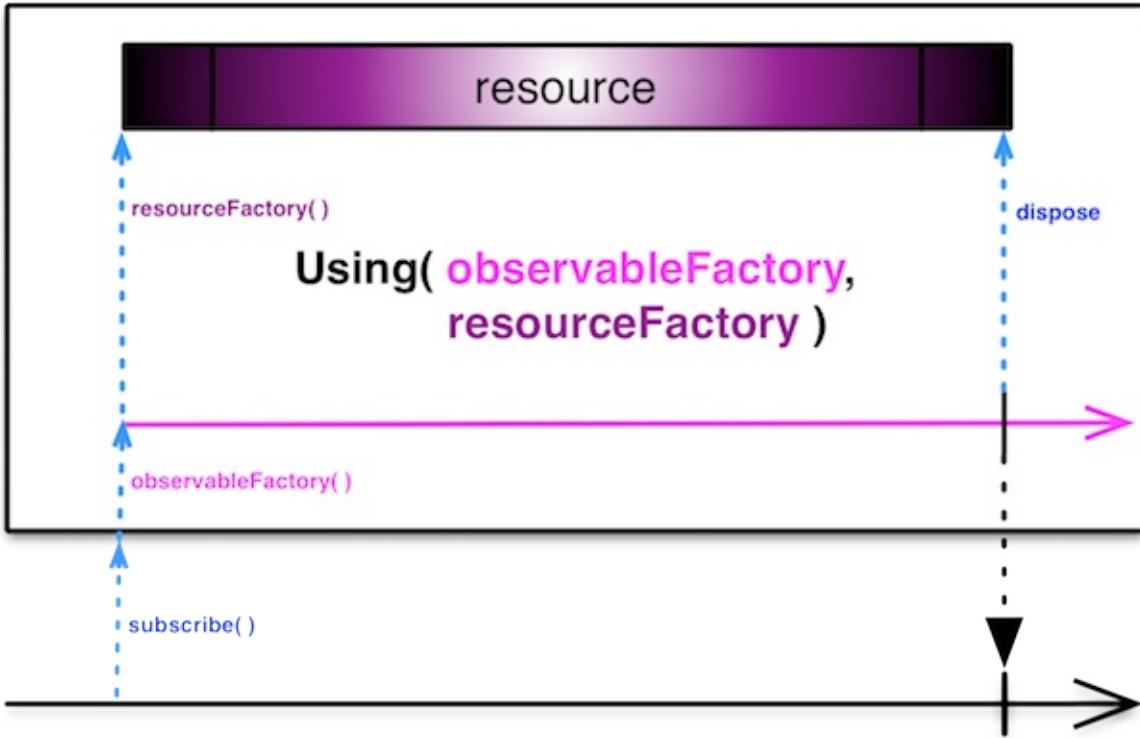


创建一个 `observable` 在一段延时后，每隔一段时间产生一个元素

```
public static func timer(  
    _ dueTime: RxTimeInterval, // 初始延时  
    period: RxTimeInterval?, // 时间间隔  
    scheduler: SchedulerType  
) -> Observable<E>
```

using

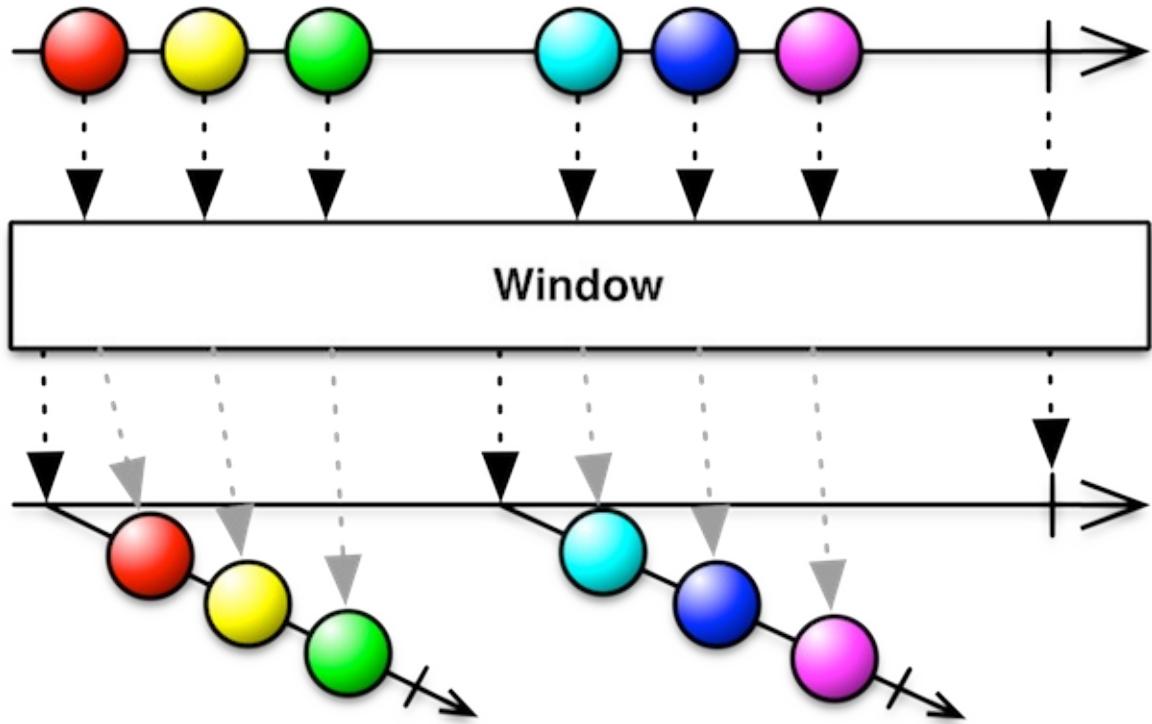
创建一个可被清除的资源，它和 `Observable` 具有相同的寿命



通过使用 **using** 操作符创建 `Observable` 时，同时创建一个可被清除的资源，一旦 `Observable` 终止了，那么这个资源就会被清除掉了。

window

将 `Observable` 分解为多个子 `Observable`，周期性的将子 `Observable` 发出来

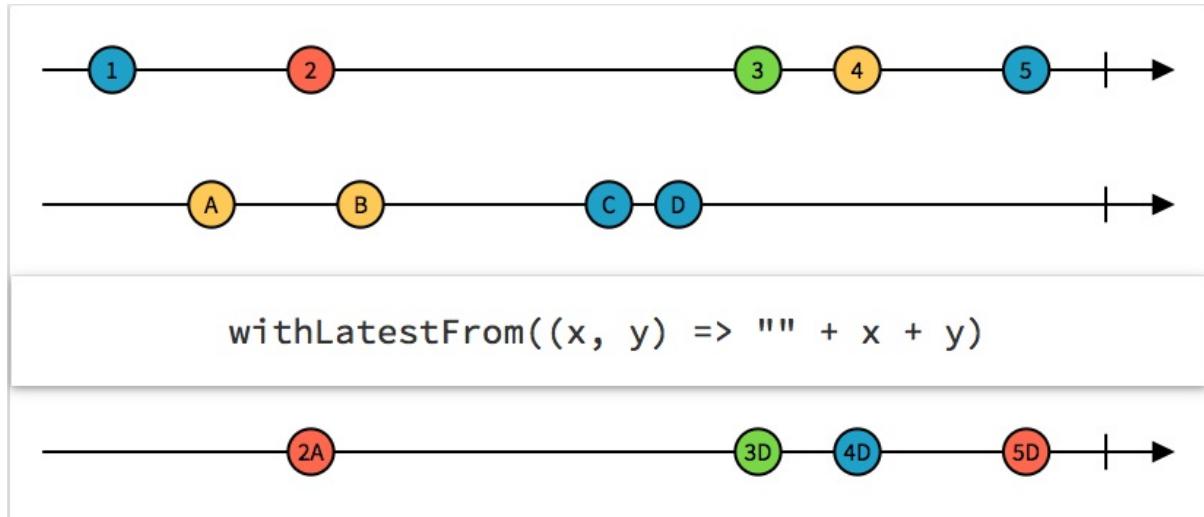


`window` 操作符和 `buffer` 十分相似，`buffer` 周期性的将缓存的元素集合发送出来，而 `window` 周期性的将元素集合以 `Observable` 的形态发送出来。

`buffer` 要等到元素搜集完毕后，才会发出元素序列。而 `window` 可以实时发出元素序列。

withLatestFrom

将两个 `Observables` 最新的元素通过一个函数组合起来，当第一个 `Observable` 发出一个元素，就将组合后的元素发送出来



`withLatestFrom` 操作符将两个 `Observables` 中最新的元素通过一个函数组合起来，然后将这个组合的结果发出来。当第一个 `Observable` 发出一个元素时，就立即取出第二个 `Observable` 中最新的元素，通过一个组合函数将两个最新的元素合并后发送出去。

演示

当第一个 `Observable` 发出一个元素时，就立即取出第二个 `Observable` 中最新的元素，然后把第二个 `Observable` 中最新的元素发送出去。

```

let disposeBag = DisposeBag()
let firstSubject = PublishSubject<String>()
let secondSubject = PublishSubject<String>()

firstSubject
    .withLatestFrom(secondSubject)
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)

firstSubject.onNext("A")
firstSubject.onNext("B")
secondSubject.onNext("1")
secondSubject.onNext("2")
firstSubject.onNext(" ")

```

输出结果：

2

当第一个 `observable` 发出一个元素时，就立即取出第二个 `Observable` 中最新的元素，将第一个 `Observable` 中最新的元素 `first` 和第二个 `observable` 中最新的元素 `second` 组合，然后把组合结果 `first+second` 发送出去。

```
let disposeBag = DisposeBag()
let firstSubject = PublishSubject<String>()
let secondSubject = PublishSubject<String>()

firstSubject
    .withLatestFrom(secondSubject) {
        (first, second) in
        return first + second
    }
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)

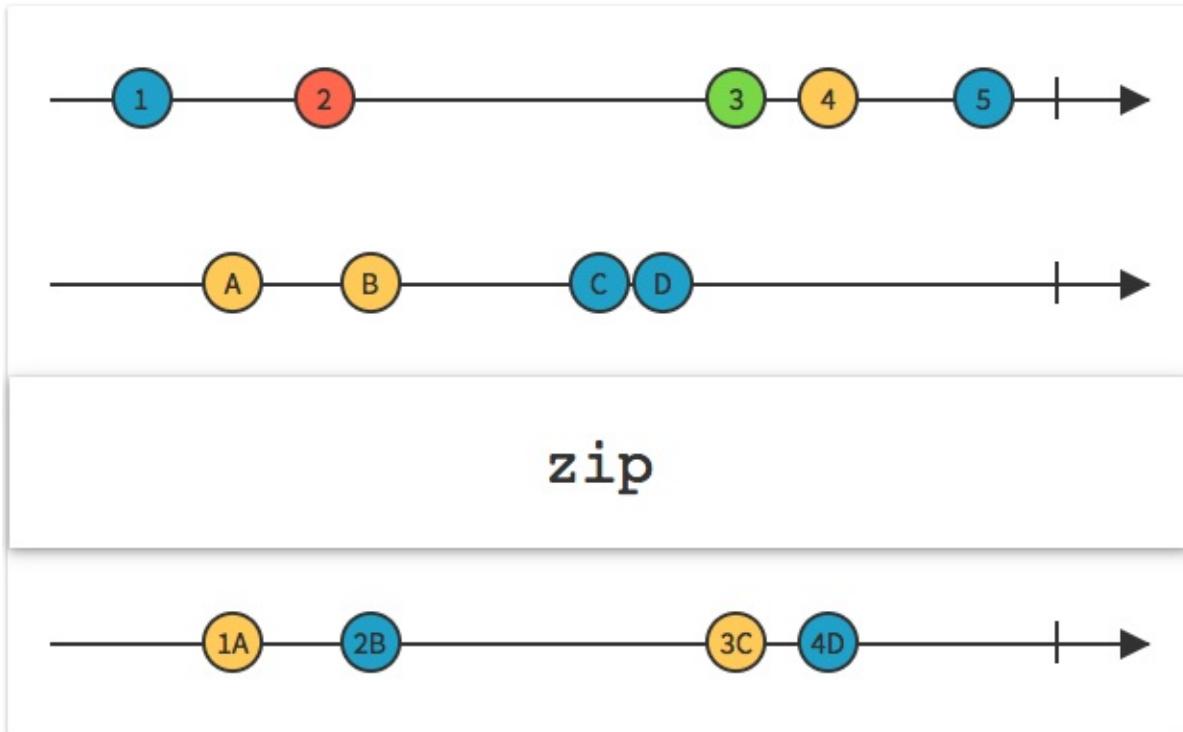
firstSubject.onNext("A")
firstSubject.onNext("B")
secondSubject.onNext("1")
secondSubject.onNext("2")
firstSubject.onNext(" ")
```

输出结果：

2

zip

通过一个函数将多个 `Observables` 的元素组合起来，然后将每一个组合的结果发出来



`zip` 操作符将多个(最多不超过8个) `Observables` 的元素通过一个函数组合起来，然后将这个组合的结果发出来。它会严格的按照序列的索引数进行组合。例如，返回的 `Observable` 的第一个元素，是由每一个源 `Observables` 的第一个元素组合出来的。它的第二个元素，是由每一个源 `Observables` 的第二个元素组合出来的。它的第三个元素，是由每一个源 `Observables` 的第三个元素组合出来的，以此类推。它的元素数量等于源 `Observables` 中元素数量最少的那个。

演示

```
let disposeBag = DisposeBag()
let first = PublishSubject<String>()
let second = PublishSubject<String>()

Observable.zip(first, second) { $0 + $1 }
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)

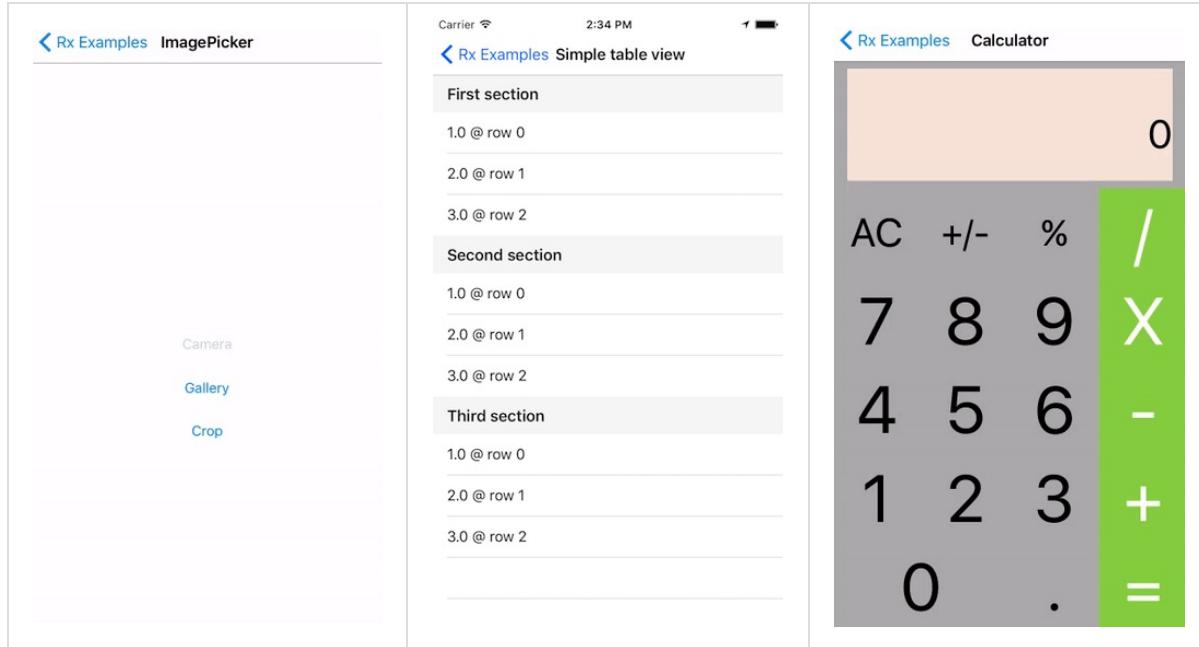
first.onNext("1")
second.onNext("A")
first.onNext("2")
second.onNext("B")
second.onNext("C")
```

```
second.onNext("D")
first.onNext("3")
first.onNext("4")
```

输出结果：

```
1A
2B
3C
4D
```

更多示例



RxExample 中包含许多具有代表性的示例。它们都是很好的学习材料。这里我们取出其中几个示例来展示如何应用 RxSwift :

- [ImagePickerController](#) - 图片选择器
- [TableViewSectionedViewController](#) - 多层级的列表页
- [Calculator](#) - 计算器

有兴趣的同学还可以研究一下 RxExample 中其他的示例。

ImagePicker - 图片选择器

[Rx Examples](#) [ImagePicker](#)

Camera

Gallery

Crop

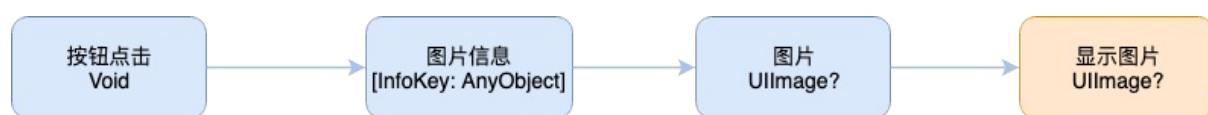
这是一个图片选择器的演示，你可以在这里下载[这个例子](#)。

简介

这个 App 主要有这样几个交互：

- 当用点击相机按钮时，让用户拍一张照片，然后显示出来。
- 当用点击相册按钮时，让用户从相册中选出照片，然后显示出来。
- 当用点击裁剪按钮时，让用户从相册中选出照片编辑，然后显示出来。

整体结构



```

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

    ...

    cameraButton.rx.tap
        .flatMapLatest { [weak self] _ in
            return UIImagePickerController.rx.createWithParent(self) { picker in
                picker.sourceType = .camera
                picker.allowsEditing = false
            }
        .flatMap { $0.rx.didFinishPickingMediaWithInfo }
        .take(1)
    }
    .map { info in
        return info[.originalImage] as? UIImage
    }
    .bind(to: imageView.rx.image)
    .disposed(by: disposeBag)

    ...
}

```

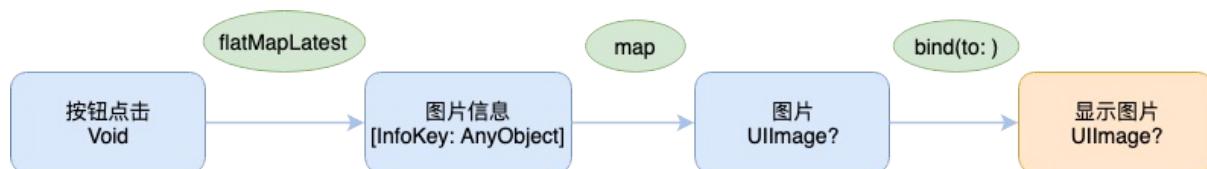
我们忽略一些细节，看一下序列的转换过程：

```

cameraButton.rx.tap
    .flatMapLatest { () -> Observable<[InfoKey: AnyObject]> ... } // 点击 -> 图片信息
    .map { [InfoKey : AnyObject] -> UIImage? ... } // 图片信息 -> 图片
    .bind(to: imageView.rx.image) // 数据绑定
    .disposed(by: disposeBag)

```

最开始的**按钮点击**是一个 `Void` 序列，接着用 `flatMapLatest` 将它异步转化为**图片信息序列** `[String : AnyObject]`，然后用 `map` 同步的从图片信息中取出图片，从而得到了一个**图片序列** `UIImage?`，最后将这个图片序列绑定到 `imageView` 上：



这是相机按钮点击后需要执行的操作。另外两个按钮（相册和裁剪）和它十分相似，只不过传入了不同的参数，通过不同的键取出图片：

```

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

```

```

...
// 相机
cameraButton.rx.tap
    .flatMapLatest { [weak self] _ in
        return UIImagePickerController.rx.createWithParent(self) { picker in
            picker.sourceType = .camera
            picker.allowsEditing = false
        }
        .flatMap { $0.rx.didFinishPickingMediaWithInfo }
        .take(1)
    }
    .map { info in
        return info[UIImagePickerControllerOriginalImage] as? UIImage
    }
    .bind(to: imageView.rx.image)
    .disposed(by: disposeBag)

// 相册
galleryButton.rx.tap
    .flatMapLatest { [weak self] _ in
        return UIImagePickerController.rx.createWithParent(self) { picker in
            picker.sourceType = .photoLibrary
            picker.allowsEditing = false
        }
        .flatMap {
            $0.rx.didFinishPickingMediaWithInfo
        }
        .take(1)
    }
    .map { info in
        return info[.originalImage] as? UIImage
    }
    .bind(to: imageView.rx.image)
    .disposed(by: disposeBag)

// 裁剪
cropButton.rx.tap
    .flatMapLatest { [weak self] _ in
        return UIImagePickerController.rx.createWithParent(self) { picker in
            picker.sourceType = .photoLibrary
            picker.allowsEditing = true
        }
        .flatMap { $0.rx.didFinishPickingMediaWithInfo }
        .take(1)
    }
    .map { info in
        return info[.editedImage] as? UIImage
    }
    .bind(to: imageView.rx.image)
    .disposed(by: disposeBag)

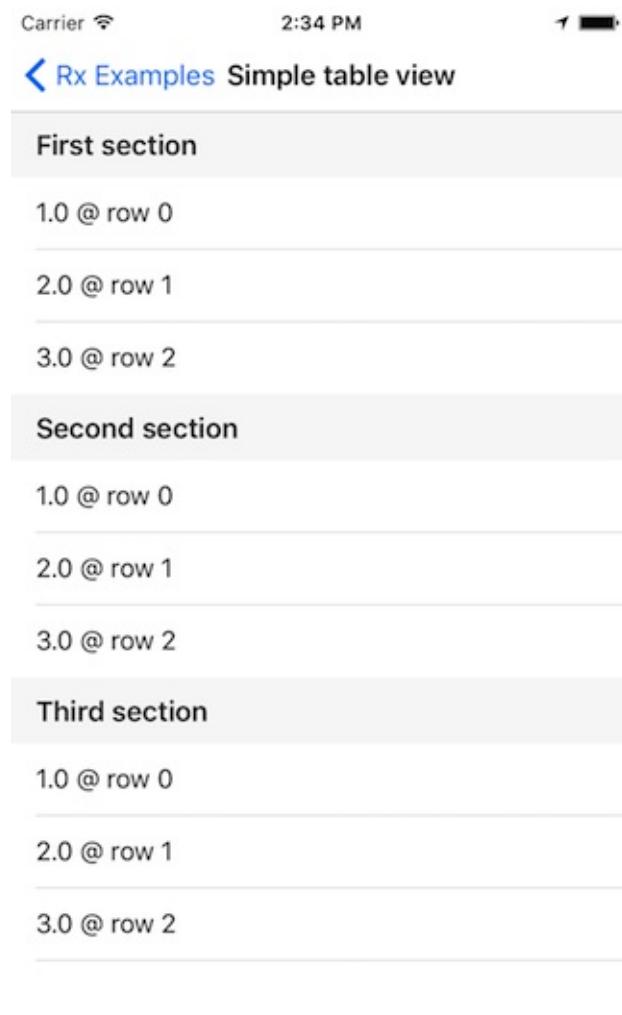
```

```
    }  
    ...
```

参考

- [flatMap](#)
- [flatMapLatest](#)
- [map](#)
- [take](#)

TableViewSectionedViewController - 多层级的列表页



演示如何使用 [RxDataSources](#) 来布局列表页，你可以在这里下载[这个例子](#)。

简介

这是一个多层次列表页，它主要需要完成这些需求：

- 每个 `Section` 显示对应的标题
- 每个 `cell` 显示对应的元素以及行号
- 根据 `cell` 的 `indexPath` 控制行高
- 当 `Cell` 被选中时，显示一个弹框

整体结构



以上这些需求，只需要一页代码就能完成：

```

class SimpleTableViewExampleSectionedViewController
: ViewController
, UITableViewDelegate {
@IBOutlet weak var tableView: UITableView!

let dataSource = RxTableViewSectionedReloadDataSource<SectionModel<String, Double>>()
    configureCell: { (_, tv, indexPath, element) in
        let cell = tv.dequeueReusableCell(withIdentifier: "Cell")!
        cell.textLabel?.text = "\u{element} @ row \u{indexPath.row}"
        return cell
    },
    titleForHeaderInSection: { dataSource, sectionIndex in
        return dataSource[sectionIndex].model
    }
}

override func viewDidLoad() {
    super.viewDidLoad()

    let dataSource = self.dataSource

    let items = Observable.just([
        SectionModel(model: "First section", items: [
            1.0,
            2.0,
            3.0
        ]),
        SectionModel(model: "Second section", items: [
            1.0,
            2.0,
            3.0
        ]),
        SectionModel(model: "Third section", items: [
            1.0,
            2.0,
            3.0
        ])
    ])

    items
        .bind(to: tableView.rx.items(dataSource: dataSource))
}

```

```

        .disposed(by: disposeBag)

    tableView.rx
        .itemSelected
        .map { indexPath in
            return (indexPath, dataSource[indexPath])
        }
        .subscribe(onNext: { pair in
            DefaultWireframe.presentAlert("Tapped `\\(pair.1)` @ `\\(pair.0)`")
        })
        .disposed(by: disposeBag)

    tableView.rx
        .setDelegate(self)
        .disposed(by: disposeBag)
}

// to prevent swipe to delete behavior
func tableView(_ tableView: UITableView, editingStyleForRowAt indexPath: IndexPath) -> UITableViewCellEditingStyle {
    return .none
}

func tableView(_ tableView: UITableView, heightForHeaderInSection section: Int) -> CGFloat {
    return 40
}
}

```

我们首先创建一个 `dataSource: RxTableViewSectionedReloadDataSource<SectionModel<String, Double>>` :

```

let dataSource = RxTableViewSectionedReloadDataSource<SectionModel<String, Double>>(
    configureCell: { (_, tv, indexPath, element) in
        let cell = tv.dequeueReusableCell(withIdentifier: "Cell")!
        cell.textLabel?.text = "\\(element) @ row \\(indexPath.row)"
        return cell
    },
    titleForHeaderInSection: { dataSource, sectionIndex in
        return dataSource[sectionIndex].model
    }
)

```

通过使用这个辅助类型，我们就不用执行数据源代理方法，而只需要提供必要的配置函数就可以布局列表页了。

第一个函数 `configureCell` 是用来配置 `Cell` 的显示，而这里的参数 `element` 就是 `SectionModel<String, Double>` 中的 `Double`。

第二个函数 `titleForHeaderInSection` 是用来配置 `Section` 的标题，而 `dataSource[sectionIndex].model` 就是 `SectionModel<String, Double>` 中的 `String`。

然后为列表页订制一个多层级的数据源 `items: Observable<[SectionModel<String, Double>]>`，用这个数据源来绑定列表页。



这里 `SectionModel<String, Double>` 中的 `String` 是用来显示 `Section` 的标题。而 `Double` 是用来绑定对应的 `Cell`。假如我们的列表页是用来显示通讯录的，并且通讯录通过首字母来分组。那么应该把数据定义为 `SectionModel<String, Person>`，然后用首字母 `String` 来显示 `Section` 标题，用联系人 `Person` 来显示对应的 `Cell`。

由于 `SectionModel<Section, ItemType>` 是一个范型，所以我们可以用它来定义任意类型的 `Section` 以及 `Item`。

最后：

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

    ...

    tableView.rx
        .setDelegate(self)
        .disposed(by: disposeBag)
}

...

func tableView(_ tableView: UITableView, heightForHeaderInSection section: Int) ->
CGFloat {
```

```
    return 40  
}
```

这个是用来控制行高的，`tableView.rx.setDelegate(self)...` 将自己设置成 `tableview` 的代理，通过 `heightForHeaderInSection` 方法提供行高。

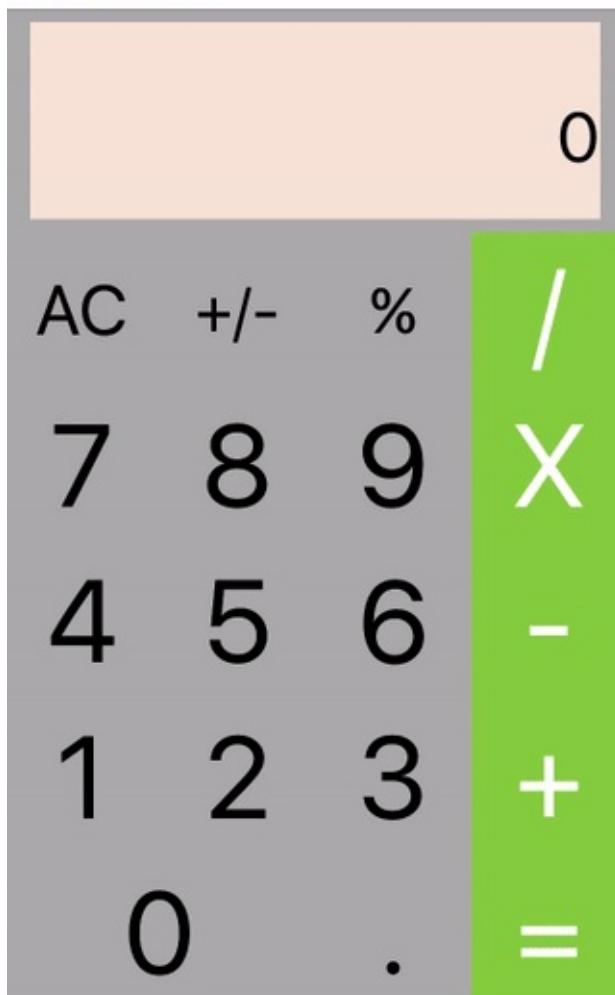
参考

- RxDataSources
- just
- map

Calculator - 计算器

$1 + 2 + 3 = 6$

[Rx Examples](#) [Calculator](#)

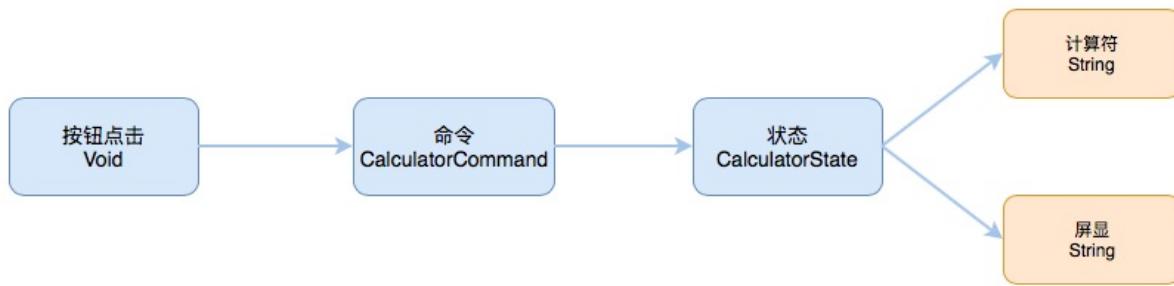


这是一个计算器应用程序，你可以在这里下载[这个例子](#)。

简介

这里的计算器是用响应式编程写的，而且它还用到了 RxFeedback 架构。它比较适合有经验的 RxSwift 使用者学习。接下来我们就来介绍一下这个应用程序是如何实现的。

整体结构



```

class CalculatorViewController: ViewController {

    @IBOutlet weak var lastSignLabel: UILabel!
    @IBOutlet weak var resultLabel: UILabel!

    @IBOutlet weak var allClearButton: UIButton!
    @IBOutlet weak var changeSignButton: UIButton!
    @IBOutlet weak var percentButton: UIButton!

    @IBOutlet weak var divideButton: UIButton!
    @IBOutlet weak var multiplyButton: UIButton!
    @IBOutlet weak var minusButton: UIButton!
    @IBOutlet weak var plusButton: UIButton!
    @IBOutlet weak var equalButton: UIButton!

    @IBOutlet weak var dotButton: UIButton!

    @IBOutlet weak var zeroButton: UIButton!
    @IBOutlet weak var oneButton: UIButton!
    @IBOutlet weak var twoButton: UIButton!
    @IBOutlet weak var threeButton: UIButton!
    @IBOutlet weak var fourButton: UIButton!
    @IBOutlet weak var fiveButton: UIButton!
    @IBOutlet weak var sixButton: UIButton!
    @IBOutlet weak var sevenButton: UIButton!
    @IBOutlet weak var eightButton: UIButton!
    @IBOutlet weak var nineButton: UIButton!

    override func viewDidLoad() {
        let commands: Observable<CalculatorCommand> = Observable.merge([
            allClearButton.rx.tap.map { _ in .clear },

            changeSignButton.rx.tap.map { _ in .changeSign },
            percentButton.rx.tap.map { _ in .percent },

            divideButton.rx.tap.map { _ in .operation(.division) },
            multiplyButton.rx.tap.map { _ in .operation(.multiplication) },
            minusButton.rx.tap.map { _ in .operation(.subtraction) },
            plusButton.rx.tap.map { _ in .operation(.addition) },

            equalButton.rx.tap.map { _ in .equal },
        ])
    }
}
  
```

```

dotButton.rx.tap.map { _ in .addDot },

zeroButton.rx.tap.map { _ in .addNumber("0") },
oneButton.rx.tap.map { _ in .addNumber("1") },
twoButton.rx.tap.map { _ in .addNumber("2") },
threeButton.rx.tap.map { _ in .addNumber("3") },
fourButton.rx.tap.map { _ in .addNumber("4") },
fiveButton.rx.tap.map { _ in .addNumber("5") },
sixButton.rx.tap.map { _ in .addNumber("6") },
sevenButton.rx.tap.map { _ in .addNumber("7") },
eightButton.rx.tap.map { _ in .addNumber("8") },
nineButton.rx.tap.map { _ in .addNumber("9") }

])

let system = Observable.system(
    CalculatorState.initial,
    accumulator: CalculatorState.reduce,
    scheduler: MainScheduler.instance,
    feedback: { _ in commands }
)
.debug("calculator state")
.share(replay: 1)

system.map { $0.screen }
.bind(to: resultLabel.rx.text)
.disposed(by: disposeBag)

system.map { $0.sign }
.bind(to: lastSignLabel.rx.text)
.disposed(by: disposeBag)
}

func formatResult(_ result: String) -> String {
    if result.hasSuffix(".0") {
        return result.substring(to: result.index(result.endIndex, offsetBy: -2))
    }
    } else {
        return result
    }
}
}
}

```

首先合成出一个命令序列，它是通过按钮点击转换过来的：



```

let commands: Observable<CalculatorCommand> = Observable.merge([
  allClearButton.rx.tap.map { _ in .clear },
  changeSignButton.rx.tap.map { _ in .changeSign },
  percentButton.rx.tap.map { _ in .percent },
  divideButton.rx.tap.map { _ in .operation(.division) },
  multiplyButton.rx.tap.map { _ in .operation(.multiplication) },
  minusButton.rx.tap.map { _ in .operation(.subtraction) },
  plusButton.rx.tap.map { _ in .operation(.addition) },
  equalButton.rx.tap.map { _ in .equal },
  dotButton.rx.tap.map { _ in .addDot },
  zeroButton.rx.tap.map { _ in .addNumber("0") },
  oneButton.rx.tap.map { _ in .addNumber("1") },
  twoButton.rx.tap.map { _ in .addNumber("2") },
  threeButton.rx.tap.map { _ in .addNumber("3") },
  fourButton.rx.tap.map { _ in .addNumber("4") },
  fiveButton.rx.tap.map { _ in .addNumber("5") },
  sixButton.rx.tap.map { _ in .addNumber("6") },
  sevenButton.rx.tap.map { _ in .addNumber("7") },
  eightButton.rx.tap.map { _ in .addNumber("8") },
  nineButton.rx.tap.map { _ in .addNumber("9") }
])

```

通过使用 `map` 方法将按钮点击事件转换为对应的命令。如：将 `allClearButton` 点击事件转换为清除命令，将 `plusButton` 点击事件转换为相加命令，将 `oneButton` 点击事件转换为添加数字1命令。最后使用 `merge` 操作符将这些命令合并。于是就得到了我们所需要的命令序列。

几乎每个页面都是有状态的。我们通过命令序列来对状态进行修改，然后产生一个新的状态。例如，刚进页面后，点击了按钮 1。那么初始状态为 0，在执行添加数字1命令后，状态就更新为 1。通过这种变换方式，就可以生成一个状态序列：

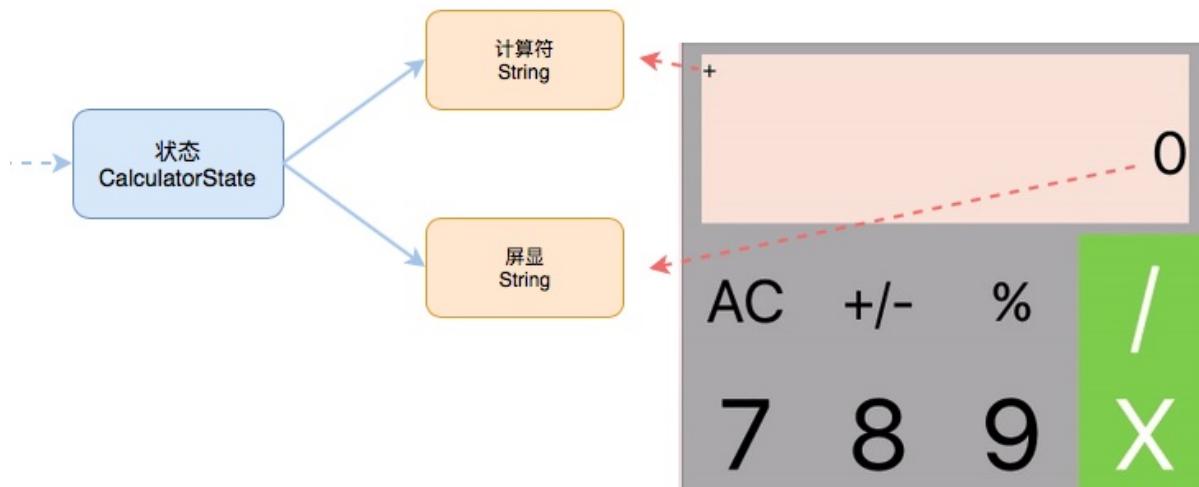


```

let system = Observable.system(
  CalculatorState.initial,
  accumulator: CalculatorState.reduce,
  scheduler: MainScheduler.instance,
  feedback: { _ in commands }
)
.debug("calculator state")
.share(replay: 1)

```

由命令序列触发，对页面状态进行更新，在用更新后的状态组成一个序列。这就是我们所需要的状态序列。接下来我们用这个状态序列来控制页面显示：



```
system.map { $0.screen }
    .bind(to: resultLabel.rx.text)
    .disposed(by: disposeBag)

system.map { $0.sign }
    .bind(to: lastSignLabel.rx.text)
    .disposed(by: disposeBag)
```

用 `state.screen` 来控制 `resultLabel` 的显示内容。用 `state.sign` 来控制 `lastSignLabel` 的显示内容。

Calculator

控制器主要负责数据绑定，而整个计算器的大脑在 **Calculator.swift** 文件内。

State:

这个页面主要有三种状态：

```
enum CalculatorState {
    case oneOperand(screen: String)
    case oneOperandAndOperator(operand: Double, operator: Operator)
    case twoOperandsAndOperator(operand: Double, operator: Operator, screen: String)
}
```

- `oneOperand` 一个操作数，例如：进入页面后，输入 `1` 时的状态
- `oneOperandAndOperator` 一个操作数和一个运算符，例如：进入页面后，输入 `1 +` 时的状态
- `twoOperandsAndOperator` 两个操作数和一个运算符，例如：进入页面后，输入 `1 + 2` 时的状

态

Command:

这个计算器提供七种命令：

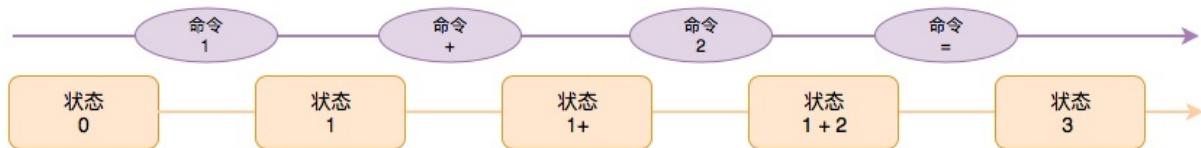
```
enum Operator {
    case addition
    case subtraction
    case multiplication
    case division
}

enum CalculatorCommand {
    case clear
    case changeSign
    case percent
    case operation(Operator)
    case equal
    case addNumber(Character)
    case addDot
}
```

- clear 清除，重置
- changeSign 改变正负号
- percent 百分比
- operation 四则运算
- equal 等于
- addNumber 输入数字
- addDot 输入“.”

reduce:

当命令产生时，将它应用到当前状态上，然后生成新的状态：



```
extension CalculatorState {
    static func reduce(state: CalculatorState, _ x: CalculatorCommand) -> CalculatorState {
        switch x {
        case .clear:
            return CalculatorState.initial
        case .addNumber(let c):
            return state.mapScreen { $0 == "0" ? String(c) : $0 + String(c) }
        case .addDot:
            return state.mapScreen { $0.range(of: ".") == nil ? $0 + "." : $0 }
        }
    }
}
```

```

        case .changeSign:
            return state.mapScreen { "\(-(Double($0) ?? 0.0))" }
        case .percent:
            return state.mapScreen { "\((Double($0) ?? 0.0) / 100.0)" }
        case .operation(let o):
            switch state {
            case let .oneOperand(screen):
                return .oneOperandAndOperator(operand: screen.doubleValue, operator: o)
            case let .oneOperandAndOperator(operand, _):
                return .oneOperandAndOperator(operand: operand, operator: o)
            case let .twoOperandsAndOperator(operand, oldOperator, screen):
                return .twoOperandsAndOperator(operand: oldOperator.perform(operand, screen.doubleValue), operator: o, screen: "0")
            }
        case .equal:
            switch state {
            case let .twoOperandsAndOperator(operand, operat, screen):
                let result = operat.perform(operand, screen.doubleValue)
                return .oneOperand(screen: String(result))
            default:
                return state
            }
        }
    }
}

```

- clear 重置当前状态
- addNumber, addDot, changeSign, percent 只需要更改屏显即可
- operation 需要根据当前状态来确定如何变化状态。
 - 如果只有一个操作数，就添加操作符。
 - 如果有一个操作数和操作符，就替换操作符。
 - 如果有两个操作数和一个操作符，将他们的计算结果作为操作数保留，然后加入新的操作符，以及一个操作数 0。
- equal 如果当前有两个操作数和一个操作符，将他们的计算结果作为操作数保留。否则什么都不做。

剩下的都是一些辅助代码，接下来我们再来看下全部代码：

ViewController:

```

class CalculatorViewController: ViewController {

    @IBOutlet weak var lastSignLabel: UILabel!
    @IBOutlet weak var resultLabel: UILabel!

    @IBOutlet weak var allClearButton: UIButton!
    @IBOutlet weak var changeSignButton: UIButton!
    @IBOutlet weak var percentButton: UIButton!
}

```

```

@IBOutlet weak var divideButton: UIButton!
@IBOutlet weak var multiplyButton: UIButton!
@IBOutlet weak var minusButton: UIButton!
@IBOutlet weak var plusButton: UIButton!
@IBOutlet weak var equalButton: UIButton!

@IBOutlet weak var dotButton: UIButton!

@IBOutlet weak var zeroButton: UIButton!
@IBOutlet weak var oneButton: UIButton!
@IBOutlet weak var twoButton: UIButton!
@IBOutlet weak var threeButton: UIButton!
@IBOutlet weak var fourButton: UIButton!
@IBOutlet weak var fiveButton: UIButton!
@IBOutlet weak var sixButton: UIButton!
@IBOutlet weak var sevenButton: UIButton!
@IBOutlet weak var eightButton: UIButton!
@IBOutlet weak var nineButton: UIButton!

override func viewDidLoad() {
    let commands: Observable<CalculatorCommand> = Observable.merge([
        allClearButton.rx.tap.map { _ in .clear },
        changeSignButton.rx.tap.map { _ in .changeSign },
        percentButton.rx.tap.map { _ in .percent },
        divideButton.rx.tap.map { _ in .operation(.division) },
        multiplyButton.rx.tap.map { _ in .operation(.multiplication) },
        minusButton.rx.tap.map { _ in .operation(.subtraction) },
        plusButton.rx.tap.map { _ in .operation(.addition) },
        equalButton.rx.tap.map { _ in .equal },
        dotButton.rx.tap.map { _ in .addDot },
        zeroButton.rx.tap.map { _ in .addNumber("0") },
        oneButton.rx.tap.map { _ in .addNumber("1") },
        twoButton.rx.tap.map { _ in .addNumber("2") },
        threeButton.rx.tap.map { _ in .addNumber("3") },
        fourButton.rx.tap.map { _ in .addNumber("4") },
        fiveButton.rx.tap.map { _ in .addNumber("5") },
        sixButton.rx.tap.map { _ in .addNumber("6") },
        sevenButton.rx.tap.map { _ in .addNumber("7") },
        eightButton.rx.tap.map { _ in .addNumber("8") },
        nineButton.rx.tap.map { _ in .addNumber("9") }
    ])
}

let system = Observable.system(
    CalculatorState.initial,
    accumulator: CalculatorState.reduce,

```

```

        scheduler: MainScheduler.instance,
        feedback: { _ in commands }
    )
    .debug("calculator state")
    .share(replay: 1)

    system.map { $0.screen }
        .bind(to: resultLabel.rx.text)
        .disposed(by: disposeBag)

    system.map { $0.sign }
        .bind(to: lastSignLabel.rx.text)
        .disposed(by: disposeBag)
}

func formatResult(_ result: String) -> String {
    if result.hasSuffix(".0") {
        return result.substring(to: result.index(result.endIndex, offsetBy: -2))
    } else {
        return result
    }
}
}
}

```

Calculator:

```

enum Operator {
    case addition
    case subtraction
    case multiplication
    case division
}

enum CalculatorCommand {
    case clear
    case changeSign
    case percent
    case operation(Operator)
    case equal
    case addNumber(Character)
    case addDot
}

enum CalculatorState {
    case oneOperand(screen: String)
    case oneOperandAndOperator(operand: Double, operator: Operator)
    case twoOperandsAndOperator(operand: Double, operator: Operator, screen: String)
}
}

```

```

extension CalculatorState {
    static let initial = CalculatorState.oneOperand(screen: "0")

    func mapScreen(transform: (String) -> String) -> CalculatorState {
        switch self {
        case let .oneOperand(screen):
            return .oneOperand(screen: transform(screen))
        case let .oneOperandAndOperator(operand, operat):
            return .twoOperandsAndOperator(operand: operand, operator: operat, screen: transform("0"))
        case let .twoOperandsAndOperator(operand, operat, screen):
            return .twoOperandsAndOperator(operand: operand, operator: operat, screen: transform(screen))
        }
    }

    var screen: String {
        switch self {
        case let .oneOperand(screen):
            return screen
        case .oneOperandAndOperator:
            return "0"
        case let .twoOperandsAndOperator(_, _, screen):
            return screen
        }
    }

    var sign: String {
        switch self {
        case .oneOperand:
            return ""
        case let .oneOperandAndOperator(_, o):
            return o.sign
        case let .twoOperandsAndOperator(_, o, _):
            return o.sign
        }
    }
}

extension CalculatorState {
    static func reduce(state: CalculatorState, _ x: CalculatorCommand) -> CalculatorState {
        switch x {
        case .clear:
            return CalculatorState.initial
        case .addNumber(let c):
            return state.mapScreen { $0 == "0" ? String(c) : $0 + String(c) }
        case .addDot:
            return state.mapScreen { $0.range(of: ".") == nil ? $0 + "." : $0 }
        }
    }
}

```

```

        case .changeSign:
            return state.mapScreen { "\(-(Double($0) ?? 0.0))" }
        case .percent:
            return state.mapScreen { "\((Double($0) ?? 0.0) / 100.0)" }
        case .operation(let o):
            switch state {
                case let .oneOperand(screen):
                    return .oneOperandAndOperator(operand: screen.doubleValue, operator: o)
                case let .oneOperandAndOperator(operand, _):
                    return .oneOperandAndOperator(operand: operand, operator: o)
                case let .twoOperandsAndOperator(operand, oldOperator, screen):
                    return .twoOperandsAndOperator(operand: oldOperator.perform(operand, screen.doubleValue), operator: o, screen: "0")
            }
        case .equal:
            switch state {
                case let .twoOperandsAndOperator(operand, operat, screen):
                    let result = operat.perform(operand, screen.doubleValue)
                    return .oneOperand(screen: String(result))
                default:
                    return state
            }
        }
    }

extension Operator {
    var sign: String {
        switch self {
            case .addition:      return "+"
            case .subtraction:   return "-"
            case .multiplication: return "x"
            case .division:      return "/"
        }
    }

    var perform: (Double, Double) -> Double {
        switch self {
            case .addition:      return (+)
            case .subtraction:   return (-)
            case .multiplication: return (*)
            case .division:      return (/)
        }
    }
}

private extension String {
    var doubleValue: Double {
        guard let double = Double(self) else {
            return Double.infinity
        }
    }
}

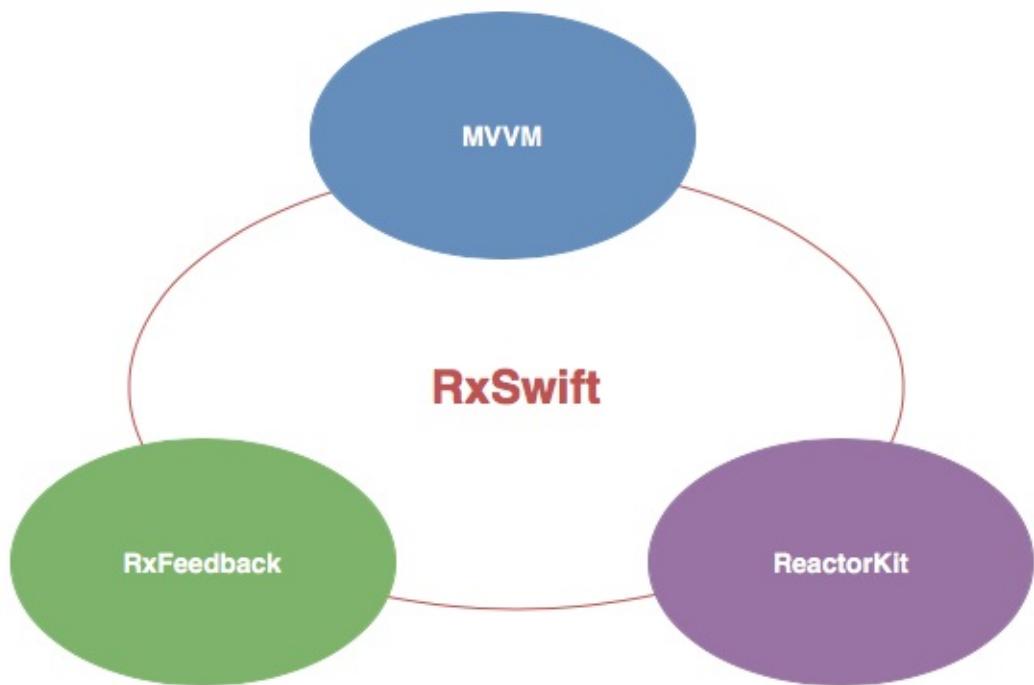
```

```
        }
    return double
}
}
```

参考

- RxFeedback
- merge
- map
- scan

RxSwift 常用架构



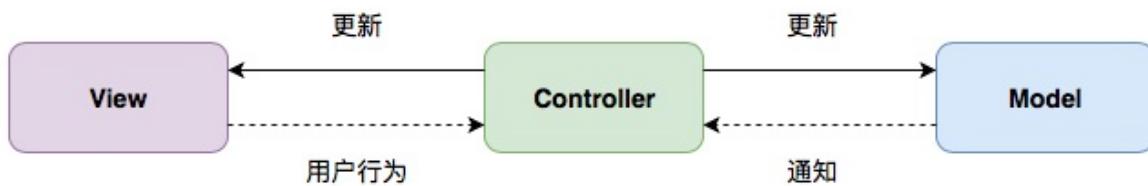
RxSwift 是一个响应式编程的基础框架，它并不会强制要求你使用某种架构。它和多个应用程序架构完美适配，这一章将介绍几个常用的架构：

- **MVVM** - 当今非常流行的 MVVM 设计模式
- **RxFEEDBACK** - 由 RxSwift 创始人（Krunoslav Zaher）提供的一个反馈循环架构
- **ReactorKit** - 结合了 Flux 和响应式编程的架构

MVVM

MVVM 是 **Model-View-ViewModel** 的简写。如果你已经对 **MVC** 非常熟悉了，那么上手 **MVVM** 也是非常容易的。

MVC

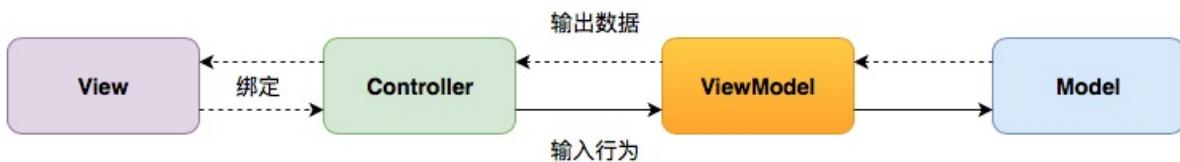


MVC 是 **Model-View-Controller** 的简写。**MVC** 主要有三层：

- **Model** 数据层，读写数据，保存 App 状态
- **View** 页面层，和用户交互，向用户显示页面，反馈用户行为
- **ViewController** 逻辑层，更新数据，或者页面，处理业务逻辑

MVC 可以帮助你很好的将数据，页面，逻辑的代码分离开来。使得每一层相对独立。这样你就能够将一些可复用的功能抽离出来，化繁为简。只不过，一旦 App 的交互变复杂，你就会发现 **ViewController** 将变得十分臃肿。大量代码被添加到控制器中，使得控制器负担过重。此时，你就需要想办法将控制器里面的代码进一步地分离出来，对 APP 进行重新分层。而 **MVVM** 就是一种进阶的分层方案。

MVVM



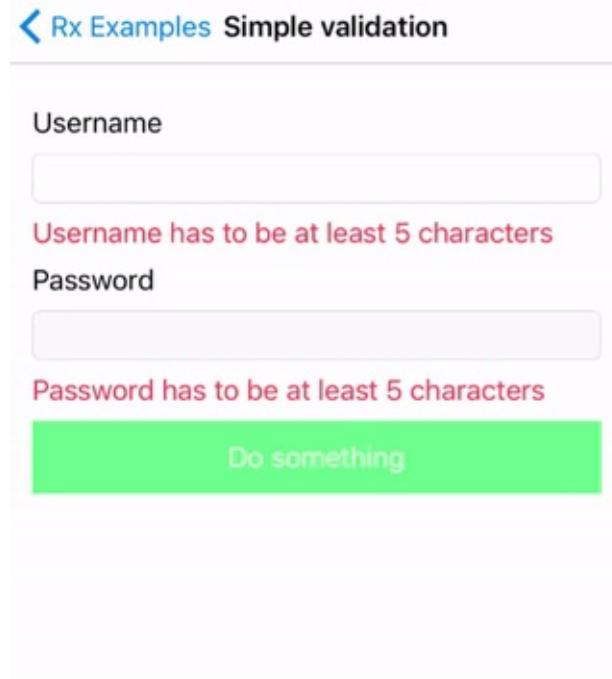
MVVM 和 **MVC** 十分相识。只不过他的分层更加详细：

- **Model** 数据层，读写数据，保存 App 状态
- **View** 页面层，提供用户输入行为，并且显示输出状态
- **ViewModel** 逻辑层，它将用户输入行为，转换成输出状态
- **ViewController** 主要负责数据绑定

没错，**ViewModel** 现在是逻辑层，而控制器只需要负责数据绑定。如此一来控制器的负担就减轻了许多。并且 **ViewModel** 与控制器以及页面相独立。那么，你就可以跨平台使用它。你也可以很容易地测试它。

示例

这里我们将用 **MVVM** 来重构输入验证。



重构前：

```
class SimpleValidationViewController : ViewController {

    ...

    override func viewDidLoad() {
        super.viewDidLoad()

        ...

        let usernameValid = usernameOutlet.rx.text.orEmpty
            .map { $0.characters.count >= minimalUsernameLength }
            .share(replay: 1)

        let passwordValid = passwordOutlet.rx.text.orEmpty
            .map { $0.characters.count >= minimalPasswordLength }
            .share(replay: 1)

        let everythingValid = Observable.combineLatest(
            usernameValid,
            passwordValid
        ) { $0 && $1 }
            .share(replay: 1)
    }
}
```

```

usernameValid
    .bind(to: passwordOutlet.rx.isEnabled)
    .disposed(by: disposeBag)

usernameValid
    .bind(to: usernameValidOutlet.rx.isHidden)
    .disposed(by: disposeBag)

passwordValid
    .bind(to: passwordValidOutlet.rx.isHidden)
    .disposed(by: disposeBag)

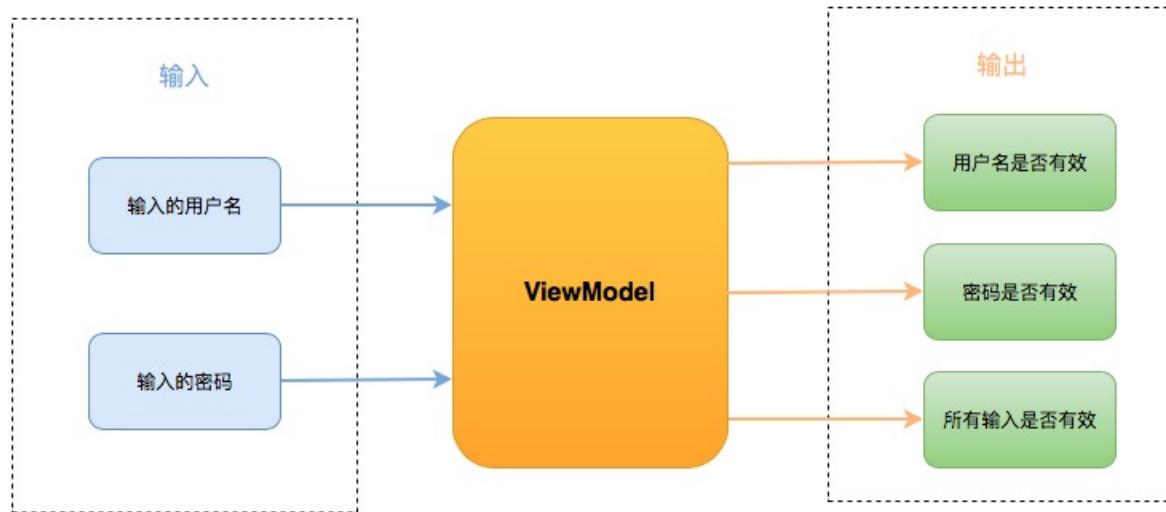
everythingValid
    .bind(to: doSomethingOutlet.rx.isEnabled)
    .disposed(by: disposeBag)

doSomethingOutlet.rx.tap
    .subscribe(onNext: { [weak self] in self?.showAlert() })
    .disposed(by: disposeBag)
}

...
}

```

ViewModel



ViewModel 将用户输入行为，转换成输出的状态：

```

class SimpleValidationViewModel {
    // 输出
}

```

```

let usernameValid: Observable<Bool>
let passwordValid: Observable<Bool>
let everythingValid: Observable<Bool>

// 输入 -> 输出
init(
    username: Observable<String>,
    password: Observable<String>
) {

    usernameValid = username
        .map { $0.characters.count >= minimalUsernameLength }
        .share(replay: 1)

    passwordValid = password
        .map { $0.characters.count >= minimalPasswordLength }
        .share(replay: 1)

    everythingValid = Observable.combineLatest(usernameValid, passwordValid) { $0 && $1 }
        .share(replay: 1)
}

}

```

输入：

- `username` 输入的用户名
- `password` 输入的密码

输出：

- `usernameValid` 用户名是否有效
- `passwordValid` 密码是否有效
- `everythingValid` 所有输入是否有效

在 `init` 方法内部，将输入转换为输出。

ViewController

ViewController 主要负责数据绑定：

```

class SimpleValidationViewController : ViewController {

    ...

    private var viewModel: SimpleValidationViewModel!

```

```

override func viewDidLoad() {
    super.viewDidLoad()

    ...

    viewModel = SimpleValidationViewModel(
        username: usernameOutlet.rx.text.orEmpty.asObservable(),
        password: passwordOutlet.rx.text.orEmpty.asObservable()
    )

    viewModel.usernameValid
        .bind(to: passwordOutlet.rx.isEnabled)
        .disposed(by: disposeBag)

    viewModel.usernameValid
        .bind(to: usernameValidOutlet.rx.isHidden)
        .disposed(by: disposeBag)

    viewModel.passwordValid
        .bind(to: passwordValidOutlet.rx.isHidden)
        .disposed(by: disposeBag)

    viewModel.everythingValid
        .bind(to: doSomethingOutlet.rx.isEnabled)
        .disposed(by: disposeBag)

    doSomethingOutlet.rx.tap
        .subscribe(onNext: { [weak self] in self?.showAlert() })
        .disposed(by: disposeBag)
}

...
}

```

输入：

- `username` 将输入的用户名传入 **ViewModel**
- `password` 将输入的密码传入 **ViewModel**

输出：

- `usernameValid` 用用户名是否有效，来控制提示语是否隐藏，密码输入框是否可用
- `passwordValid` 用密码是否有效，来控制提示语是否隐藏
- `everythingValid` 用两者是否同时有效，来控制按钮是否可点击

当 App 的交互变复杂时，你仍然可以保持控制器结构清晰。这样可以大大的提升代码可读性。将来代码维护起来也就会容易许多了。

示例

下一节将用 [Github Signup](#) 来演示如何使用 **MVVM**。

注意△：这里介绍的 **MVVM** 并不是严格意义上的 **MVVM**。但我们通常都管它叫 **MVVM**，而且它配合 **RxSwift** 使用起来非常方便。如需了解什么是严格意义上的 **MVVM**，请参考微软的 [The MVVM Pattern](#)。

Github Signup

[Rx Examples GitHub Signup](#)

Username
Password
Password Repeat

Sign up

Proving that observable sequences have wanted properties (UIThread, errors handled, sharing of side effects) is done manually. (but has some performance gain that shouldn't be noticeable in practice)

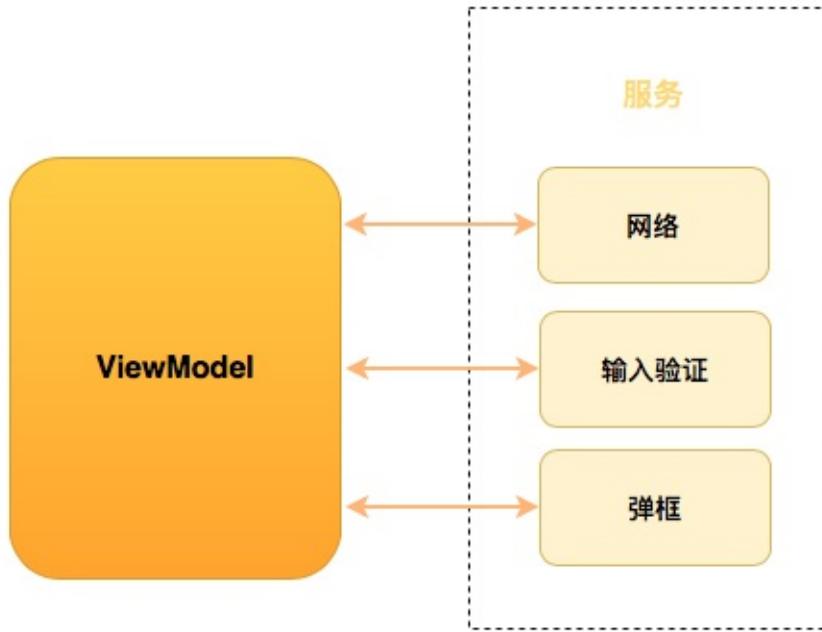
这是一个模拟用户注册的程序，你可以在这里[下载这个例子](#)。

简介

这个 App 主要有这样几个交互：

- 当用户输入用户名时，验证用户名是否有效，是否已被占用，将验证结果显示出来。
 - 当用户输入密码时，验证密码是否有效，将验证结果显示出来。
 - 当用户输入重复密码时，验证重复密码是否相同，将验证结果显示出来。
 - 当所有验证都有效时，注册按钮才可点击。
 - 当点击注册按钮后发起注册请求（模拟），然后将结果显示出来。
-

Service



```

// GitHub 网络服务
protocol GitHubAPI {
    func usernameAvailable(_ username: String) -> Observable<Bool>
    func signup(_ username: String, password: String) -> Observable<Bool>
}

// 输入验证服务
protocol GitHubValidationService {
    func validateUsername(_ username: String) -> Observable<ValidationResult>
    func validatePassword(_ password: String) -> ValidationResult
    func validateRepeatedPassword(_ password: String, repeatedPassword: String) -> ValidationResult
}

// 弹框服务
protocol Wireframe {
    func open(url: URL)
    func promptFor<Action: CustomStringConvertible>(_ message: String, cancelAction: Action, actions: [Action]) -> Observable<Action>
}

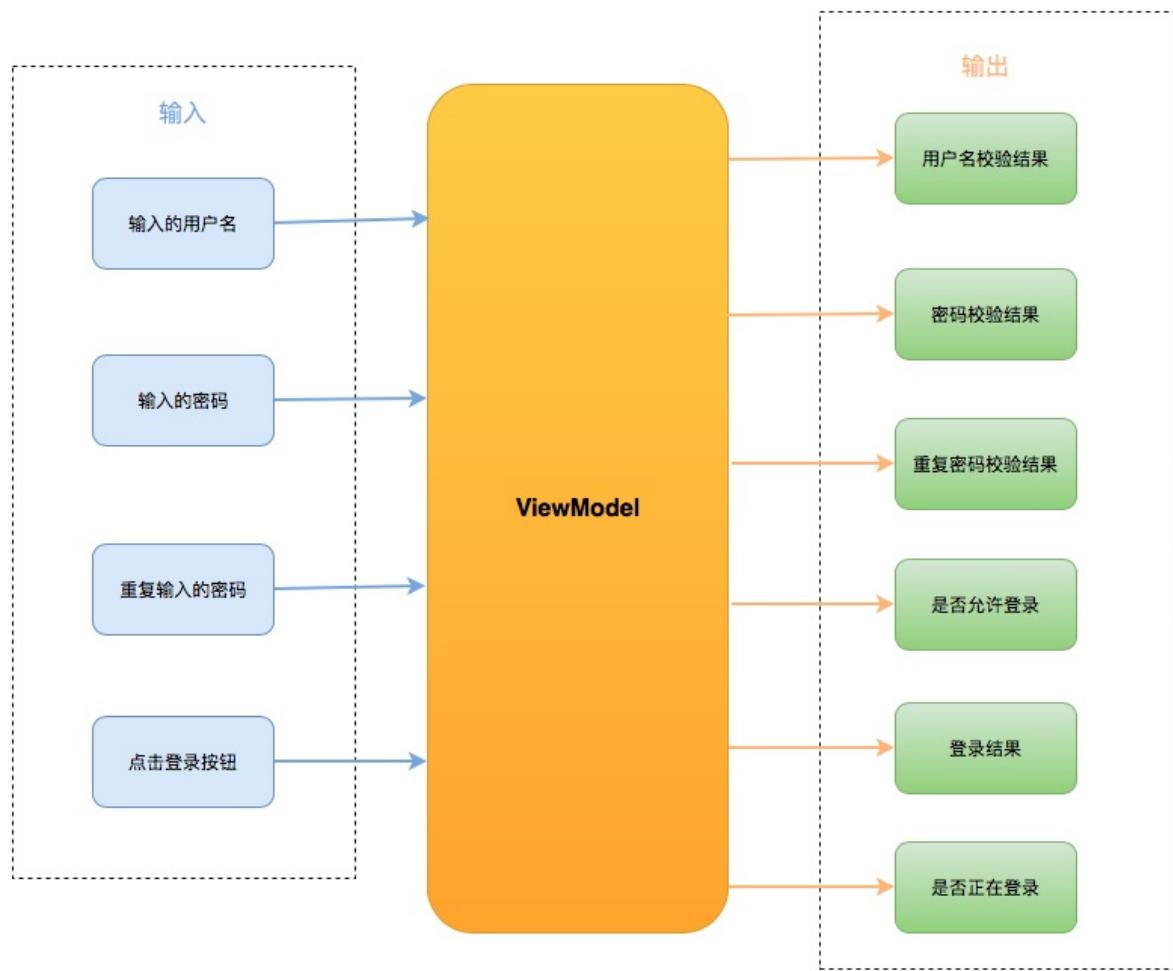
```

这里需要集成三个服务：

- **GitHubAPI** 提供 GitHub 网络服务
- **GitHubValidationService** 提供输入验证服务
- **Wireframe** 提供弹框服务

这个例子目前只提供了这三个服务，实际上这一层还可以包含其他的一些服务，例如：数据库，定位，蓝牙...

ViewModel



ViewModel 需要集成这些服务，并且将用户输入，转换为状态输出：

```

class GithubSignupViewModel1 {

    // 输出
    let validatedUsername: Observable<ValidationResult>
    let validatedPassword: Observable<ValidationResult>
    let validatedPasswordRepeated: Observable<ValidationResult>
    let signupEnabled: Observable<Bool>
    let signedIn: Observable<Bool>
    let signingIn: Observable<Bool>

    // 输入 -> 输出
    init(input: (           // 输入
        username: Observable<String>,
        password: Observable<String>,
        repeatedPassword: Observable<String>,
        loginTaps: Observable<Void>
    ),
    dependency: (          // 服务

```

```

        API: GitHubAPI,
        validationService: GitHubValidationService,
        wireframe: Wireframe
    )
)
{
    ...
    validatedUsername = ...
    validatedPassword = ...
    validatedPasswordRepeated = ...
    ...
    self.signInningIn = ...
    ...
    signedIn = ...
    signupEnabled = ...
}
}

```

集成服务：

- **API** GitHub 网络服务
- **validationService** 输入验证服务
- **wireframe** 弹框服务

输入：

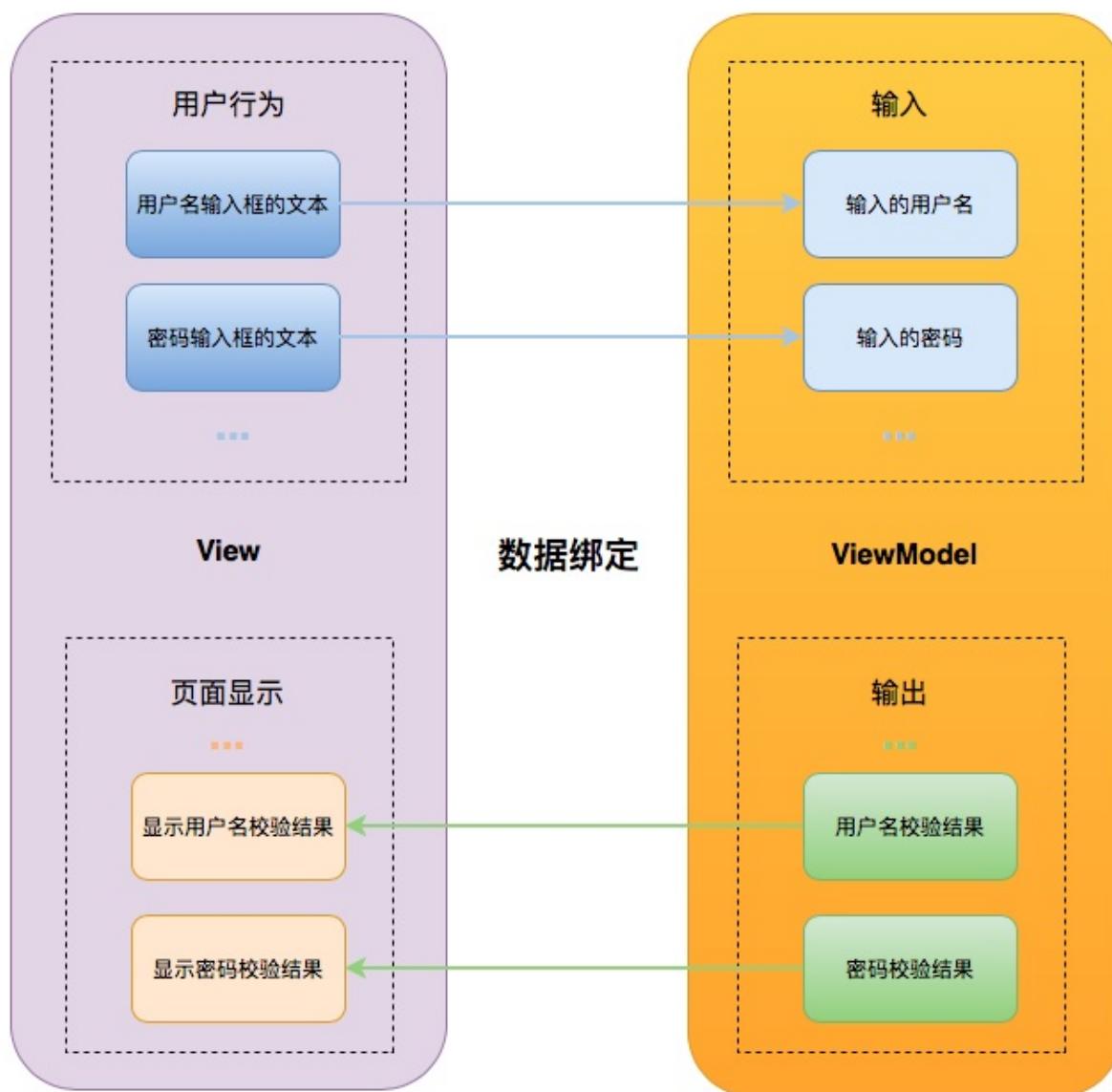
- **username** 输入的用户名
- **password** 输入的密码
- **repeatedPassword** 重复输入的密码
- **loginTaps** 点击登录按钮

输出：

- **validatedUsername** 用户名校验结果
- **validatedPassword** 密码校验结果
- **validatedPasswordRepeated** 重复密码校验结果
- **signupEnabled** 是否允许登录
- **signedIn** 登录结果
- **signingIn** 是否正在登录

在 `init` 方法内部，将输入转换为输出。

ViewController



ViewController 主要负责数据绑定：

```

...
class GitHubSignupViewController : ViewController {
    @IBOutlet weak var usernameOutlet: UITextField!
    @IBOutlet weak var usernameValidationOutlet: UILabel!

    @IBOutlet weak var passwordOutlet: UITextField!
    @IBOutlet weak var passwordValidationOutlet: UILabel!

    @IBOutlet weak var repeatedPasswordOutlet: UITextField!
    @IBOutlet weak var repeatedPasswordValidationOutlet: UILabel!

    @IBOutlet weak var signupOutlet: UIButton!
    @IBOutlet weak var signingUpOulet: UIActivityIndicatorView!
}

```

```

override func viewDidLoad() {
    super.viewDidLoad()

    let viewModel = GithubSignupViewModel1(
        input: (
            username: usernameOutlet.rx.text.orEmpty.asObservable(),
            password: passwordOutlet.rx.text.orEmpty.asObservable(),
            repeatedPassword: repeatedPasswordOutlet.rx.text.orEmpty.asObservable(),
            loginTaps: signupOutlet.rx.tap.asObservable()
        ),
        dependency: (
            API: GitHubDefaultAPI.sharedAPI,
            validationService: GitHubDefaultValidationService.sharedValidationService,
            wireframe: DefaultWireframe.shared
        )
    )

    // bind results to {
    viewModel.signupEnabled
        .subscribe(onNext: { [weak self] valid in
            self?.signupOutlet.isEnabled = valid
            self?.signupOutlet.alpha = valid ? 1.0 : 0.5
        })
        .disposed(by: disposeBag)

    viewModel.validatedUsername
        .bind(to: usernameValidationOutlet.rx.validationResult)
        .disposed(by: disposeBag)

    viewModel.validatedPassword
        .bind(to: passwordValidationOutlet.rx.validationResult)
        .disposed(by: disposeBag)

    viewModel.validatedPasswordRepeated
        .bind(to: repeatedPasswordValidationOutlet.rx.validationResult)
        .disposed(by: disposeBag)

    viewModel.signingIn
        .bind(to: signingUpOulet.rx.isAnimating)
        .disposed(by: disposeBag)

    viewModel.signedIn
        .subscribe(onNext: { signedIn in
            print("User signed in \(signedIn)")
        })
        .disposed(by: disposeBag)
    //}
}

```

```

        let tapBackground = UITapGestureRecognizer()
        tapBackground.rx.event
            .subscribe(onNext: { [weak self] _ in
                self?.view.endEditing(true)
            })
            .disposed(by: disposeBag)
        view.addGestureRecognizer(tapBackground)
    }
}

```

将用户行为传入给 **ViewModel**:

- **username** 将用户名输入框的当前文本传入
- **password** 将密码输入框的当前文本传入
- ...

将 **ViewModel** 的输出状态显示出来:

- **validatedUsername** 用对应的 `label` 将用户名验证结果显示出来
- **validatedPassword** 用对应的 `label` 将密码验证结果显示出来
- ...

整体结构

以下是全部的核心代码:

```

// ViewModel
class GithubSignupViewModel1 {
    // outputs {

        let validatedUsername: Observable<ValidationResult>
        let validatedPassword: Observable<ValidationResult>
        let validatedPasswordRepeated: Observable<ValidationResult>

        // Is signup button enabled
        let signupEnabled: Observable<Bool>

        // Has user signed in
        let signedIn: Observable<Bool>

        // Is signing process in progress
        let signingIn: Observable<Bool>

    // }

    init(input: (
        username: Observable<String>,
        password: Observable<String>,
        repeatedPassword: Observable<String>,

```

```

        loginTaps: Observable<Void>
    ),
    dependency: (
        API: GitHubAPI,
        validationService: GitHubValidationService,
        wireframe: Wireframe
    )
) {
    let API = dependency.API
    let validationService = dependency.validationService
    let wireframe = dependency.wireframe

    /**
     Notice how no subscribe call is being made.
     Everything is just a definition.

     Pure transformation of input sequences to output sequences.
    */
}

validatedUsername = input.username
    .flatMapLatest { username in
        return validationService.validateUsername(username)
            .observeOn(MainScheduler.instance)
            .catchErrorJustReturn(.failed(message: "Error contacting server"))
    }
    .share(replay: 1)

validatedPassword = input.password
    .map { password in
        return validationService.validatePassword(password)
    }
    .share(replay: 1)

validatedPasswordRepeated = Observable.combineLatest(input.password, input.repeatedPassword, resultSelector: validationService.validateRepeatedPassword)
    .share(replay: 1)

let signingIn = ActivityIndicator()
self.signingIn = signingIn.asObservable()

let usernameAndPassword = Observable.combineLatest(input.username, input.password) { ($0, $1) }

signedIn = input.loginTaps.withLatestFrom(usernameAndPassword)
    .flatMapLatest { (username, password) in
        return API.signup(username, password: password)
            .observeOn(MainScheduler.instance)
            .catchErrorJustReturn(false)
            .trackActivity(signingIn)
    }
}

```

```

        .flatMapLatest { loggedIn -> Observable<Bool> in
            let message = loggedIn ? "Mock: Signed in to GitHub." : "Mock: Sign
in to GitHub failed"
            return wireframe.promptFor(message, cancelAction: "OK", actions: []
)
        }
        // propagate original value
        .map { _ in
            loggedIn
        }
    }
    .share(replay: 1)

    signupEnabled = Observable.combineLatest(
        validatedUsername,
        validatedPassword,
        validatedPasswordRepeated,
        signingIn.asObservable()
    ) { username, password, repeatPassword, signingIn in
        username.isValid &&
        password.isValid &&
        repeatPassword.isValid &&
        !signingIn
    }
    .distinctUntilChanged()
    .share(replay: 1)
}

}

// ViewController
class GitHubSignupViewController1 : ViewController {
    @IBOutlet weak var usernameOutlet: UITextField!
    @IBOutlet weak var usernameValidationOutlet: UILabel!

    @IBOutlet weak var passwordOutlet: UITextField!
    @IBOutlet weak var passwordValidationOutlet: UILabel!

    @IBOutlet weak var repeatedPasswordOutlet: UITextField!
    @IBOutlet weak var repeatedPasswordValidationOutlet: UILabel!

    @IBOutlet weak var signupOutlet: UIButton!
    @IBOutlet weak var signingUpOulet: UIActivityIndicatorView!

    override func viewDidLoad() {
        super.viewDidLoad()

        let viewModel = GithubSignupViewModel1(
            input: (
                username: usernameOutlet.rx.text.orEmpty.asObservable(),
                password: passwordOutlet.rx.text.orEmpty.asObservable(),
                repeatedPassword: repeatedPasswordOutlet.rx.text.orEmpty.asObservab
le(),

```

```

        loginTaps: signupOutlet.rx.tap.asObservable()
    ),
    dependency: (
        API: GitHubDefaultAPI.sharedAPI,
        validationService: GitHubDefaultValidationService.sharedValidations
    ervice,
        wireframe: DefaultWireframe.shared
    )
)

// bind results to {
viewModel.signupEnabled
    .subscribe(onNext: { [weak self] valid in
        self?.signupOutlet.isEnabled = valid
        self?.signupOutlet.alpha = valid ? 1.0 : 0.5
    })
    .disposed(by: disposeBag)

viewModel.validatedUsername
    .bind(to: usernameValidationOutlet.rx.validationResult)
    .disposed(by: disposeBag)

viewModel.validatedPassword
    .bind(to: passwordValidationOutlet.rx.validationResult)
    .disposed(by: disposeBag)

viewModel.validatedPasswordRepeated
    .bind(to: repeatedPasswordValidationOutlet.rx.validationResult)
    .disposed(by: disposeBag)

viewModel.signingIn
    .bind(to: signingUpOulet.rx.isAnimating)
    .disposed(by: disposeBag)

viewModel.signedIn
    .subscribe(onNext: { signedIn in
        print("User signed in \(signedIn)")
    })
    .disposed(by: disposeBag)
//}

let tapBackground = UITapGestureRecognizer()
tapBackground.rx.event
    .subscribe(onNext: { [weak self] _ in
        self?.view.endEditing(true)
    })
    .disposed(by: disposeBag)
view.addGestureRecognizer(tapBackground)
}
}

```



[这里](#)还有一个 **Driver** 版的演示代码，有兴趣的同学可以了解一下。

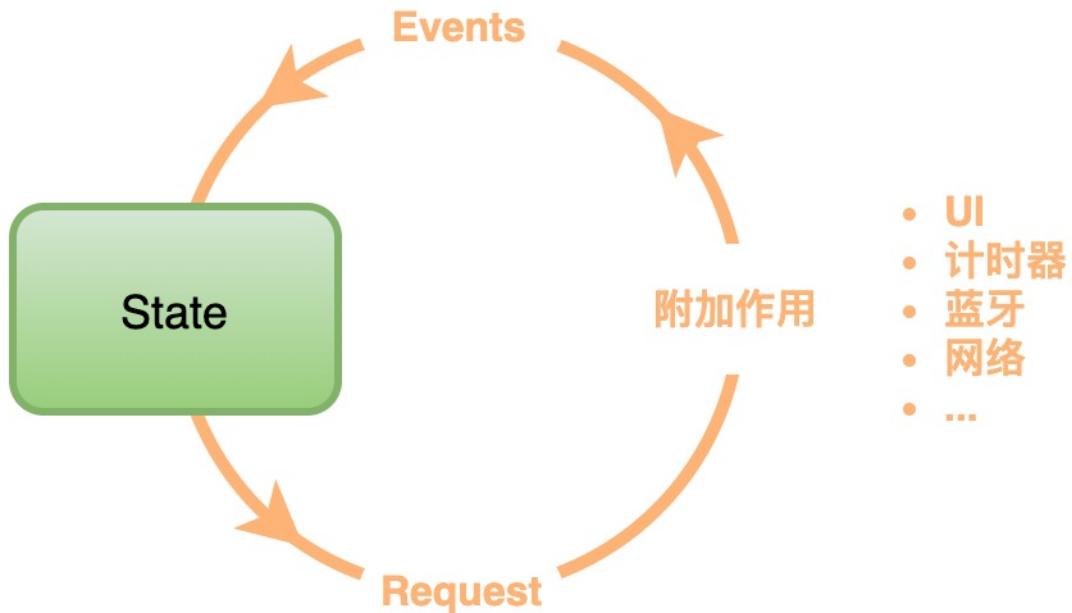
RxFEedback

作者

Krunoslav Zaher 是 RxFEedback 的作者。他也是 RxSwift 的创始人以及 ReactiveX 组织的核心成员。他有 16 年以上的编程经验（VR 引擎，BPM 系统，移动端应用程序，机器人等），最近在研究响应式编程。

介绍

RxSwift 最简单的架构



```
typealias Feedback<State, Event> = (Observable<State>) -> Observable<Event>

public static func system<State, Event>(
    initialState: State,
    reduce: @escaping (State, Event) -> State,
    feedback: Feedback<State, Event>...
) -> Observable<State>
```

为什么？

- 直接
 - 已经发生 -> Event

- 即将发生 -> Request
- 执行 Request -> Feedback loop
- 声明式
 - 首先系统行为被明确声明出来，然后在调用 subscribe 后开始运作 => 编译时就保证了不会有“未处理状态”
- 容易调试
 - 大多数逻辑是 纯函数，可以通过 xCode 调试器调试，或者将命令打印出来
- 适用于任何级别
 - 整个系统
 - 应用程序 (state 被储存在数据库中，CoreData, Firebase, Realm)
 - view controller (state 被储存在 system 操作符)
 - 在 feedback loop 中 (feedback loop 中 调用另一个 system 操作符)
- 容易做依赖注入
- 易测试
 - Reducer 是 纯函数，只需调用他并断言结果即可
 - 伴随 附加作用 的测试 -> TestScheduler
- 可以处理循环依赖
- 完全从 附加作用 中分离业务逻辑
 - 业务逻辑可以在不同平台之间转换

示例



```

Observable.system(
    initialState: 0,
    reduce: { (state, event) -> State in
        switch event {
            case .increment:
                return state + 1
            case .decrement:
                return state - 1
        }
    },
    scheduler: MainScheduler.instance,
    feedback:
        // UI is user feedback
        bind(self) { me, state -> Bindings<Event> in
            let subscriptions = [
                state.map(String.init).bind(to: me.label.rx.text)
            ]
        }
)

```

```

        let events = [
            me.plus.rx.tap.map { Event.increment },
            me_MINUS_.rx.tap.map { Event.decrement }
        ]

        return Bindings(
            subscriptions: subscriptions,
            events: events
        )
    }
)

```

这是一个简单计数的例子，只是用于演示 RxFeedback 架构。

State

系统状态用 **State** 表示：

```
typealias State = Int
```

- 这里的状态就是计数的数值

Event

事件用 **Event** 表示：

```

enum Event {
    case increment
    case decrement
}

```

- `increment` 增加数值事件
- `decrement` 减少数值事件

当产生 **Event** 时更新状态：

```

Observable.system(
    initialState: 0,
    reduce: { (state, event) -> State in
        switch event {
            case .increment:
                return state + 1
            case .decrement:
                return state - 1
        }
    },
    scheduler: MainScheduler.instance,
)

```

```
    feedback: ...
)
```

- increment 状态数值加一
- decrement 状态数值减一

Feedback Loop

将状态输出到 UI 页面上，或者将 UI 事件输入到反馈循环里面去：

```
Observable.system(
    initialState: 0,
    reduce: { ... },
    scheduler: MainScheduler.instance,
    feedback:
        // UI is user feedback
        bind(self) { me, state -> Bindings<Event> in
            let subscriptions = [
                state.map(String.init).bind(to: me.label.rx.text)
            ]

            let events = [
                me.plus.rx.tap.map { Event.increment },
                me_MINUS_rx.tap.map { Event.decrement }
            ]

            return Bindings(
                subscriptions: subscriptions,
                events: events
            )
        }
)
```

- 将状态数值用 `label` 显示出来
- 将增加按钮的点击，作为增加数值事件传入
- 将减少按钮的点击，作为减少数值事件传入

安装

CocoaPods

CocoaPods 是一个 Cocoa 项目的依赖管理工具。你可以通过以下命令安装他：

```
$ gem install cocoapods
```

将 RxFeedback 整合到项目中来，你需要在 `Podfile` 中指定他：

```
pod 'RxFeedback', '~> 3.0'
```

然后运行以下命令：

```
$ pod install
```

Carthage

[Carthage](#) 是一个分散式依赖管理工具，他将构建你的依赖并提供二进制框架。

你可以通过以下 [Homebrew](#) 命令安装 Carthage：

```
$ brew update
$ brew install carthage
```

将 RxFeedback 整合到项目中来，你需要在 `Cartfile` 中指定他：

```
github "NoTests/RxFeedback" ~> 3.0
```

运行 `carthage update` 去构建框架，然后将 `RxFedback.framework` 拖入到 Xcode 项目中来。由于 `RxFedback` 对 `RxSwift` 和 `RxCocoa` 有依赖，所以你也需要将 `RxSwift.framework` 和 `RxCocoa.framework` 拖入到 Xcode 项目中来。

Swift Package Manager

[Swift Package Manager](#) 是一个自动分发 Swift 代码的工具，他已经被集成到 Swift 编译器中。

一旦你配置好了 Swift 包，添加 RxFeedback 就非常简单了，你只需要将他添加到文件 `Package.swift` 的 `dependencies` 的值中。

```
dependencies: [
    .package(url: "https://github.com/NoTests/RxFeedback.swift.git", majorVersion: 1)
]
```

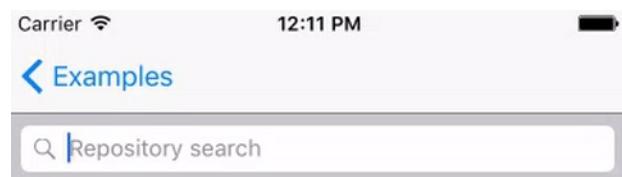
与其他架构的区别

- [Elm](#) - 非常相似，feedback loop 用作 **附加作用**，而不是 `Cmd`，要执行的 **附加作用** 被编码到 state 中，并且通过 feedback loop 完成请求
- [Redux](#) - 也很像，不过采用 feedback loops 而不是 middleware
- [Redux-Observable](#) - observables 观察状态，与视图和状态之间的 middleware
- [Cycle.js](#) - 一言难尽 :), 请咨询 [@andrestaltz](#)
- [MVVM](#) - 将状态和 **附加作用** 分离，而且不需要 View

示例

下一节将用 [Github Search](#) 来演示如何使用 RxFeedback。

Github Search (示例)



这个例子是我们经常会遇见的**Github 搜索**。它是使用 [RxFeedback](#) 重构以后的版本，你可以在这里下载[这个例子](#)。

简介

这个 App 主要有这样几个交互：

- 输入搜索关键字，显示搜索结果
- 当请求时产生错误，就给出错误提示
- 当用户滑动列表到底部时，加载下一页

State

State (状态)

- `search` 搜索关键字
- `loadNextPage` 加载下一页的 URL
- `results` 搜索结果
- `lastError` 错误
- ...

这个是用于描述当前状态：

```
fileprivate struct State {
    var search: String {
        didSet { ... }
    }
    var nextPageURL: URL?
    var shouldLoadNextPage: Bool
    var results: [Repository]
    var lastError: GitHubServiceError?
}

...

extension State {
    var loadNextPage: URL? { return ... }
}
```

我们这个例子（Github 搜索）就有这样几个状态：

- `search` 搜索关键字
- `nextPageURL` 下一页的 URL
- `shouldLoadNextPage` 是否可以加载下一页
- `results` 搜索结果
- `lastError` 搜索时产生的错误
- `loadNextPage` 加载下一页的触发

我们通常会使用这些状态来控制页面布局。

或者，用被请求的状态，触发另外一个事件。

Event

Event (事件)

- `searchChanged` 搜索关键字变更
- `response` 网络请求结果
- `scrollingNearBottom` 触发加载下页

这个是用于描述所产生的事件:

```
fileprivate enum Event {
    case searchChanged(String)
    case response(SearchRepositoriesResponse)
    case startLoadingNextPage
}
```

事件通常会使状态发生变化，然后产生一个新的状态:

```
extension State {
    ...
    static func reduce(state: State, event: Event) -> State {
        switch event {
            case .searchChanged(let search):
                var result = state
                result.search = search
                result.results = []
                return result
            case .startLoadingNextPage:
                var result = state
                result.shouldLoadNextPage = true
                return result
            case .response(.success(let response)):
                var result = state
                result.results += response.repositories
                result.shouldLoadNextPage = false
                result.nextPageURL = response.nextURL
                result.lastError = nil
                return result
            case .response(.failure(let error)):
                var result = state
                result.shouldLoadNextPage = false
                result.lastError = error
                return result
        }
    }
}
```

当发生某个事件时，更新当前状态:

- `searchChanged` 搜索关键字变更

将搜索关键字更新成当前值，并且清空搜索结果。

- `startLoadingNextPage` 触发加载下页

允许加载下一页，如果下一页的 URL 存在，就加载下一页。

- `response(.success(...))` 搜索结果返回成功

将搜索结果加入到对应的数组里面去，然后将相关状态更新。

- `response(.failure(...))` 搜索结果返回失败

保存错误状态。

Feedback Loop

Feedback Loop

订阅：

- `lastError` 控制状态 `label` 的显示
- `results` 控制 `tableView` 的内容
- `loadNextPage` 控制加载下页 `label` 的显示
- ...

事件：

- `searchChanged` 搜索关键字变更
- `scrollingNearBottom` 触发加载下页

Feedback Loop 是用来引入[附加作用](#)的。

例如，你可以将状态输出到 UI 页面上，或者将 UI 事件输入到反馈循环里面去：

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

    ...

    Driver.system(
        initialState: State.empty,
        reduce: State.reduce,
        feedback:
            // UI, user feedback
            UI.bind(self) { me, state in
                let subscriptions = [
                    state.map { $0.search }.drive(me.searchText!.rx.text),
                    state.map { $0.lastError?.displayMessage }.drive(me.status!.rx.text)
                ]
            }
    )
}
```

```

        OrHide),
            state.map { $0.results }.drive(searchResults.rx.items(cellIdentifier: "repo"))(configureRepository),
            state.map { $0.loadNextPage?.description }.drive(me.loadNextPage!.rx.textOrHide),
        ]
    let events = [
        me.searchText!.rx.text.orEmpty.changed.asDriver().map(Event.searchChanged),
        triggerLoadNextPage(state)
    ]
    return UI.Bindings(subscriptions: subscriptions, events: events)
},
// NoUI, automatic feedback
...
)
.drive()
.disposed(by: disposeBag)
}

```

这里定义的 `subscriptions` 就是如何将状态输出到 UI 页面上，而 `events` 则是如何将 UI 事件输入到反馈循环里面去。

被请求的状态

被请求的 State

- `loadNextPage` 加载下一页的 URL

被请求的状态是，用于发出异步请求，以事件的形式返回结果。

```

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

    Driver.system(
        initialState: State.empty,
        reduce: State.reduce,
        feedback:
            // UI, user feedback
            ...
            // NoUI, automatic feedback
        react(query: { $0.loadNextPage }, effects: { resource in
            return URLSession.shared.loadRepositories(resource: resource)
                .asDriver(onErrorJustReturn: .failure(.offline))
        })
    )
}

```

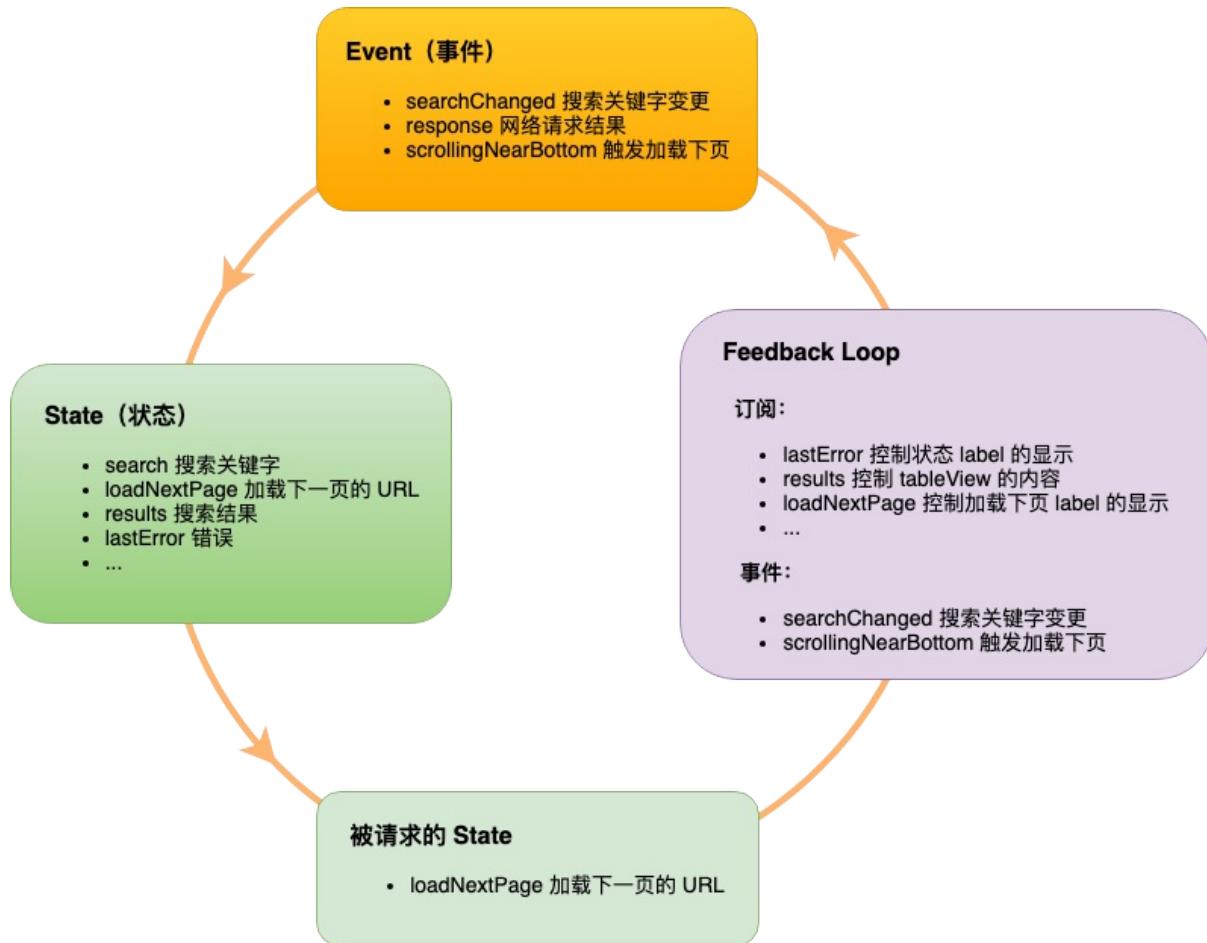
```

        .map(Event.response)
    })
)
.drive()
.disposed(by: disposeBag)
}

```

这里 `loadNextPage` 就是被请求的状态，当状态 `loadNextPage` 不为 `nil` 时，就请求加载下一页。

整体结构



现在我们看一下这个例子整体结构，这样可以帮助你理解这种架构。然后，以下是核心代码：

```

...
fileprivate struct State {
    var search: String {
        didSet {
            if search.isEmpty {
                self.nextPageURL = nil
                self.shouldLoadNextPage = false
                self.results = []
                self.lastError = nil
            }
        }
    }
}

```

```

        return
    }
    self.nextPageURL = URL(string: "https://api.github.com/search/repositories?q=\(search.URLEscaped)")
    self.shouldLoadNextPage = true
    self.lastError = nil
}
}

var nextPageURL: URL?
var shouldLoadNextPage: Bool
var results: [Repository]
var lastError: GitHubServiceError?
}

fileprivate enum Event {
    case searchChanged(String)
    case response(SearchRepositoriesResponse)
    case startLoadingNextPage
}

// transitions
extension State {
    static var empty: State {
        return State(search: "", nextPageURL: nil, shouldLoadNextPage: true, results: [], lastError: nil)
    }
    static func reduce(state: State, event: Event) -> State {
        switch event {
        case .searchChanged(let search):
            var result = state
            result.search = search
            result.results = []
            return result
        case .startLoadingNextPage:
            var result = state
            result.shouldLoadNextPage = true
            return result
        case .response(.success(let response)):
            var result = state
            result.results += response.repositories
            result.shouldLoadNextPage = false
            result.nextPageURL = response.nextURL
            result.lastError = nil
            return result
        case .response(.failure(let error)):
            var result = state
            result.shouldLoadNextPage = false
            result.lastError = error
            return result
        }
    }
}

```

```

        }

    }

    // queries
    extension State {
        var loadNextPage: URL? {
            return self.shouldLoadNextPage ? self.nextPageURL : nil
        }
    }

    class GithubPaginatedSearchViewController: UIViewController {
        @IBOutlet weak var searchText: UISearchBar?
        @IBOutlet weak var searchResults: UITableView?
        @IBOutlet weak var status: UILabel?
        @IBOutlet weak var loadNextPage: UILabel?

        private let disposeBag = DisposeBag()

        override func viewDidLoad() {
            super.viewDidLoad()

            let searchResults = self.searchResults!

            searchResults.register(UITableViewCell.self, forCellReuseIdentifier: "repo")
        }

        let triggerLoadNextPage: (Driver<State>) -> Driver<Event> = { state in
            return state.flatMapLatest { state -> Driver<Event> in
                if state.shouldLoadNextPage {
                    return Driver.empty()
                }

                return searchResults.rx.nearBottom.map { _ in Event.startLoadingNextPage }
            }
        }

        func configureRepository(_: Int, repo: Repository, cell: UITableViewCell) {
            cell.textLabel?.text = repo.name
            cell.detailTextLabel?.text = repo.url.description
        }

        let bindUI: (Driver<State>) -> Driver<Event> = UI.bind(self) { me, state in
            let subscriptions = [
                state.map { $0.search }.drive(me.searchText!.rx.text),
                state.map { $0.lastError?.displayMessage }.drive(me.status!.rx.textOrHide),
                state.map { $0.results }.drive(searchResults.rx.items(cellIdentifier: "repo"))(configureRepository),
                state.map { $0.loadNextPage?.description }.drive(me.loadNextPage!.rx.textOrHide),
            ]
        }
    }
}

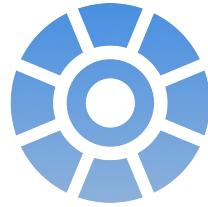
```

```
        ]
    let events = [
        me.searchText!.rx.text.orEmpty.changed.asDriver().map(Event.searchChanged),
        triggerLoadNextPage(state)
    ]
    return UI.Bindings(subscriptions: subscriptions, events: events)
}

Driver.system(
    initialState: State.empty,
    reduce: State.reduce,
    feedback:
    // UI, user feedback
    bindUI,
    // NoUI, automatic feedback
    react(query: { $0.loadNextPage }, effects: { resource in
        return URLSession.shared.loadRepositories(resource: resource)
            .asDriver(onErrorJustReturn: .failure(.offline))
            .map(Event.response)
    })
)
)
.drive()
.disposed(by: disposeBag)
}
}
...

```

这是使用 RxFeedback 重构以后的 **Github Search**。你可以对比一下使用 [ReactorKit](#) 重构以后的 [Github Search](#) 两者有许多相似之处。



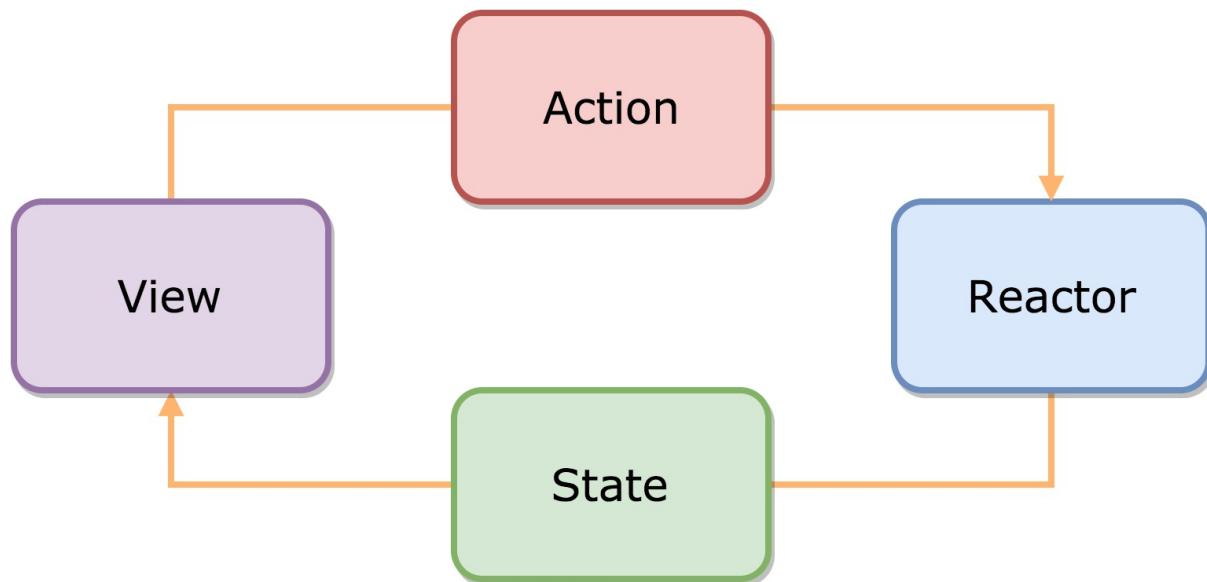
ReactorKit

作者

Jeon Suyeol 是 [ReactorKit](#) 的作者。他也发布了一些富有创造性的框架，如 [Then](#), [URLNavigator](#), [SwiftyImage](#) 以及一些开源项目 [RxTodo](#), [Drrrible](#)。他也是多个组织的成员 [RxSwiftCommunity](#), [Moya](#), [SwiftKorea](#)。

介绍

[ReactorKit](#) 结合了 [Flux](#) 和[响应式编程](#)。用户行为和页面状态都是通过序列相互传递。这些序列都是单向的：页面只能发出用户行为，然而反应器（Reactor）只能发出状态。



View

View 用于展示数据。`ViewController` 和 `Cell` 都可以看作是 **View**。**View** 将用户输入绑定到 **Action** 的序列上，同时将页面状态绑定到 UI 组件上。

定义一个 **View** 只需要让它遵循 `View` 协议即可。然后你的类将自动获得一个 `reactor` 属性。这个属性应该在 **View** 的外面被设置：

```

class ProfileViewController: UIViewController, View {
    var disposeBag = DisposeBag()
}

profileViewController.reactor = UserViewReactor() // 注入 reactor

```

当 `reactor` 属性被设置时，`bind(reactor:)` 方法就会被调用。执行这个方法来进行用户输入绑定和状态输出绑定。

```

func bind(reactor: ProfileViewReactor) {
    // action (View -> Reactor)
    refreshButton.rx.tap.map { Reactor.Action.refresh }
        .bind(to: reactor.action)
        .disposed(by: self.disposeBag)

    // state (Reactor -> View)
    reactor.state.map { $0.isFollowing }
        .bind(to: followButton.rx.isSelected)
        .disposed(by: self.disposeBag)
}

```

Reactor

Reactor 是与 UI 相互独立的一层，主要负责状态管理。**Reactor** 最重要的作用就是将业务逻辑从 **View** 中抽离。每一个 **View** 都有对应的 **Reactor** 并且将所有的逻辑代理给 **Reactor**。**Reactor** 不需要依赖 **View**，所以它很容易被测试。

遵循 `Reactor` 协议即可定义一个 **Reactor**。这个协议需要定义三个类型：`Action`，`Mutation` 和 `State`。它也需要一个 `initialState` 属性。

```

class ProfileViewReactor: Reactor {
    // 代表用户行为
    enum Action {
        case refreshFollowingStatus(Int)
        case follow(Int)
    }

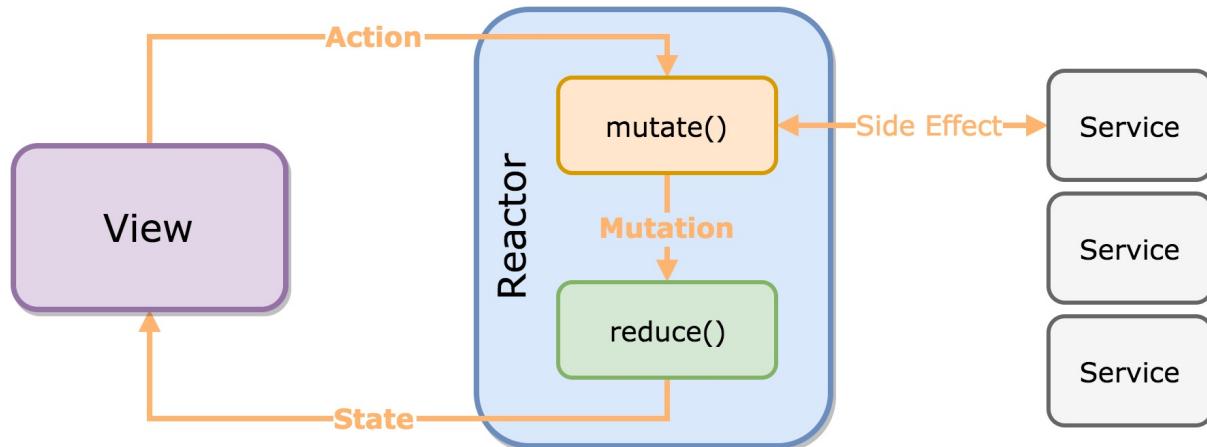
    // 代表附加作用
    enum Mutation {
        case setFollowing(Bool)
    }

    // 代表页面状态
    struct State {
        var isFollowing: Bool = false
    }
}

```

```
let initialState: State = State()
}
```

Action 代表用户行为，**State** 代表页面状态。**Mutation** 是 **Action** 和 **State** 的桥梁。**Reactor** 通过两步将用户行为序列转换为页面状态序列：`mutate()` 和 `reduce()`。



mutate()

`mutate()` 接收一个 **Action**，然后创建一个 `Observable<Mutation>`。

```
func mutate(action: Action) -> Observable<Mutation>
```

每种**附加作用**，如，异步操作，API 调用都是在这个方法内执行。

```
func mutate(action: Action) -> Observable<Mutation> {
    switch action {
        case let .refreshFollowingStatus(userID): // receive an action
            return UserAPI.isFollowing(userID) // create an API stream
                .map { (isFollowing: Bool) -> Mutation in
                    return Mutation.setFollowing(isFollowing) // convert to Mutation stream
                }

        case let .follow(userID):
            return UserAPI.follow()
                .map { _ -> Mutation in
                    return Mutation.setFollowing(true)
                }
    }
}
```

reduce()

`reduce()` 通过旧的 **State** 以及 **Mutation** 创建一个新的 **State**。

```
func reduce(state: State, mutation: Mutation) -> State
```

这个方法是一个纯函数。它将同步的返回一个 **State**。不会产生其他的作用。

```
func reduce(state: State, mutation: Mutation) -> State {
    var state = state // create a copy of the old state
    switch mutation {
        case let .setFollowing(isFollowing):
            state.isFollowing = isFollowing // manipulate the state, creating a new state
            return state // return the new state
    }
}
```

transform()

`transform()` 转换每一种序列。有三种转换方法：

```
func transform(action: Observable<Action>) -> Observable<Action>
func transform(mutation: Observable<Mutation>) -> Observable<Mutation>
func transform(state: Observable<State>) -> Observable<State>
```

执行这些方法可以转换或者组合其他的序列。例如，`transform(mutation:)` 最适合用来组合一个全局事件，生成一个 **Mutation** 序列。

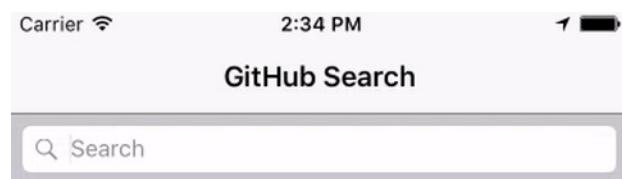
它也可用来做调试：

```
func transform(action: Observable<Action>) -> Observable<Action> {
    return action.debug("action") // Use RxSwift's debug() operator
}
```

示例

下一节将用 [Github Search](#) 来演示如何使用 [ReactorKit](#)。

Github Search (示例)



我们还是使用**Github** 搜索来演示如何使用 [ReactorKit](#)。这个例子是使用 [ReactorKit](#) 重构以后的版本，你可以在这里下载[这个例子](#)。

简介

这个 App 主要有这样几个交互：

- 输入搜索关键字，显示搜索结果
- 当用户滑动列表到底部时，加载下一页
- 当用户点击某一条搜索结果时，用 Safari 打开链接

Action

Action 用于描述用户行为：

```
enum Action {
```

```
    case updateQuery(String?)  
    case loadNextPage  
}
```

-
- `updateQuery` 搜索关键字变更
 - `loadNextPage` 触发加载下页
-

Mutation

Mutation 用于描述状态变更：

```
enum Mutation {  
    case setQuery(String?)  
    case setRepos([String], nextPage: Int?)  
    case appendRepos([String], nextPage: Int?)  
    case setLoadingNextPage(Bool)  
}
```

- `setQuery` 更新搜索关键字
 - `setRepos` 更新搜索结果
 - `appendRepos` 添加搜索结果
 - `setLoadingNextPage` 设置是否正在加载下一页
-

State

这个是用于描述当前状态：

```
struct State {  
    var query: String?  
    var repos: [String] = []  
    var nextPage: Int?  
    var isLoadingNextPage: Bool = false  
}
```

- `query` 搜索关键字
- `repos` 搜索结果
- `nextPage` 下一页页数
- `isLoadingNextPage` 是否正在加载下一页

我们通常会使用这些状态来控制页面布局。

mutate()

将 Action 转换为 Mutation:

```

func mutate(action: Action) -> Observable<Mutation> {
    switch action {
    case let .updateQuery(query):
        return Observable.concat([
            // 1) set current state's query (.setQuery)
            Observable.just(Mutation.setQuery(query)),

            // 2) call API and set repos (.setRepos)
            self.search(query: query, page: 1)
                // cancel previous request when the new `updateQuery` action is fired
                .takeUntil(self.action.filter(isUpdateQueryAction))
                .map { Mutation.setRepos($0, nextPage: $1) },
        ])
    }

    case .loadNextPage:
        guard !self.currentState.isLoadingNextPage else { return Observable.empty() }
        // prevent from multiple requests
        guard let page = self.currentState.nextPage else { return Observable.empty() }
    }
    return Observable.concat([
        // 1) set loading status to true
        Observable.just(Mutation.setLoadingNextPage(true)),

        // 2) call API and append repos
        self.search(query: self.currentState.query, page: page)
            .takeUntil(self.action.filter(isUpdateQueryAction))
            .map { Mutation.appendRepos($0, nextPage: $1) },

        // 3) set loading status to false
        Observable.just(Mutation.setLoadingNextPage(false)),
    ])
}
}

```

- 当用户输入一个新的搜索关键字时，就从服务器请求 repos，然后转换成更新 repos 事件 (Mutation)。
- 当用户触发加载下页时，就从服务器请求 repos，然后转换成添加 repos 事件。

reduce()

reduce() 通过旧的 State 以及 Mutation 创建一个新的 State:

```

func reduce(state: State, mutation: Mutation) -> State {
    switch mutation {

```

```

case let .setQuery(query):
    var newState = state
    newState.query = query
    return newState

case let .setRepos(repos, nextPage):
    var newState = state
    newState.repos = repos
    newState.nextPage = nextPage
    return newState

case let .appendRepos(repos, nextPage):
    var newState = state
    newState.repos.append(contentsOf: repos)
    newState.nextPage = nextPage
    return newState

case let . setLoadingNextPage(isLoadingNextPage):
    var newState = state
    newState.isLoadingNextPage = isLoadingNextPage
    return newState
}

}

```

- `setQuery` 更新搜索关键字
- `setRepos` 更新搜索结果，以及下一页页数
- `appendRepos` 添加搜索结果，以及下一页页数
- `setLoadingNextPage` 设置是否正在加载下一页

bind(reactor:)

在 `View` 层进行用户输入绑定和状态输出绑定：

```

func bind(reactor: GitHubSearchViewReactor) {
    // Action
    searchBar.rx.text
        .throttle(0.3, scheduler: MainScheduler.instance)
        .map { Reactor.Action.updateQuery($0) }
        .bind(to: reactor.action)
        .disposed(by: disposeBag)

    tableView.rx.contentOffset
        .filter { [weak self] offset in
            guard let `self` = self else { return false }
            guard self.tableView.frame.height > 0 else { return false }
            return offset.y + self.tableView.frame.height >= self.tableView.contentSize.height - 100
        }
}

```

```

    }

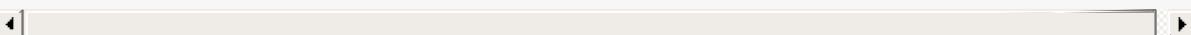
    .map { _ in Reactor.Action.loadNextPage }
    .bind(to: reactor.action)
    .disposed(by: disposeBag)

    // State
    reactor.state.map { $0.repos }
        .bind(to: tableView.rx.items(cellIdentifier: "cell")) { indexPath, repo, cell in

            cell.textLabel?.text = repo
        }
        .disposed(by: disposeBag)

    // View
    tableView.rx.itemSelected
        .subscribe(onNext: { [weak self, weak reactor] indexPath in
            guard let `self` = self else { return }
            self.tableView.deselectRow(at: indexPath, animated: false)
            guard let repo = reactor?.currentState.repos[indexPath.row] else { return }
            guard let url = URL(string: "https://github.com/\(repo)") else { return }
            let viewController = SFSafariViewController(url: url)
            self.present(viewController, animated: true, completion: nil)
        })
        .disposed(by: disposeBag)
    }

```



- 将用户更改输入关键字行为绑定到用户行为上
- 将用户要求加载下一页行为绑定到用户行为上
- 将搜索结果输出到列表页上
- 当用户点击某一条搜索结果时，用 Safari 打开链接

整体结构

我们已经了解 [ReactorKit](#) 每一个组件的功能了，现在我们看一下完整的核心代码：

GitHubSearchViewReactor.swift

```

final class GitHubSearchViewReactor: Reactor {
    enum Action {
        case updateQuery(String?)
        case loadNextPage
    }

    enum Mutation {
        case setQuery(String?)
        case setRepos([String], nextPage: Int?)
        case appendRepos([String], nextPage: Int?)
    }
}

```

```

        case setLoadingNextPage(Bool)
    }

    struct State {
        var query: String?
        var repos: [String] = []
        var nextPage: Int?
        var isLoadingNextPage: Bool = false
    }

    let initialState = State()

    func mutate(action: Action) -> Observable<Mutation> {
        switch action {
        case let .updateQuery(query):
            return Observable.concat([
                // 1) set current state's query (.setQuery)
                Observable.just(Mutation.setQuery(query)),

                // 2) call API and set repos (.setRepos)
                self.search(query: query, page: 1)
                    // cancel previous request when the new `updateQuery` action is fired
                    .takeUntil(self.action.filter(isUpdateQueryAction))
                    .map { Mutation.setRepos($0, nextPage: $1) },
            ])
        }

        case .loadNextPage:
            guard !self.currentState.isLoadingNextPage else { return Observable.empty() }
            // prevent from multiple requests
            guard let page = self.currentState.nextPage else { return Observable.empty() }
        }
        return Observable.concat([
            // 1) set loading status to true
            Observable.just(Mutation.setLoadingNextPage(true)),

            // 2) call API and append repos
            self.search(query: self.currentState.query, page: page)
                .takeUntil(self.action.filter(isUpdateQueryAction))
                .map { Mutation.appendRepos($0, nextPage: $1) },

            // 3) set loading status to false
            Observable.just(Mutation.setLoadingNextPage(false)),
        ])
    }

    func reduce(state: State, mutation: Mutation) -> State {
        switch mutation {
        case let .setQuery(query):
            var newState = state
            newState.query = query

```

```

        return newState

    case let .setRepos(repos, nextPage):
        var newState = state
        newState.repos = repos
        newState.nextPage = nextPage
        return newState

    case let .appendRepos(repos, nextPage):
        var newState = state
        newState.repos.append(contentsOf: repos)
        newState.nextPage = nextPage
        return newState

    case let . setLoadingNextPage(isLoadingNextPage):
        var newState = state
        newState.isLoadingNextPage = isLoadingNextPage
        return newState
    }

}

...
}

```

GitHubSearchViewController.swift

```

class GitHubSearchViewController: UIViewController, View {
    @IBOutlet var searchBar: UISearchBar!
    @IBOutlet var tableView: UITableView!

    var disposeBag = DisposeBag()

    override func viewDidLoad() {
        super.viewDidLoad()
        tableView.contentInset.top = 44 // search bar height
        tableView.scrollIndicatorInsets.top = tableView.contentInset.top
    }

    func bind(reactor: GitHubSearchViewReactor) {
        // Action
        searchBar.rx.text
            .throttle(0.3, scheduler: MainScheduler.instance)
            .map { Reactor.Action.updateQuery($0) }
            .bind(to: reactor.action)
            .disposed(by: disposeBag)

        tableView.rx.contentOffset
            .filter { [weak self] offset in
                guard let `self` = self else { return false }

```

```
        guard self.tableView.frame.height > 0 else { return false }
        return offset.y + self.tableView.frame.height >= self.tableView.contentSize
.height - 100
    }
    .map { _ in Reactor.Action.loadNextPage }
    .bind(to: reactor.action)
    .disposed(by: disposeBag)

    // State
    reactor.state.map { $0.repos }
        .bind(to: tableView.rx.items(cellIdentifier: "cell")) { indexPath, repo, cell
in
        cell.textLabel?.text = repo
    }
    .disposed(by: disposeBag)

    // View
    tableView.rx.itemSelected
        .subscribe(onNext: { [weak self, weak reactor] indexPath in
            guard let `self` = self else { return }
            self.tableView.deselectRow(at: indexPath, animated: false)
            guard let repo = reactor?.currentState.repos[indexPath.row] else { return }
            guard let url = URL(string: "https://github.com/\(repo)") else { return }
            let viewController = SFSafariViewController(url: url)
            self.present(viewController, animated: true, completion: nil)
        })
        .disposed(by: disposeBag)
    }
}
```

这是使用 [ReactorKit](#) 重构以后的 Github Search。ReactorKit 分层非常详细，分工也是非常明确的。当你在处理大型应用程序时，这可以帮助你更好的管理代码。



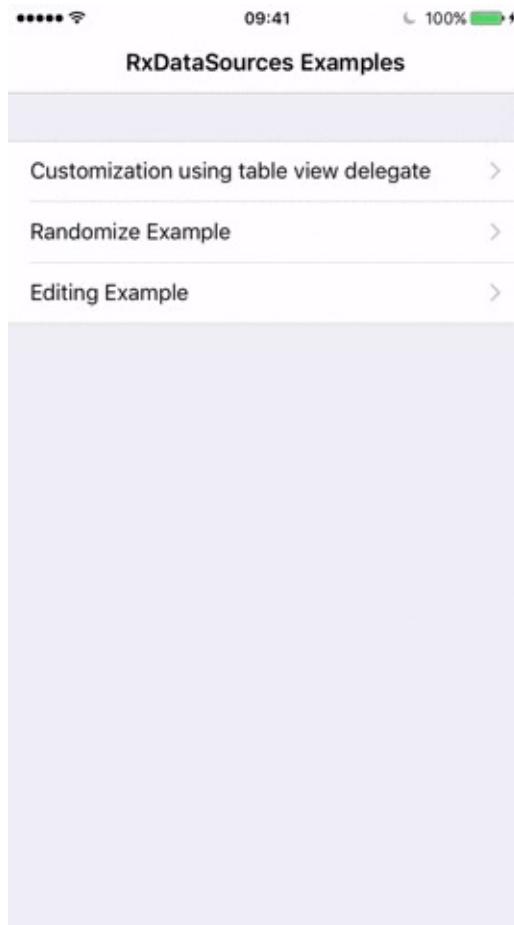
RxSwift 生态系统

RxCocoa 给 **UI框架** 提供了 **Rx** 支持，让我们能够使用按钮点击序列，输入框当前文本序列等。不过 **RxCocoa** 也只是 **RxSwift 生态系统** 中的一员。**RxSwift 生态系统**还给其他框架提供了 **Rx** 支持：

- [RxDataSources](#) - UITableView 和 UICollectionView 数据源
- [RxGesture](#) - 页面手势
- [RxMKMapView](#) - 地图
- [RxCoreMotion](#) - 陀螺仪
- [RxAlamofire](#) - 网络请求
- [RxCoreData](#) - CoreData 数据库
- [RxRealm](#) - Realm 数据库
- [RxMediaPicker](#) - 图片选择器
- [Action](#) - 行为
- [RxWebKit](#) - WebView
- [RxEventHub](#) - 全局通知
- [RxSwiftExt](#) - 添加一些有用的操作符
- ...

RxDataSources

书写 `tableView` 或 `collectionView` 的数据源是一件非常繁琐的事情，有一大堆的代理方法需要被执行。[RxDataSources](#) 可以帮助你简化这一过程。你可以用它来布局多层级的列表页，并且它还可以提供动画支持。



你只需要几行代码就可以布局一个多个 `Section` 的 `tableView` :

```
let dataSource = RxTableViewSectionedReloadDataSource<SectionModel<String, Int>>()
Observable.just([SectionModel(model: "title", items: [1, 2, 3])])
    .bind(to: tableView.rx.items(dataSource: dataSource))
    .disposed(by: disposeBag)
```

你可以点击 [RxDataSources](#) 来了解更多信息。

RxAlamofire

Alamofire 是一个非常流行的网络请求框架。RxAlamofire 是用 RxSwift 封装的 Alamofire。它使得网络请求调用变得更加平滑，处理请求结果变得更简洁，更高效：

```
let stringURL = ""

// 使用 NSURLSession
let session = URLSession.sharedSession()

_ = session.rx
    .json(.get, stringURL)
    .observeOn(MainScheduler.instance)
```

```

    .subscribe { print($0) }

// 使用 Alamofire 引擎

_ = json(.get, stringURL)
.observeOn(MainScheduler.instance)
.subscribe { print($0) }

// 使用 Alamofire manager

let manager = Manager.sharedInstance

_ = manager.rx.json(.get, stringURL)
.observeOn(MainScheduler.instance)
.subscribe { print($0) }

// URLHTTPResponse + Validation + String
_ = manager.rx.request(.get, stringURL)
.flatMap {
    $0
        .validate(statusCode: 200 ..< 300)
        .validate(contentType: ["text/json"])
        .rx.string()
}
.observeOn(MainScheduler.instance)
.subscribe { print($0) }

```

你可以点击 [RxAlamofire](#) 来了解更多信息。

RxRealm

[Realm](#) 是一个十分前卫的跨平台数据库，他想要替换 **Core Data** 和 **SQLite**。[RxRealm](#) 是用 [RxSwift](#) 封装的 [Realm](#)。它使我们可以用 **Rx** 的方式监听数据变化，或者将数据写入数据库。

监听数据：

```

let realm = try! Realm()
let laps = realm.objects(Lap.self)

Observable.collection(from: laps)
.map {
    laps in "\((laps.count)) laps"
}
.subscribe(onNext: { text in
    print(text)
})

```

添加数据:

```
let realm = try! Realm()
let messages = [Message("hello"), Message("world")]

Observable.from(messages)
.subscribe(realm.rx.add())
```

删除数据:

```
let realm = try! Realm()
let messages = realm.objects(Message.self)

Observable.from(messages)
.subscribe(realm.rx.delete())
```

你可以点击 [RxRealm](#) 来了解更多信息。



ReactiveX 生态系统

我们之前提到过 [RxSwift](#) 是 [Rx](#) 的 [Swift](#) 版本。而 [ReactiveX](#) (简写: [Rx](#)) 是一个跨平台框架。它不仅可以用来写 [iOS](#)，你还可以用它来写 [Android](#), [Web](#) 前端和后台。并且每个平台都和 [RxSwift](#) 一样有一套 [Rx](#) 生态系统。[Rx](#) 支持多种编程语言，如: [Swift](#), [Java](#), [JS](#), [C#](#), [Scala](#), [Kotlin](#), [Go](#) 等。只要你掌握了其中一门语言，你很容易就能够熟悉其他的语言。

Android

[RxJava](#) 是 [Android](#) 平台上非常流行的响应式编程框架，它也是 [Rx](#) 的 [Java](#) 版本。

我们还是用 [输入验证](#) 来做演示：

[iOS \(RxSwift\) 版:](#)

```
...
let usernameValid = usernameOutlet.rx.text.orEmpty
.map { $0.characters.count >= minimalUsernameLength }
.share(replay: 1)

let passwordValid = passwordOutlet.rx.text.orEmpty
.map { $0.characters.count >= minimalPasswordLength }
.share(replay: 1)

let everythingValid = Observable
.combineLatest(usernameValid, passwordValid) { $0 && $1 }
.share(replay: 1)
```

```

usernameValid
    .bind(to: passwordOutlet.rx.isEnabled)
    .disposed(by: disposeBag)

usernameValid
    .bind(to: usernameValidOutlet.rx.isHidden)
    .disposed(by: disposeBag)

passwordValid
    .bind(to: passwordValidOutlet.rx.isHidden)
    .disposed(by: disposeBag)

everythingValid
    .bind(to: doSomethingOutlet.rx.isEnabled)
    .disposed(by: disposeBag)
...

```

Android (RxJava) 版:

```

...
final Observable<Boolean> usernameValid = RxTextView.textChanges(usernameEditText)
    .map(text -> text.length() >= minimalUsernameLength)
    .compose(Rx.shareReplay(1));

final Observable<Boolean> passwordValid = RxTextView.textChanges(passwordEditText)
    .map(text -> text.length() >= minimalPasswordLength)
    .compose(Rx.shareReplay(1));

final Observable<Boolean> everythingValid = Observable
    .combineLatest(usernameValid, passwordValid, (isUsernameValid, isPasswordValid) -> isUsernameValid && isPasswordValid)
    .compose(Rx.shareReplay(1));

disposables.add(usernameValid
    .subscribe(RxView.enabled(passwordEditText)));

disposables.add(usernameValid
    .subscribe(RxView.visibility(usernameValidTextView)));

disposables.add(passwordValid
    .subscribe(RxView.visibility(passwordValidTextView)));

disposables.add(everythingValid
    .subscribe(RxView.enabled(doSomethingButton)));
...

```

这两段代码的逻辑是一样的，一个是 **iOS (RxSwift)** 版本，另一个是 **Android (RxJava)** 版本。仔细对比以后，你会发现它们的书写方式都是差不多的。

这样一来，你就可以用同一套逻辑来写跨平台应用，而且这个应用是纯原生的。这不仅节省了开发时间，而且还提升了 App 的质量。

Web 前端

[RxJS](#) 是 **Web 前端** 平台上非常流行的响应式编程框架，它也是 [Rx](#) 的 **JS** 版本。而且主流的前端框架都提供了 [Rx](#) 支持，如：[jQuery](#), [RxJS-DOM](#), [AngularJS](#), [RxEmber](#) 等。

下面这个例子是用 [RxJS](#) 写的，它和 [GitHub 搜索](#) 十分相似，只不过他搜索的是维基百科：

```
var $input = $('#input'),
    $results = $('#results');

Rx.Observable.fromEvent($input, 'keyup')
    .map(e => e.target.value)
    .filter(text => text.length > 2)
    .throttle(500 /* ms */);
    .distinctUntilChanged();
    .flatMapLatest(searchWikipedia);
    .subscribe(data => {
        var res = data[1];
        $results.empty();
        $.each(res, (_, value) => $('- ' + value + '
').appendTo($results));
    }, error => {
        $results.empty();
        $('- Error: ' + error + '
').appendTo($results);
    });
});
```

当用户输入一个稳定的关键字后，向维基百科请求搜索结果，然后显示出来。

即便你没有学过 **Web 前端** 开发，但是只要你熟悉 [Rx](#)，以上代码你也能够看懂。

总结

由于 [Rx](#) 支持多种后台语言，如：**Java**, **JS**, **Go**。所以你也可以用它来写后台。

如果你已经能够熟练使用 [RxSwift](#)，那么你就已经具有某种“天赋”，这种“天赋”可以帮助你快速上手其他平台。你只需要学习一些和平台相关的知识，就可以写出交互相当复杂的应用程序。因为你的 [Rx](#) 技巧是可以跨平台复用的。

另外，你的学习效率也会更高，如果你在 [RxSwift](#) 中学到了某些技巧，那么这个技巧通常也可以被应用到 **Android** 或者其他的平台。如果你在 [RxJava](#) 中学到了某些技巧，那么这个技巧通常也可以被应用到 **iOS** 平台。因此，你的学习资源也就不再局限于 [RxSwift](#)，你还可以浏览其他平台上关于 [Rx](#) 的教程。

下一章将提供一些关于 RxSwift 的学习资源。

学习资源

书籍

- [RxSwift](#) - By Raywenderlich

视频

- [Learning Path: RxSwift from Start to Finish](#) - By Realm 团队
- [RxSwift in Practice](#) - By Raywenderlich
- [Reactive Programming with RxSwift](#) - By RxSwift 图书的核心作者
- [Boxue.io RxSwift Online Course](#) - 泊学 RxSwift 中文视频教程

博客

- [Marin Todorov](#) - RxSwift 图书的作者
- [Adam](#) - 富有激情的 iOS 开发者

教程

- [Getting Started With RxSwift and RxCocoa](#) - RxSwift 入门教程
- [RxSwift by Examples #1 – The basics.](#) - RxSwift 基础教程
- [ViewModel in RxSwift world](#) - MVVM 使用教程
- [8 Mistakes to Avoid while Using RxSwift—Part 1](#) - 使用 RxSwift 应该避免的 8 个错误
- [RxSwift: share vs replay vs shareReplay](#) - 几个 **share** 操作符的区别

开源项目

- [CleanArchitectureRxSwift](#) - Example of Clean Architecture of iOS app using RxSwift
- [PinPlace](#) - Routing app. Build with MVVM+RxSwift and .
- [RxTodo](#) - iOS Todo Application using RxSwift and ReactorKit
- [Drrrible](#) - Dribbble for iOS using ReactorKit
- [RxMarbles](#) - RxMarbles iOS app

关于本文档

- 更新日期: **19年5月21日**
- 首发日期: **17年9月1日**
- 对应 **RxSwift** 版本: **5.0.0**
- 使用工具: [GitBook](#)
- 托管平台: [Github Pages](#)
- 文档整理人: 罗杰 (Beeth0ven)
- 整理人邮箱: beeth0vendev@gmail.com
- 文档库地址: <https://github.com/beeth0ven/RxSwift-Chinese-Documentation>
- **RxSwift** 地址: <https://github.com/ReactiveX/RxSwift>

问题反馈

如果你发现文档存在问题，可以通过以下任意一种方式将问题反馈给作者：

- (推荐) 在存在问题页面，点击左上方的编辑页面按钮，对文档进行修正，最后提交 [Pull Request](#)
- 前往 [文档库](#) 提 [issues](#)，并注明文档哪些地方存在问题
- 加入到 **RxSwift** QQ 交流群: **871293356**，将问题反馈给整理人
- 通过邮件将问题反馈给整理人: beeth0vendev@gmail.com

文档更新日志

文档变更将被记录在此文件内。

2.0.0

19年5月21日（RxSwift 5）

- RxSwift 5 更新了什么？
 - 引入[食谱章节](#)
 - [Signal](#)
 - [RxRelay](#)
 - 纯函数
 - 附加作用
 - [共享附加作用](#)
 - 更新文档以适配 RxSwift 5
 - 更新 QQ 群号为：871293356
-

1.2.0

18年2月15日

- 纠正错别字
 - 给 [retry](#) 操作符加入演示代码
 - 给 [replay](#) 操作符加入演示代码
 - 给 [connect](#) 操作符加入演示代码
 - 给 [publish](#) 操作符加入演示代码
 - 给 [reduce](#) 操作符加入演示代码
 - 给 [skipUntil](#) 操作符加入演示代码
 - 给 [skipWhile](#) 操作符加入演示代码
 - 给 [skip](#) 操作符加入演示代码
-

1.1.0

17年12月7日

- 纠正错别字
 - 给 [takeUntil](#) 操作符加入演示代码
 - 给 [takeWhile](#) 操作符加入演示代码
-

- 给 `takeLast` 操作符加入演示代码
 - 加入 `debug` 操作符
 - 给 `AsyncSubject` 加入演示代码
 - 给 `take` 操作符加入演示代码
 - 给 `elementAt` 操作符加入演示代码
 - 给 `BehaviorSubject` 加入演示代码
-

1.0.0

17年10月18日 (RxSwift 4)

- 加入[文档电子书下载地址](#)
 - 去掉学习资源[《如何将代理转换为序列》](#)，因为 RxSwift 4 重构了 `DelegateProxy` #1379
 - 使用 `share(replay: 1)` 替换 `shareReplay(1)`
 - 给 [RxJava 演示代码](#) 中的变量加上 `final` 关键字，声明为常量
 - 示例[多层级的列表页](#)更新到 RxSwift 4，使用新的 `RxDataSources` 构建方法
 - 文档[首页](#)更新到 RxSwift 4
-

0.2.0

17年10月9日

- 给 `ReplaySubject` 加入演示代码
- 给 `PublishSubject` 加入演示代码
- 给 `distinctUntilChanged` 操作符加入演示代码
- 给 `scan` 操作符加入演示代码
- 给 `startWith` 操作符加入演示代码
- 给 `merge` 操作符加入演示代码
- (RxSwift 4) 使用 `Binder` 替换 `UIBindingObserver`，更简洁实用

0.1.1

17年9月18日

- 更新 `RxFedback` 配图，与官方保持一致
 - 修复 `Maybe` 中的描述问题
-

0.1.0

17年9月4日

- 加入学习资源《泊学 RxSwift 中文视频教程》
 - 给 `concat` 操作符加入演示代码
 - 给 `concatMap` 操作符加入演示代码
 - 将操作符列表移动到《如何选择操作符？》章节下，便于查找
 - 给 `combineLatest` 操作符加入演示代码
 - 给 `catchError` 操作符加入演示代码
 - 给 `filter` 操作符加入演示代码
 - 给 `flatMap` 操作符加入演示代码
 - 给 `flatMapLatest` 操作符加入演示代码
 - 给 `map` 操作符加入演示代码
 - 给 `zip` 操作符加入演示代码
 - 给 `withLatestFrom` 操作符加入演示代码
-

0.0.1

17年9月1日（RxSwift 3.6.1）

- 加入 56 个[操作符中文说明](#)
- 加入[图片选择器示例](#)
- 加入[多层级的列表页示例](#)
- 加入[计算器示例](#)
- 加入[MVVM 架构](#)
- 加入[RxFeedback 架构](#)
- 加入[ReactorKit 架构](#)
- 加入[RxSwift 生态系统和 ReactiveX 生态系统 章节](#)
- 加入[文档更新日志](#)
- 加入学习资源《[几个 share 操作符的区别](#)》
- 加入学习资源《[如何将代理转换为序列](#)》

食谱



这是一张食谱，记录了许多“美味佳肴”：

- RxRelay
- RxSwift 5 更新了什么？
- 纯函数
- 附加作用
- ...

RxSwift 5 更新了什么？

原文：[What's new in RxSwift 5](#) 作者：[freak4pc](#)

译注：在此感谢 [freak4pc](#) 为社区所作出的贡献。

RxSwift 5 终于发布了，我 ([freak4pc](#)) 认为这是一个很好的机会，来分享这次发布中最有价值的更新。

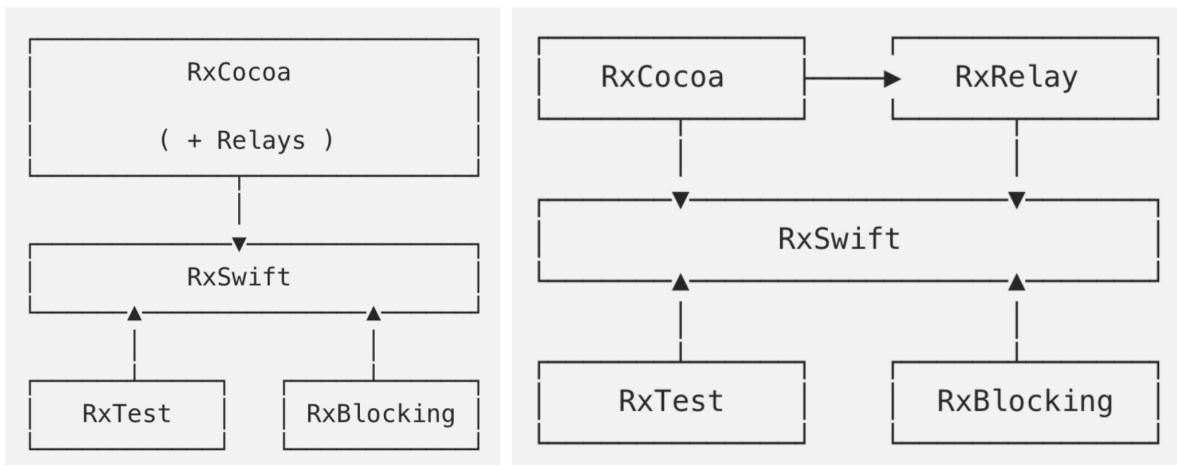
别担心，这次发布几乎是向前兼容的，只有少数弃用和重命名的 API。但它还包含了许多改进，我 ([freak4pc](#)) 将在下面详细介绍。

RxRelay 现在是一个独立的框架

[RxRelay](#) 是一个在 Subjects 之上很好的抽象层。它可以让我们发出元素，而不用担心 error 和 completed 这样的终止事件。由于它们被添加到 RxSwift 中，并且是 RxCocoa 项目的一部分。

许多开发者对此很不乐意。因为他们如果要使用 Relays 就必须引入 RxCocoa，即便他们编写的代码与 RxCocoa 没有任何关系。这其实是很不合理的。并且这样一来 Linux 用户就无法使用 Relays，因为 Linux 无法导入 RxCocoa。

由于上述原因，[我们将 Relays 拆分成一个独立的框架 - RxRelay](#) -- 并且调整 RxSwift 的依赖图，如下：



左侧： RxSwift 4 依赖图，右侧： RxSwift 5 依赖图

这样我们可以引入 [RxRelay](#)，而不需要导入整个 RxCocoa 框架。并且这也和 [RxJava](#) 保持一致，因为在 [RxJava](#) 那边他也是一个独立的框架。

注意：这是一个向前兼容的改动，由于 RxCocoa 依赖于 RxRelay。意味着，只要导入 RxCocoa 并不需要导入 RxRelay，一切就会和以前一样正常工作。

TimeInterval → DispatchTimeInterval

RxSwift 5 中重构了 `Schedulers`, 弃用了 `TimeInterval`, 转而使用 `DispatchTimeInterval`。这样就与底层时间 API 保持一致, 不会丢失精度。

他会影响到所有基于时间的操作符, 如: `debounce`, `timeout`, `delay`, `take` 等等。作为额外的收获, 他还解决了 `take` 入参的歧义, 因为之前无法判断参数代表多少秒, 还是多少个。

RxSwift 4

```
observable.delay(3, scheduler: MainScheduler.instance)
observable.throttle(0.5, scheduler: MainScheduler.instance)
observable.window(timeSpan: 2.5, count: 3, scheduler: MainScheduler.instance)
observable.take(1, scheduler: MainScheduler.instance)
```

RxSwift 5

```
observable.delay(.seconds(3), scheduler: MainScheduler.instance)
observable.throttle(.milliseconds(500), scheduler: MainScheduler.instance)
observable.window(timeSpan: .milliseconds(2500), count: 3, scheduler: MainScheduler.instance)
observable.take(.seconds(1), scheduler: MainScheduler.instance)
```

Variable 最终被弃用了

`Variable` 是早期添加到 RxSwift 的概念, 通过 “setting” 和 “getting” 他可以帮助我们, 从原先命令式的思维方式, 过渡到“响应式的思维方式”。

这种做法被证实是有问题的, 许多开发者滥用 `Variable`, 来构建 **重度命令式** 系统, 而不是 Rx 的 **声明式** 系统。这对于新手非常常见, 并且让他们无法意识到, 这是代码的坏味道。所以在 RxSwift 4.x 中 `Variable` 被轻度弃用, 仅仅给出一个运行时警告。

在 RxSwift 5.x 中, 他被[官方的正式的弃用了](#), 并且在需要时, 推荐使用 `BehaviorRelay` 或者 `BehaviorSubject`。

RxSwift 4

```
let variable = Variable("Hello, RxSwift")
variable.value = "Goodbye, RxSwift"
print(variable.value) // "Goodbye, RxSwift"
```

RxSwift 5

```
let relay = BehaviorRelay(value: "Hello, RxSwift")
relay.accept("Goodbye, RxSwift")
print(relay.value) // "Goodbye, RxSwift"
```

补充 do(on:) 重载方法

do 是一个很棒的操作符，当你想添加一些[附加作用](#)，如：打印日志。

为了与 RxJava 对齐，RxSwift 现在不仅提供 `do(onNext:)`，而且提供 `after` 重载方法，例如：`do(afterNext:)`。`onNext` 代表元素发送了，但未被转发到下游。而 `afterNext` 代表元素发送了，并已经被转发到下游。

RxSwift 4

```
Observable.of("🍎", "🍏", "🍊", "🍋")
    .do(onNext: { print("Intercepted:", $0) },
        onError: { print("Intercepted error:", $0) },
        onCompleted: { print("Completed") })
```

RxSwift 5

```
Observable.of("🍎", "🍏", "🍊", "🍋")  
    .do(onNext: { print("Intercepted:", $0) },  
        afterNext: { print("Intercepted after:", $0) },  
        onError: { print("Intercepted error:", $0) },  
        afterError: { print("Intercepted after error:", $0) },  
        onCompleted: { print("Completed") },  
        afterCompleted: { print("After completed") })
```

bind(to:) 现在支持多个观察者

在一些情况下，你不得不将流绑定到多个观察者上。在 RxSwift 4 中，你通常需要重复绑定代码：

```
isFormEnabled  
    .bind(to: txtName.rx.isEnabled)  
    .disposed(by: disposeBag)  
  
isFormEnabled  
    .bind(to: txtEmail.rx.isEnabled)  
    .disposed(by: disposeBag)  
  
isFormEnabled  
    .bind(to: txtPassword.rx.isEnabled)  
    .disposed(by: disposeBag)  
  
isFormEnabled  
    .bind(to: btnSubmit.rx.isEnabled)  
    .disposed(by: disposeBag)
```

RxSwift 5 现在支持**绑定多个观察者**:

```
isFormEnabled
    .bind(to: txtName.rx.isEnabled,
          txtEmail.rx.isEnabled,
          txtPassword.rx.isEnabled,
          btnSubmit.rx.isEnabled)
    .disposed(by: disposeBag)
```

新增 compactMap 操作符

作为开发者，你通常要处理 可选型 。为了将它拆包，社区有专门的解决办法。例如： RxSwiftExt 的 `unwrap` 操作符，或者 RxOptional 的 `filterNil` 操作符。

RxSwift 5 新增了一个新的操作符 `compactMap`，从而对齐了 [Swift 标准库](#)，将这种功能带入到核心库。

RxSwift 4

```
observable // Observable<String?>
    .filter { $0 != nil }
    .map { $0! } // Observable<String>

observable.unwrap() // RxSwiftExt

observable.filterNil() // RxOptional
```

RxSwift 5



```
observable.compactMap { $0 }
```

toArray() 现在返回 Single<T>

toArray() 操作符将所有的元素以数组的形式发送出去，在流完成时。

自从 RxSwift 诞生以来，这个操作符一直都是返回 Observable<T>，但是在特征序列被引入以后，尤其是 Single。将返回类型改为 Single 会更加合适，这不仅提供了类型安全，并且还保证该操作符只会发出一个元素。

RxSwift 4



```
observable.toArray() // Observable<T>
```

RxSwift 5



```
observable.toArray() // Single<T>
```

更新范型约束名称

RxSwift 是范型约束的重度使用者。早些时期，他使用单个字母来表示某个类型。例如，`ObservableType.E` 代表 `Observable` 的元素。

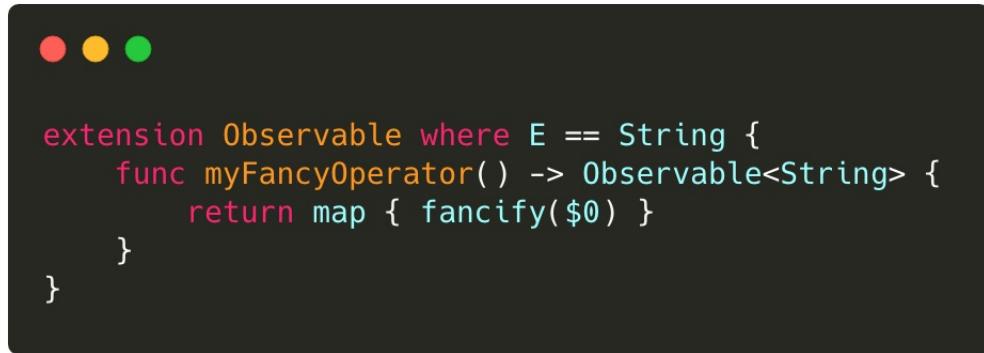
这是可行的，但是有时候会造成混淆，如 `o`，既可以代表 `Observable`，也可以代表 `Observer`。他们的首字母都是 `o`。另外 `s` 既可以代表 `Subject`，也可以代表 `Sequence`。

此外，这些单字母约束并不能提供良好的“自解释”代码，并使得项目贡献者以外的人，难以理解他们的含义。

出于这些原因，我们对私有和公共接口的大多数通用约束进行了全面修改，使其更清晰和详细。

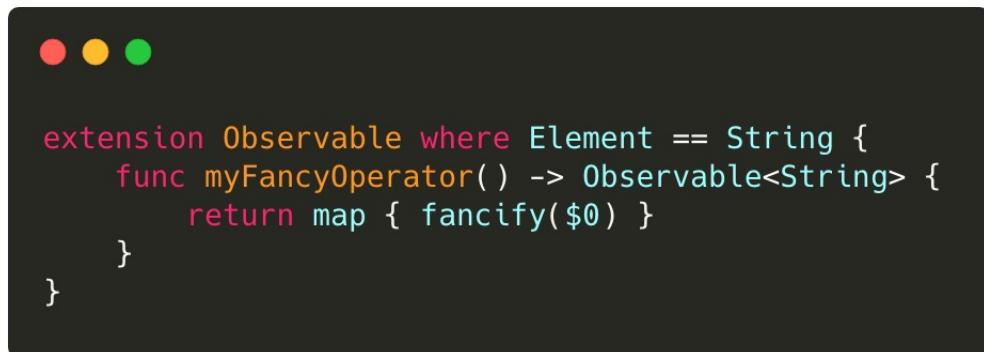
影响最大的重命名是将 `E` 和 `ElementType` 改为 `Element`。

RxSwift 4



```
extension Observable where E == String {
    func myFancyOperator() -> Observable<String> {
        return map { fancify($0) }
    }
}
```

RxSwift 5



```
extension Observable where Element == String {
    func myFancyOperator() -> Observable<String> {
        return map { fancify($0) }
    }
}
```

这次重命名的范围广泛。下面有一个完整的清单。大多数都是内部 API，只有少数与我们这些开发者相关：

- `E` 和 `ElementType` 改为 `Element`。
- `TraitType` 改为 `Trait`。
- `SharedSequence.S` 改为 `SharedSequence.SharingStrategy`。
- `o` 改为 `Observer` 和 `Source`，仅在适用的情况下。

- `c` 和 `s` 分别被改为 `Collection` 和 `Sequence`。
- `S` 改为 `Subject`，仅在适用的情况下。
- `R` 改为 `Result`。
- `ReactiveCompatible.CompatibleType` 改为 `ReactiveCompatible.ReactiveBase`。

社区项目

许多 RxSwift 社区 项目已经迁移到 RxSwift 5，并且发布了相应的版本。因此迁移过程将会非常平滑。其中一部分已经迁移的项目是：[RxSwiftExt](#), [RxDataSources](#), [RxAlamofire](#), [RxOptional](#)...

综上所叙

上面列出来的更改都是与我们开发者息息相关的。还有许多的小修小补，就超出了本文的讨论范围。例如，在Linux下完全修复与Swift 5的兼容性问题等等。

请随意查看完整的更新日志，并参与到官方库的讨论中来：<https://github.com/ReactiveX/RxSwift>

希望你喜欢这篇文章:-)

参考

- [What's new in RxSwift 5](#)
- [RxJava](#)
- [RxSwift 5](#)
- [freak4pc](#)
- [Schedulers](#)
- [timeout](#)
- [debounce](#)
- [delay](#)
- [take](#)
- [Variable](#)
- 命令式编程
- 声明式编程
- [BehaviorRelay](#)
- [BehaviorSubject](#)
- [do](#)
- [附加作用](#)
- [Single](#)

RxRelay

RxRelay 既是 可监听序列 也是 观察者。

他和 **Subjects** 相似，唯一的区别是不会接受 `onError` 或 `onCompleted` 这样的终止事件。

在将非 Rx 样式的 API 转化为 Rx 样式时，**Subjects** 是非常好用的。不过一旦 **Subjects** 接收到了终止事件 `onError` 或 `onCompleted`。他就无法继续工作了，也不会转发后续任何事件。有些时候这是合理的，但在多数场景中这并不符合我们的预期。

在这些场景中一个更严谨的做法就是，创造一种特殊的 **Subjects**，这种 **Subjects** 不会接受终止事件。有了他，我们将 API 转化为 Rx 样式时，就不必担心一个意外的终止事件，导致后续事件转发失效。

我们将这种特殊的 **Subjects** 称作 **RxRelay**：

PublishRelay

PublishRelay 就是 **PublishSubject** 去掉终止事件 `onError` 或 `onCompleted`。

演示

```
let disposeBag = DisposeBag()
let relay = PublishRelay<String>()

relay
    .subscribe { print("Event:", $0) }
    .disposed(by: disposeBag)

relay.accept(" ")
relay.accept(" ")
```

输出结果：

```
Event: next( )
Event: next( )
```

BehaviorRelay

BehaviorRelay 就是 **BehaviorSubject** 去掉终止事件 `onError` 或 `onCompleted`。

演示

```
let disposeBag = DisposeBag()
```

```
let relay = BehaviorRelay(value: "")  
  
relay  
.subscribe { print("Event:", $0) }  
.disposed(by: disposeBag)  
  
relay.accept("")  
relay.accept("")
```

输出结果：

```
Event: next()  
Event: next()  
Event: next()
```

BehaviorRelay 将取代 **Variable**，因为 **Variable** 很容易会引导我们使用[命令式编程](#)，而不是[声明式编程](#)。

纯函数

什么是纯函数?

在函数式编程里我们会经常谈到这两个概念。一个是 [纯函数](#)。另一个是 [附加作用](#)（副作用）。这里我们就结合实际来介绍一下 [纯函数](#) 和 [附加作用](#)。

下面我们给出两个函数 `increaseA` 和 `increaseB`，他们其中一个是 [纯函数](#)，另一个不是 [纯函数](#)：

```
var state = 0

func increaseA() {
    state += 1
}

increaseA()

print(state) // 结果: 1
```

```
func increaseB(state: Int) -> Int {
    return state + 1
}

let state = increaseB(state: 0)

print(state) // 结果: 1
```

他们的作用差不多，使 `state + 1`，我们可以猜测一下 `increaseA` 和 `increaseB` 哪一个是 [纯函数](#)？

...

...

...

... 经过 10 秒后

现在公布答案： `increaseB` 是 [纯函数](#)，`increaseA` 不是 [纯函数](#)

为什么 `increaseB` 是 [纯函数](#)？

因为他特别 [纯洁](#)：除了用入参 `state` 计算返回值以外没做任何其他的事情。

那为什么 `increaseA` 不是 [纯函数](#)？

因为他修改了函数本体以外的值 `state`，他拥有这个 [附加作用](#)，因此他 [并不纯洁](#) 就不是 [纯函数](#)。

我们再来做以下两个测试，然后猜测他们能不能测试成功：

```
func testIncreaseA() {
    increaseA()
    state == 1 // 结果: ???
}
```

```
func testIncreaseB() {
    let state = increaseB(state: 0)
    state == 1 // 结果: true
}
```

...

...

...

... 经过 20 秒后

嗯... 这里我们可以肯定第二个测试 `testIncreaseB` 会成功。`0 + 1` 肯定等于 `1`。那第一个测试呢？这可不好说了，我们并不知道 `increaseA` 是在什么环境下被调用的，不知道在这个环境下初始 `state` 是多少。如果他是 `0` 那测试就会成功的，如果他不是 `0` 那测试就会失败的。因此在不知道所处环境时，我们无法判断测试是否会成功。

由于 `increaseA` 存在修改外部 `state` 的 **附加作用** 所以他不是 **纯函数**。事实上如果函数有以下任意一种作用，他也不是纯函数：

- 发起网络请求
- 刷新 UI
- 读写数据库
- 获取位置信息
- 使用蓝牙模块
- 打印输出
- ...

我们将这些作用称为函数的 **附加作用**（副作用）。

而 **纯函数** 的定义就是：**没有 附加作用 的函数，并且在参数相同时，返回值也一定相同。**

因此在已知执行逻辑时，**纯函数** 所产生的结果是可以被预测的。一些现代化的库都利用了这个特性来做状态管理，如：[RxFeedback](#), [Redux](#), [ReactorKit](#) 等等。

纯函数用于状态管理

我们用一个足够简单的例子来演示，如何用 **纯函数** 做状态管理：

```
typealias State = Int

enum Event {
```

```

        case increase
        case decrease
    }

func reduce(_ state: State, event: Event) -> State {
    switch event {
    case .increase:
        return state + 1
    case .decrease:
        return state - 1
    }
}

```

这个例子似乎过于简单，以至于我们看不出他有什么特别的。好吧，我承认他的主要目的是向大家演示，用 [纯函数](#) 做状态管理的基本单元是什么。

首先，我们得有个状态：

```
typealias State = Int
```

然后，我们要有各种事件：

```

enum Event {
    case increase
    case decrease
}

```

最后，我们要有一个 [纯函数](#) 来管理我们的状态：

```

func reduce(_ state: State, event: Event) -> State {
    switch event {
    case .increase:
        return state + 1
    case .decrease:
        return state - 1
    }
}

```

这样，我们就可以做测试了，当 `App` 处于某个状态时，发生了某个事件，会产生一个结果，这个结果是否符合我们的预期：

```

func testReduce() {

    let state1 = reduce(0, event: .increase)
    state1 == 1 // 结果: true

    let state2 = reduce(10, event: .decrease)
}

```

```
    state2 == 9 // 结果: true
}
```

以上两个测试都是成功的。当然这里的状态管理过于简单。而真实应用程序的状态都是非常复杂的。并且程序的行为都是很难预测的。要解决这个问题，我们要感谢 [纯函数](#)，还记得他的特征吗？

[纯函数](#) 在参数相同时，返回值也一定相同。

我们再来看下 `reduce` 方法：

```
func reduce(_ state: State, event: Event) -> State { ... }
```

我们有没有获得一点点灵感...

...

...

...

...

...

... 经过 60 秒后

希望你已经获得答案了。

当程序处于某个特定状态时，发生了某个特定事件，会产生某个**唯一的结果**。这个结果与**所处的环境无关**，不论是处于应用程序运行环境，还是在测试环境。这个结果只和**初始状态以及发生的事件有关**。因此，程序的行为是**可以被预测的**，而且程序运行时的状态更新，可以在测试环境中被模拟出来。

...

...

...

...

... 经过 60 秒后

现在，我们来看一个相对复杂的例子：

登录状态管理

```
typealias UserID = String

enum LoginError: Error, Equatable {
```

```

        case usernamePasswordMismatch
        case offline
    }

struct State: Equatable {

    var username: String
    var password: String

    var loading: Bool
    var data: UserID?
    var error: LoginError?

    enum Event {
        case onUpdateUsername(String)
        case onUpdatePassword(String)
        case onTriggerLogin
        case onLoginSuccess(UserID)
        case onLoginError(LoginError)
    }

    static func reduce(_ state: State, event: Event) -> State {
        var newState = state
        switch event {
        case .onUpdateUsername(let username):
            newState.username = username
        case .onUpdatePassword(let password):
            newState.password = password
        case .onTriggerLogin:
            newState.loading = true
            newState.data = nil
            newState.error = nil
        case .onLoginSuccess(let userId):
            newState.loading = false
            newState.data = userId
        case .onLoginError(let error):
            newState.loading = false
            newState.error = error
        }
        return newState
    }
}

```

我们重新走下流程，用 [纯函数](#) 做状态管理：

首先，我们得有个状态：

```

struct State: Equatable {

    var username: String // 输入的用户名

```

```

var password: String      // 输入的密码

var loading: Bool         // 登录中
var data: UserID?          // 登录成功
var error: LoginError?    // 登录失败

...
}

```

然后，我们要有各种事件：

```

enum Event {
    case onUpdateUsername(String)      // 更新用户名
    case onUpdatePassword(String)      // 更新密码
    case onTriggerLogin               // 触发登录
    case onLoginSuccess(UserID)        // 登录成功
    case onLoginError(LoginError)      // 登录失败
}

```

最后，我们要有一个 [纯函数](#) 来管理我们的状态：

```

static func reduce(_ state: State, event: Event) -> State {
    var newState = state
    switch event {
        case .onUpdateUsername(let username):
            newState.username = username
        case .onUpdatePassword(let password):
            newState.password = password
        case .onTriggerLogin:
            newState.loading = true
            newState.data = nil
            newState.error = nil
        case .onLoginSuccess(let userId):
            newState.loading = false
            newState.data = userId
        case .onLoginError(let error):
            newState.loading = false
            newState.error = error
    }
    return newState
}

```

现在我们可以在测试环境模拟各种事件，并且判断结果是否符合预期：

- [更新用户名事件](#)

```

func testOnUpdateUsername() {
    let state = State(

```

```

        username: '',
        password: '',
        loading: false,
        data: nil,
        error: nil
    )

let newState = State.reduce(state, event: .onUpdateUsername("beeth0ven"))

let expect = State(
    username: "beeth0ven",
    password: "",
    loading: false,
    data: nil,
    error: nil
)

newState == expect // 结果: true
}

```

- 更新密码事件

```

func testOnUpdatePassword() {

    let state = State(
        username: "beeth0ven",
        password: "",
        loading: false,
        data: nil,
        error: nil
    )

    let newState = State.reduce(state, event: .onUpdatePassword("123456"))

    let expect = State(
        username: "beeth0ven",
        password: "123456",
        loading: false,
        data: nil,
        error: nil
    )

    newState == expect // 结果: true
}

```

- 触发登录事件

```
func testOnTriggerLogin() {
```

```

let state = State(
    username: "beeth0ven",
    password: "123456",
    loading: false,
    data: nil,
    error: nil
)

let newState = State.reduce(state, event: .onTriggerLogin)

let expect = State(
    username: "beeth0ven",
    password: "123456",
    loading: true,
    data: nil,
    error: nil
)

newState == expect // 结果: true
}

```

- 登录成功事件

```

func testOnLoginSuccess() {
    let state = State(
        username: "beeth0ven",
        password: "123456",
        loading: true,
        data: nil,
        error: nil
    )

    let newState = State.reduce(state, event: .onLoginSuccess("userID007"))

    let expect = State(
        username: "beeth0ven",
        password: "123456",
        loading: false,
        data: "userID007",
        error: nil
    )

    newState == expect // 结果: true
}

```

- 登录失败事件

```

func testOnLoginError() {
}

```

```

let state = State(
    username: "beeth0ven",
    password: "123456",
    loading: true,
    data: nil,
    error: nil
)

let newState = State.reduce(state, event: .onLoginError(.usernamePasswordMismatch))

let expect = State(
    username: "beeth0ven",
    password: "123456",
    loading: false,
    data: nil,
    error: .usernamePasswordMismatch
)

newState == expect // 结果: true
}

```

这样我们可以轻易掌控程序的运行状态，以及各种状态更新。

现在，我们知道如何用 [纯函数](#) 做状态管理了。不过当前的代码形态，离投入生产环境，还存在好几个过度形态。这些过度形态有的是围绕如何引入 [附加作用](#)，而做了一些应用架构。在这个问题上，不同地架构也提出了不同的解决方案，如：[RxFeedback](#) 用 `feedbackLoop` 引入 [附加作用](#)，[Redux](#) 用 `middleware` 引入 [附加作用](#) 等等。这里就不一一介绍了，这些库的官方网站都会有相关说明。

最后，我们还是将代码演化到下一个形态，这里我选择使用 [Redux](#) 流派。因为个人的觉得他的知识依赖要少一些，可以让更多读者从中获益。

下一步 -- 引入 Store

```

class Store {

    // 观察者，用于响应状态更新，第一个 State? 为旧状态，第二个 State 为当前状态
    typealias Observer = (State?, State) -> Void

    private(set) var state: State // 当前状态
    private var _observers: [UUID: Observer] // 所有的观察者

    // 初始化
    init(initialState: State) {
        self.state = initialState
        self._observers = [:]
    }
}

```

```

// 发出事件
func dispatch(event: State.Event) {
    let oldState = self.state
    self.state = State.reduce(self.state, event: event)
    _publish(oldState: oldState, newState: self.state)
}

// 订阅状态更新
func subscribe(observer: @escaping Observer) -> UUID {
    let subscriptionID = UUID() // UUID 是唯一标识符，该 id 可用于取消订阅
    _observers[subscriptionID] = observer
    observer(nil, self.state) // 订阅时，将当前状态回放给该观察者
    return subscriptionID
}

// 取消订阅
func unsubscribe(_ subscriptionID: UUID) {
    _observers.removeValue(forKey: subscriptionID)
}

// 私有方法，通知所有的观察者，状态已经更新了
private func _publish(oldState: State?, newState: State) {
    _observers.values.forEach { observer in
        observer(oldState, newState)
    }
}
}

```

如何使用 `Store` :

```

func useStore() {

let initailState = State(
    username: "",
    password: "",
    loading: false,
    data: nil,
    error: nil
)

let store = Store(initailState: initailState)

// 以下变量 newStates 和 oldStates 用于录制状态历史
var newStates: [State] = []
var oldStates: [State?] = []

let subscriptionID = store.subscribe { (oldState, newState) in
    newStates.append(newState)
    oldStates.append(oldState)
}
}

```

```
// 模拟真实事件
store.dispatch(event: .onUpdateUsername("beeth0ven"))
store.dispatch(event: .onUpdatePassword("123456"))

// 取消订阅
store.unsubscribe(subscriptionID)

// 描述预期
let expectNewStates = [
    State(
        username: "", 
        password: "", 
        loading: false, 
        data: nil, 
        error: nil
    ),
    State(
        username: "beeth0ven", 
        password: "", 
        loading: false, 
        data: nil, 
        error: nil
    ),
    State(
        username: "beeth0ven", 
        password: "123456", 
        loading: false, 
        data: nil, 
        error: nil
    )
]

let expectOldStates = [
    nil,
    State(
        username: "", 
        password: "", 
        loading: false, 
        data: nil, 
        error: nil
    ),
    State(
        username: "beeth0ven", 
        password: "", 
        loading: false, 
        data: nil, 
        error: nil
    )
]
```

```
// 比对结果
newStates == expectNewStates // 结果: true
oldStates == expectOldStates // 结果: true
}
```

以上是在单元测试环境下，

首先下创建 Store：

```
let initailState = State(
  username: "",
  password: "",
  loading: false,
  data: nil,
  error: nil
)

let store = Store(initailState: initailState)
```

然后，订阅程序状态，并且将这些状态录制下来：

```
var newStates: [State] = []
var oldStates: [State?] = []

let subscriptionID = store.subscribe { (oldState, newState) in
  newStates.append(newState)
  oldStates.append(oldState)
}
```

然后，模拟输入用户名事件和输入密码事件：

```
store.dispatch(event: .onUpdateUsername("beeth0ven"))
store.dispatch(event: .onUpdatePassword("123456"))
```

然后，取消订阅：

```
store.unsubscribe(subscriptionID)
```

最后，比对录制的状态是否符合预期：

```
let expectNewStates = [
  State(
    username: "",
    password: "",
    loading: false,
    data: nil,
    error: nil
  )
]
```

```

),
State(
    username: "beeth0ven",
    password: "",
    loading: false,
    data: nil,
    error: nil
),
State(
    username: "beeth0ven",
    password: "123456",
    loading: false,
    data: nil,
    error: nil
)
]

let expectOldStates = [
nil,
State(
    username: "",
    password: "",
    loading: false,
    data: nil,
    error: nil
),
State(
    username: "beeth0ven",
    password: "",
    loading: false,
    data: nil,
    error: nil
)
]

newStates == expectNewStates // 结果: true
oldStates == expectOldStates // 结果: true

```

这就是如何在测试环境里面使用 `Store`，那么在 `App` 里面如何使用 `Store` 呢。一个 **相对简单（并未优化）** 的方法，就是将 `Store` 注入到对应的组件里面，这里以 `ViewController` 为例：

- `ViewController` 可以使用 `store.subscribe` 方法订阅程序的状态。当状态更新时，比对新旧状态，然后刷新过时了的 UI。
- 当用户触发某个事件时，调用 `store.dispatch` 方法将事件发出去，如：当用户点击登录按钮时，就调用 `store.dispatch(event: .onTriggerLogin)`。
- 在 `ViewController` 的 `deinit` 方法里面注销订阅 `store.unsubscribe(subscriptionID)`。

总结

本节主要介绍了 [纯函数](#) 和 [附加作用](#)，期间还演示如何用 [纯函数](#) 做状态管理的。最后还演化出了一个极简版的 [Redux](#)。希望大家可以从中获益！

参考

- [纯函数](#)
- [附加作用](#)
- [RxFeedback](#)
- [Redux](#)
- [ReactorKit](#)

附加作用（副作用）

什么是函数的附加作用？

如果一个函数除了计算返回值以外，还有其他可观测作用，我们就称这个函数拥有[附加作用](#)。

哪些是函数的附加作用？

网络请求

如果一个函数发起了网络请求，那他就是有[附加作用](#)的。这个[附加作用](#)是获取或写入了函数本体以外的全局状态（数据库存储的状态可看作是全局状态）。

获取位置信息

如果一个函数获取了位置信息，那他就是有[附加作用](#)的。这个[附加作用](#)是获取了函数本体以外的位置信息（也可以看作是全局状态）。

获取 UI 状态

如果一个函数获取了 UI 状态，那他就是有[附加作用](#)的。这个[附加作用](#)是读取函数本体以外的 UI 状态（也可以看作是全局状态）。

其他类型的附加作用

以上，网络请求，定位和UI是比较常见的[附加作用](#)。以下也是[附加作用](#)：

- 读写全局变量
- 读写本地数据库
- 读写文件
- 使用蓝牙模块
- 打印输出
- ...

App 的附加作用

有[附加作用](#)并不是什么坏事情。事实上，正是因为有了他，App 才更有价值。我们以几个常见 App 为例：

饿了么

饿了么是一个订餐 App，他最主要的 [附加作用](#) 是更新程序本体以外的状态。即：饿 -> 饱（将我们从很饿变为很饱）。

抖音

抖音是一个影音娱乐 App，他最主要的 [附加作用](#) 是更新程序本体以外的状态。即： -> （将我们变得更开心）。

滴滴

滴滴是一个叫车 App，他最主要的 [附加作用](#) 是更新程序本体以外的状态。即： -> （从起点到终点）。

Observable 中的 附加作用

在解释 Observable 中的 [附加作用](#) 之前，我们先要理解一个概念，即： Observable 其实是一个函数：

```
// 去除了不相关的范型约束，便于理解
func subscribe(_ observer: Observer) -> Disposable
```

你没有看错！以上 `subscribe` 函数就是 Observable。

换句话说 Observable 的 [附加作用](#)，指的就是 `subscribe` 函数里面的 [附加作用](#)。

...

...

...

...

...

... 经过 60 秒后

示例：

之前在介绍 Observable 时，举了这样一个例子：



```
typealias JSON = Any
```

```

let json: Observable<JSON> = Observable.create { (observer) -> Disposable in

    let task = URLSession.shared.dataTask(with: ...) { data, _, error in

        guard error == nil else {
            observer.onError(error!)
            return
        }

        guard let data = data,
              let jsonObject = try? JSONSerialization.jsonObject(with: data, options:
.mutableLeaves)
        else {
            observer.onError(DataError.cantParseJSON)
            return
        }

        observer.onNext(jsonObject)
        observer.onCompleted()
    }

    task.resume()

    return Disposables.create { task.cancel() }
}

```

这里的闭包 `{ (observer) -> Disposable in ... }` 可以看作是 `subscribe` 函数，这个函数的 **附加作用** 就是发起网络请求去获取一个 `JSON`。所以 `let json: Observable<JSON>` 的 **附加作用** 也是发起网络请求去获取一个 `JSON`。

现在我们应该能够理解，什么是 `Observable` 的 **附加作用** 了。

为什么我喜欢称它为 **附加作用**，而不是“**副作用**”？

首先澄清一下，我们这里所介绍的 **附加作用**，就是大家平时说的“**副作用**”。

最近听音乐时，不经意间切到了这一首歌：《爱的副作用》（这首歌可能你也听过）。

于是我就很好奇，这个“**副作用**”到底指的是 **不好的作用**，还是**附加作用**。从标题上看不出来，后来我看了下歌词：

```

...
但是我还是想不透
后来的我害怕什么
难道爱也有副作用
藏在血液里头
让我的心偶尔有点痛
...

```

我知道了，这里的“副作用”应该是**不好的作用**。虽然解释成**附加作用**也说得通。但是**不好的作用**更符合语境。

那么这里就有个问题，“副作用”是一个**多义词**。他既可以代表**不好的作用**，也可以代表**附加作用**。而且，在有些语境下这两种意义都说得通。我们便无法读出作者的本意。

在计算机领域也是一样的，虽然很多时候我们都应该知道“副作用”指的是**附加作用**，但是用**不好的作用**也解释得通。这样就会产生歧义。

所以，我觉得如果有一个词，专门表示计算机领域的“副作用”会更好。如：**附加作用**。如此一来，读者不需要做多余的判断，就能解读作者的意图。

共享 附加作用

文档中一些特征序列，会有如下特性：

共享 附加作用：

- Driver
- Signal
- ControlEvent
- ...

不共享 附加作用：

- Single
- Completable
- Maybe
- ...

那什么是共享 附加作用，什么是不共享 附加作用？

共享 附加作用：

```
...
let observable: Observable<Teacher> = API.teacher(teacherId: 1)
let shareSideEffects: Driver<Teacher> = observable.asDriver(onErrorDriveWith: .empty())

let observer0: (Teacher) -> () = ...
let observer1: (Teacher) -> () = ...

shareSideEffects.drive(onNext: observer0)
shareSideEffects.drive(onNext: observer1) // 第二次订阅
```

如果一个序列共享 附加作用，那在第二次订阅时，不会重新发起网络请求，而是共享第一次网络请求（附加作用）。

不共享 附加作用：

```
...
let observable: Observable<Teacher> = API.teacher(teacherId: 1)
let notShareSideEffects: Single<Teacher> = observable.asSingle()

let observer0: (Teacher) -> () = ...
let observer1: (Teacher) -> () = ...

notShareSideEffects.subscribe(onSuccess: observer0)
```

```
notShareSideEffects.subscribe(onSuccess: observer1) // 第二次订阅
```

如果一个序列不共享 [附加作用](#)，那在第二次订阅时，会重新发起网络请求，而不是共享第一次网络请求（[附加作用](#)）。

因此我们需要注意，如果一个网络请求序列，他不共享 [附加作用](#)，那每一次订阅时就会单独发起网络请求。这时最好改用 [共享附加作用](#) 的序列，或者使用 `share` 操作符。