Combine实用教程

前言

在现代 GUI 编程中，开发者会处理大量事件（包括网络，屏幕输入，系统通知等），根据事件去让界面发生变化。而对异步事件的处理，会让代码和状态变得尤其复杂。为了帮助开发者简化异步编程，使代码更加简洁、易于维护，WWDC 2019 发布了基于 Swift 的响应式异步编程框架 — Combine。

教程内容

本教程将从 Combine 概念入手，逐步学习 Combine 的知识点，主要内容：

- Concept
- Publisher
- Subscriber
- Subscription
- Cancellable
- Subject
- Operator
- Type Erasure
- Publisher in Foundation
- Common Operators
- Scheduler
- Future
- Practice

作者

- YungFan，杨帆，高校教师，开发者。
- Email：yungfan@vip.163.com。
- GitHub：https://github.com/yungfan。
- 在线课程：https://ke.qq.com/cgi-bin/agency?aid=67223。
- 微信公众号：YungFan。
- 博客：**掘金**、**腾讯云·云社区** 和 **简书**。

勘误

各位读者在阅读本教程时，如果对其中的内容有疑义或者修改建议，欢迎联系作者。

版权

更新与修订

时间	章节	更新/修订内容
2020.08	**Publisher in Foundation**	@Published—增加协议中使用
2020.09	**Subscriber**	增加iOS 14中Assign支持绑定@Published属性
2021.03	**Publisher in Foundation**	Timer Publisher—增加SwiftUI中使用
2021.04	**Future**	案例
2021.05	**Publisher**	内置Publisher—修改相关描述
	Appendix I	
2021.06	**Publisher in Foundation**	Timer Publisher—SwiftUI中使用增加案例二
2021.09	**所有章节**	采用新的渲染样式

Concept

观察者模式

观察者模式是设计模式的一种，在软件开发中经常会用到，iOS 开发中的`KVO`、`通知`等都用到了观察者模式。观察者模式中有两个角色，一个是被观察者，一个是观察者。比如一个宝宝在睡觉，爸爸妈妈不能在一直守候在身边，他们自己处理各自的事情，但是一旦听到宝宝的哭声，他们就去看宝宝。这就是一个典型的观察者模式。宝宝是被观察者（也称为发布者），爸爸妈妈是观察者（也称为订阅者），只要被观察者发出了某些事件比如宝宝哭声就是一个事件，通知到观察者，观察者就可以做相应的处理工作。

响应式编程

响应式编程（Reactive Programming）是**面向异步数据流的编程思想**。一个**事件**及其对应的**数据**被**发布**出来，最后被**订阅者**消化和使用。期间这些事件和数据需要通过一系列**操作变形**，成为我们最终需要的事件和数据。业界比较知名的响应式编程框架是 ReactiveX 系列，Rx 也有 Swift 版本 — `RxSwift`。

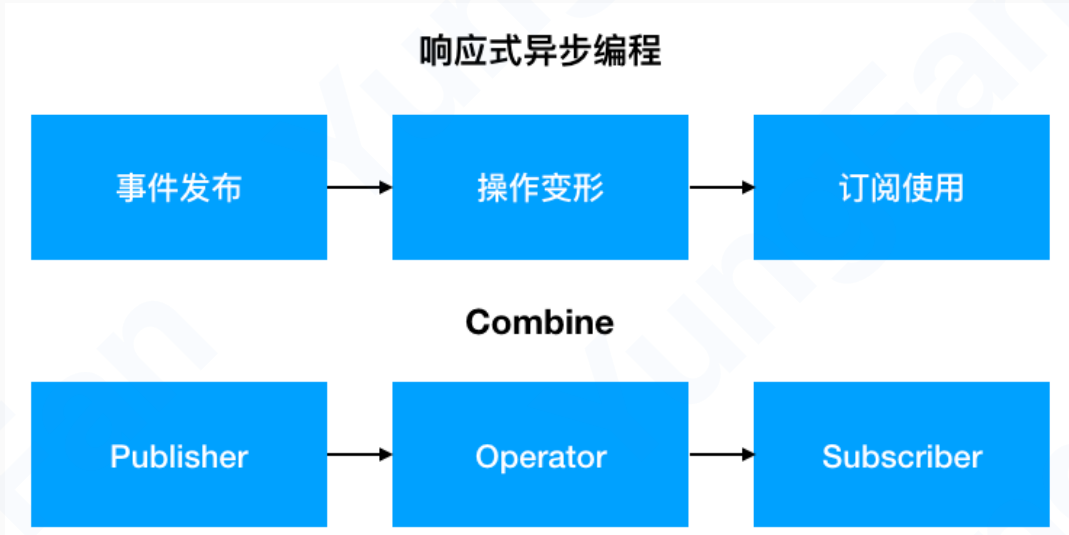
观察者模式与响应式编程

观察者模式提供了一种思想，即**发布-订阅**思想。从中可以衍生出很多编程模型，比如发布者-订阅者模型、事件-事件源-监听器模型、被观察者-发射器-订阅者模型，这些模型应该都属于响应式编程的范畴。甚至可以说只要是基于观察者模式实现的编程思维，都属于响应式编程。

Combine简介

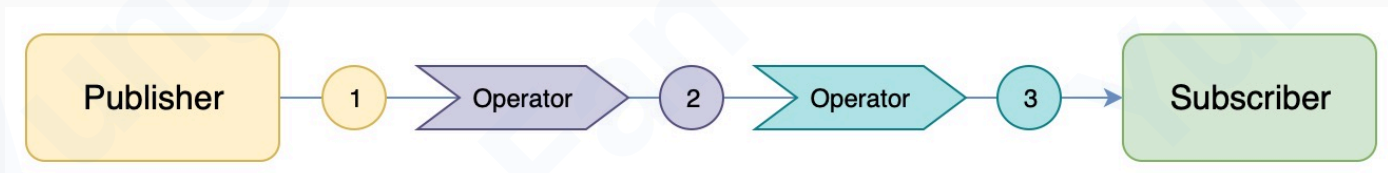
Combine 是 Apple 在 Swift 语言层面实现的响应式异步编程框架。官方介绍：Combine 提供了一个声明式的 Swift API，可以用来处理开发中常见的**Target/Action、Notification、KVO、callback/closure 以及各种异步网络请求**。它可以使代码更加简洁、易于维护，也免除了饱受诟病的嵌套闭包和回调地狱。

核心概念



- **Publisher**（发布者）：负责发布数据。
- **Subscriber**（订阅者）：负责订阅数据。
- **Operator**（操作符）：负责在`Publisher`和`Subscriber`之间数据的转换。

Combine最简单地可以描述为：



或者说

`Combine = Publishers + Subscribers + Operators`

特点

1. `基于泛型`：支持泛型。
2. `类型安全`：Swift 会检查类型安全问题。
3. `组合优先`：Apple 建议使用多个自定义 Publisher 将小部分的事情组合起来。
4. `请求驱动`（Request Driven）：基于请求和响应的设计思想，Subscriber 向 Publisher 发出请求，Publisher 向 Subscriber 给予响应。

学习要求

1. 熟悉 Swift 语法。
2. 熟悉基于 UIKit 或 SwiftUI 的 iOS 开发知识。
3. 开发环境：macOS 10.15 及以上。
4. 开发工具：Xcode 11 及以上。
5. 运行环境：iOS 13 / iPadOS 13 / macOS 10.15 / tvOS 13 / watchOS 6 及以上。

Publisher

介绍

- `Publisher` 的主要工作是随着时间推移向一个或多个 `Subscriber` 发布数据和事件。
- `Publisher` 最主要的工作其实有两个：
  - 被 `Subscriber` 订阅。
  - 发布数据和事件。

定义

Combine 中包括 Publisher 在内的一系列角色都使用协议来进行定义。

```
public protocol Publisher {
    /// 发布数据的类型
    associatedtype Output

    /// 失败的错误类型
    associatedtype Failure: Error

    /// Subscriber不会主动调用该方法，而是在调用subscribe()方法时内部会调用此方法
    func receive<S>(subscriber: S) where S: Subscriber, Self.Failure == S.Failure, Self.Output == S.Input
}

extension Publisher {
    /// 将指定的Subscriber订阅到此Publisher
    /// 供外部调用，不直接使用receive(subscriber:)
    public func subscribe<S>(_ subscriber: S) where S: Subscriber, Self.Failure == S.Failure, Self.Output == S.Input
}
```

1. `Output` 及 `Failure` 定义了 Publisher 所发布的数据的类型和失败的错误类型。如果不会失败，则 `Failure` 使用 `Never` 。
2. Publisher 只能发布一个结束事件，一旦发出了其生命周期就结束了，不能再发出任何数据和事件。
3. Subscriber 调用 `subscribe()` 方法订阅 Publisher 时会调用 `receive()` 方法。 它规定：Publisher 的 `Output` 必须与 Subscriber 的 `Input` 类型匹配， `Failure` 也是如此。

内置Publisher

- `Just`： 只提供一个数据便终止的 Publisher， 失败类型为 `Never` 。（★）
- `Sequence`： 根据指定数据序列（数组、区间等）创建的 Publisher 。（★）
- `Future`： 异步操作的 Publisher， 用一个闭包初始化。（★）
- `Deferred`： 在运行提供的闭包之前等待订阅的 Publisher。（★）
- `Share`： 它是一个类而非结构体， 可以与多个 Subscriber 共享同一个 Publisher。
- `Multicast`： 可以让多个 Subscriber 订阅同一个 Publisher 时产生相同的订阅效果。
- `Empty`： 一个不发布数据的 Publisher。
- `Fail`： 由于指定 Error 而立即终止的 Publisher 。
- `Record`： 允许记录一系列 `Input` 和 `Completion`， 供 Subscriber 订阅使用。
- `Optional`： 如果可选型有值，则 Publisher 向 Subscriber 发布一次该可选数据，如果为 nil 则不发布任何数据。
- `ObservableObject`： 与 SwiftUI 一起使用，符合 `ObservableObject` 协议的对象即可作为 Publisher 。（★）
- `@Published`： 属性包装器，用来将一个属性数据转变为 Publisher 。（★）

自定义

一般情况下，不需要自定义 Publisher， Apple 也不推荐这样做。 大部分情况下，内置的 Publisher 和 Subject（后面讲解） 已经足够使用。

Subscriber

介绍

Publisher 根据 `Subscriber` 的请求提供数据。如果没有任何订阅请求，Publisher 不会主动发布任何数据。所以可以这样说，`Subscriber` 负责向 Publisher 请求数据并接收数据（或失败）。

定义

```
public protocol Subscriber: CustomCombineIdentifierConvertible {
    /// 接收数据的类型
    associatedtype Input

    /// 可能接收到的失败的错误类型
    associatedtype Failure: Error

    /// 接收到订阅的消息，完成订阅可以开始请求数据了
    func receive(subscription: Subscription)

    /// 接收到产生的值的消息，Publisher发布了新的数据
    func receive(_ input: Self.Input) -> Subscribers.Demand

    /// 接收到产生已经终止的消息，Publisher发布了完成事件
    func receive(completion: Subscribers.Completion<Self.Failure>)
}
```

其中 `Input` 和 `Failure` 分别表示了 Subscriber 能够接收的数据类型和失败的错误类型。如果不会接收失败，则 `Failure` 使用 `Never` 。

内置Subscriber

Sink

在闭包中处理数据或 completion 事件。每当收到新值时，就会调用 `receiveValue` 。还有一个可选的 `receiveCompletion` ，当接收完所有的值之后调用。

```
import Combine

// Just发送单个数据
let publisher = Just(1)

// sink订阅
let subscription = publisher.sink(receiveCompletion: { _ in
    print("receiveCompletion")
}, receiveValue: { value in
    print(value)
})

/* 输出
1
receiveCompletion
*/
```

Assign

- 将 Publisher 的 Output 数据设置到**类**中的**属性**。
- 参数：某个类对象和该对象上的某个属性 **KeyPath**。
- 应用：可以直接把发布的值绑定到数据模型或者 UI 控件的属性上。

```
import Combine

// 创建对象
class Student {
    var name: String = ""
}

let stu = Student()

print(stu.name)

// Just发送单个数据
let publisher = Just("ZhangSan")

// assign订阅，设置到Student的name属性上
publisher.assign(to: \.name, on: stu)

print(stu.name)

/* 输出

ZhangSan
*/
```

- **iOS 14 之后可以直接绑定到 `@Publisher` 属性上。**

```
import Combine

// 创建对象
class Student {
    @Published var name: String = ""
}

let stu = Student()

print(stu.name)

// Just发送单个数据
let publisher = Just("Lisi")

// assign订阅，设置到Student的@Published属性上
publisher.assign(to: &stu.$name)

print(stu.name)

/* 输出

Lisi
*/
```

自定义

动手实现一个 Subscriber，会对 Publisher 和 Subscriber 之间的关系更加明晰。

```
import Combine

// 1. 通过数组创建一个Publisher
let publisher = [1, 2, 3, 4, 5, 6].publisher

// 2. 自定义一个Subscriber
class CustomSubscriber: Subscriber {
    // 3. 指定接收值的类型和失败类型
    typealias Input = Int
    typealias Failure = Never

    // 4. Publisher首先会调用该方法
    func receive(subscription: Subscription) {
        // 接收订阅的值的最大量，通过.max()设置最大值，还可以是.unlimited
```

```
        subscription.request(.max(6))
    }

    // 5. 接收到值时的方法，返回接收值的最大个数变化
    func receive(_ input: Int) -> Subscribers.Demand {
        // 输出接收到的值
        print("Received value", input)
        // 返回.none，意思就是不改变最大接收数量（永远为上面方法设置的大小，如果上面设置的最大值小于Publisher发送的数据，不会走completion），也可以通过.max()设置增大多少
        return .none
    }

    // 6. 实现接收到完成事件的方法
    func receive(completion: Subscribers.Completion<Never>) {
        print("Received completion", completion)
    }
}

// 订阅Publisher
publisher.subscribe(CustomSubscriber())

/*输出
Received value 1
Received value 2
Received value 3
Received value 4
Received value 5
Received value 6
Received completion finished
*/
```

Subscription

订阅流程

从宏观角度看，Combine 的订阅流程如下：

1. Subscriber 调用 Publisher 的 `subscribe(_ subscriber:)` 方法开始订阅。
2. Publisher 调用 Subscriber 的 `receive(subscription:)` 发送确认信息给 Subscriber。该方法接收一个 Subscription。
3. Subscriber 调用 2 中创建的 Subscription 上的 `request(_: Demand)` 方法**首次**告诉 Publisher 需要的数据及其最大值。
4. Publisher 调用 Subscriber 的 `receive(_: Input)` 发送不超过第 3 步 `Demand` 指定个数的数据给 Subscriber，并返回一个新的 `Demand`，告诉 Publisher 下次发送的最大数据量。
5. 同4
6. Publisher 调用 Subscriber 的 `receive(completion :)` 向 Subscriber 发送 completion 事件。这里的 completion 可以是正常 `.finished`，也可以是 `.failure` 的，如果是 `.failure` 的会携带一个错误信息。注意：如果中途取消了订阅，Publisher 将不发送完成事件。

介绍

当 Publisher 发布新值时，`Subscription` 负责协调 Publisher 和 Subscriber。在某种程度上可以说 Publisher 只负责发布数据，订阅流程的大部分工作是由 Subscriber 和 `Subscription` 完成的。`Subscription` 定义如下：

```
public protocol Subscription: Cancellable, CustomCombineIdentifierConvertible {
    ///  告诉 Publisher 可以发送多少个数据到 Subscriber
    func request(_ demand: Subscribers.Demand)
}
```

1. Subscriber 调用 Publisher 的 `subscribe(_ subscriber:)` 方法开始订阅。
2. Publisher 会调用 `receive(subscriber:)`，在该方法中创建 Subscription 对象并调用 Subscriber 的 `receive(subscription: Subscription)` 方法传递给 Subscriber。
3. Subscriber 调用 Subscription 的 `request(_ demand:)` 方法**首次**告诉 Subscription 需要的数据及其最大值。
4. Subscription 调用 Subscriber 的 `receive(_ input:)` 方法发送数据给 Subscriber，并且返回一个 `Subscribers.Demand`，告诉 Subscription 下次需要的最大数据量。
5. 当最后一次值发布完毕，Subscription 会调用一次 Subscriber 的 `receive(completion:)` 结束订阅流程。

Back pressure

Combine 约定 Subscriber 控制数据流，用于解决 Publisher 发布数据过多过快的问题，要确保 Subscriber 收到的数据不会超过它请求的数据量。这个特性称之为 **Back pressure**。Subscription 中 `request(_: Demand)` 方法的返回值 `Subscribers.Demand` 就是用于告诉 Publisher，此次发布数据的最大值，这个值是累加的。`Subscribers.Demand` 常见的取值有 `Demand.unlimited`、`Demand.none` 和 `Demand.max(Int)`。之前介绍的 `sink` 在调用 Subscription 的 `request(_: Demand)` 方法时，Back pressure 的值就是 `Demand.unlimited`。

Subject

介绍

`Subject` 是一种特殊的 Publisher，最大的特点是可以**手动发送数据**。定义如下：

```
public protocol Subject: AnyObject, Publisher {
    func send(_ value: Self.Output)
    func send(completion: Subscribers.Completion<Self.Failure>)
    func send(subscription: Subscription)
}
```

从定义可以看到，Subject 暴露了 3 个 `send()` 方法，可以通过 `send()` 方法来**手动发布** `Output` 数据以及 `Completion` 事件。

内置Subject

PassthroughSubject

通过 send 发送数据或事件给下游的 Publisher 或 Subscriber， 并**不会对接收到的数据进行保留**。

```
import Combine

// 创建PassthroughSubject
let subject = PassthroughSubject<String, Never>()

// 订阅
let subscription = subject.sink(receiveCompletion: { _ in
    print("receiveCompletion")
}, receiveValue: { value in
    print(value)
})

// 发送数据
subject.send("Hello")
subject.send("Combine")
subject.send(completion: .finished)

/* 输出
Hello
Combine
receiveCompletion
*/
```

CurrentValueSubject

- 与 PassthroughSubject 不同的是它会**保留一个最后的数据**，并在被订阅时将这个数据发送给下游的 Publisher 或 Subscriber。
- CurrentValueSubject 初始化时需要提供一个当前值，并可以通过其 value 属性设置和获取当前值。

```
import Combine

// 创建CurrentValueSubject，需要初始化一个数据
let subject = CurrentValueSubject<String, Never>("Hello")
// 获取当前值
print(subject.value)

// 发送数据
subject.send("Combine")
print(subject.value)
```

```swift
// 发送数据
subject.send("SwiftUI")
print(subject.value)

// 订阅
let subscription = subject.sink { value in
    print(value)
}

/* 输出
Hello
Combine
SwiftUI
SwiftUI
*/
```

Cancellable

介绍

在开发中，当 Subscriber 不想接收 Publisher 发布的数据时，可以取消订阅以释放资源。Combine 中提供了一个 `Cancellable` 协议，该协议中定义了一个 `cancel()` 方法，用于取消订阅流程。定义如下：

```
protocol Cancellable {
    func cancel()
}
```

AnyCancellable

Combine 中还定义了一个 `AnyCancellable` 类，它实现了 Cancellable 协议，特点是会在 `deinit` 时自动执行 `cancel()` 方法。定义如下：

```
final public class AnyCancellable : Cancellable, Hashable {
}

extension AnyCancellable {
    /// 将此AnyCancellable存储在指定的集合中。
    final public func store(in set: inout Set<AnyCancellable>)
}
```

- 前面介绍的 `sink` 和 `assign` 的返回值都是 `AnyCancellable`，所以它们可以调用 `cancel()` 方法来取消订阅。
- 当 `AnyCancellable` 所在类执行 `deinit` 时， `AnyCancellable` 的 `deinit` 也会被触发，并自动释放资源。
- 案例

```
// 创建PassthroughSubject
let subject = PassthroughSubject<String, Never>()

// 订阅
let subscription = subject.sink(receiveCompletion: { _ in
    print("receiveCompletion")
}, receiveValue: { value in
    print(value)
})

// 发送数据
subject.send("Hello")
// 中途取消
subscription.cancel()
// 后续发送都会失败
subject.send("Combine")
subject.send(completion: .finished)

/* 输出
Hello
*/
```

注意

当 `AnyCancellable` 对象被释放后，整个订阅流程也会随之结束。所以在实际开发中需要把这个 `AnyCancellable` 对象当做一个属性存储起来或者存储到 `Set<AnyCancellable>` 中。

应用

模拟网络原因导致的网络请求中断

```
import UIKit
import Combine

let dataPublisher = URLSession.shared.dataTaskPublisher(for: URL(string: "https://www.baidu.com")!)

let cancellableSink = dataPublisher
    .sink(receiveCompletion: { completion in
        switch completion {
        case .finished:
            print("received finished")
            break
        case .failure(let error):
            print("received error: ", error)
        }}, receiveValue: { someValue in
            print("received \(someValue)")
    })

// 可以取消
cancellableSink.cancel()
```

模拟用户取消上传数据

```
import UIKit
import Combine

let request = URLRequest(url: URL(string: "https://xxxxx")!)
let image = UIImage(named: "largeImage")
let imgFile: Data = image!.pngData()!

// 上传Publisher
let downloadPublisher = Future<Data?, Never> { promise in
    URLSession.shared.uploadTask(with: request, from: imgFile) { (data, _, _) in
        promise(.success(data))
    }.resume()
}

// 订阅
let subscription = downloadPublisher.sink { data in
    print("Received data: \(data)")
}

// 可以在完成之前调用cancel取消任务
subscription.cancel()
```

Operator

介绍

默认情况下，订阅某个 Publisher，Subscriber 中的 `Input` 和 `Failure` 要与 Publisher 的 `Output` 和 `Failure` 类型相同，但实际开发中往往是不同的，此时就需要借助 `Operator` 进行**转换。Operator 遵守 Publisher 协议，负责从数据流上游的 Publisher 订阅值，经过转换生成新的 Publisher 发送给下游的 Subscriber。**

> **总结**：Publisher，Operator 和 Subscriber 三者组成了数据流从发布，转换，到订阅的完整链条。

简单案例

Publisher 发布的值为 `Int` 类型的 `520`，最后订阅以后输出 `String` 类型的值 `I Love You`。中间通过 `map` 这个 Operator 进行转换。

```
import Combine

let subscription = Just(520)
    .map { value -> String in
        return "I Love You"
}.sink { receivedValue in
    print("最终的结果: \(receivedValue)")
}

/* 输出
最终的结果: I Love You
*/
```

内置Operator

Operator 非常多，其中很多与 Swift 标准库的函数非常像，比如 `map`，`fliter` 等。Operator 可以通过链式方式进行调用，在后面的案例中会进行讲解。下面按照功能对 Operator 进行了简单的分类。

转换

```
scan
tryScan
setFailureType
map
tryMap
flatMap
```

过滤

```
compactMap
tryCompactMap
replaceNil
replaceEmpty
filter
tryFilter
replaceError
removeDuplicates
tryRemoveDuplicates
```

合规

```
collect
reduce
tryReduce
ignoreOutput
```

数学运算

```
max
tryMax
count
min
tryMin
```

匹配

```
allSatisfy
tryAllSatisfy
contains
containsWhere
tryContainsWhere
```

序列

```
firstWhere
tryFirstWhere
first
lastWhere
tryLastWhere
last
dropWhile
tryDropWhile
dropUntilOutput
prepend
drop
prefixUntilOutput
prefixWhile
tryPrefixWhile
output
```

组合

```
combineLatest
merge
zip
```

异常处理

```
catch
```

```
tryCatch
assertNoFailure
retry
mapError
```

调整类型

```
switchToLatest
eraseToAnyPublisher
```

时间控制

```
debounce
delay
measureInterval
throttle
timeout
```

编解码

```
encode
decode
```

资源管理

```
shared
multicast
```

调试

```
breakpoint
handleEvents
print
```

Type Erasure

介绍

- Publisher 中的 Output 和 Failure 两个关联类型如果进行多次嵌套会让类型变得非常复杂，难以阅读，而实际开发中往往需要经过多次的操作才能得到合适的 Publisher。
- 对于 Subscriber 来说，只需要关心 Publisher 的 Output 和 Failure 两个类型就能顺利订阅，它并不需要具体知道这个 Publisher 是如何得到、如何嵌套的。
- 为了对复杂类型的 Publisher 进行**类型擦除**，Combine 提供了 `eraseToAnyPublisher()` 方法将复杂的 Publisher 转化为对应的**通用类型** `AnyPublisher` 。
- 类型擦除后的 Publisher 变得简单明了易于理解，在实际开发中经常使用。

案例

- 案例一

```
import Combine

// p1类型: Publishers.FlatMap<Publishers.Sequence<[Int], Never>, Publishers.Sequence<[[Int]], Never>>
let p1 = [[1, 2, 3], [4, 5, 6]]
    .publisher
    .flatMap { $0.publisher }

// p2类型: Publishers.Map<Publishers.FlatMap<Publishers.Sequence<[Int], Never>, Publishers.Sequence<[[Int]], Never>>, Int>
let p2 = p1.map { $0 * 2 }

// p3类型: AnyPublisher<Int, Never>
let p3 = p2.eraseToAnyPublisher()
```

- 案例二

```
import Combine

let subject = PassthroughSubject<Int, Never>()

// PassthroughSubject转变成了AnyPublisher类型
let publisher = subject.eraseToAnyPublisher()

// 订阅
let subscription = publisher
    .sink(receiveValue: { print($0) })

subject.send(10)
subject.send(100)

// AnyPublisher没有send()方法
//publisher.send(10)
```

Publisher in Foundation

介绍

为了方便使用，Foundation 对一些常用的操作提供了基于 Publisher 的 API，可以直接在开发中使用，主要有以下几种：

- Sequence Publisher
- URLSession Publisher
- Notification Publisher
- KVO Publisher
- Timer Publisher
- @Published

Sequence Publisher

通过序列构造 Publisher，如数组，区间和字典等。

```
import Combine

// 数组
["a", "b", "c"].publisher // Combine.Publishers.Sequence<Array<String>, Never>

// 区间
(1...10).publisher // Combine.Publishers.Sequence<ClosedRange<Int>, Never>

// stride
stride(from: 0, to: 10, by: 2).publisher // Combine.Publishers.Sequence<StrideTo<Int>, Never>

// 字典
["name" : "zhangsan", "age" : "15"].publisher // Combine.Publishers.Sequence<Dictionary<String, String>, Never>
```

URLSession Publisher

这是 URLSession 在 iOS 13 之后新增的一种网络 API，通过这个 API 可以更加简单的完成网络请求、数据转换等操作。

```
let url = URL(string: "https://www.baidu.com")

// 创建Publisher
let publisher = URLSession.shared.dataTaskPublisher(for: url!)

// 订阅
let subscripton = publisher.sink(receiveCompletion: { print($0)
}) { (data, response) in
    print(String(data: data, encoding: .utf8)!)
}
```

- 应用。

```
import Combine
import UIKit

// 服务器返回的数据对应的Model
struct NewsModel: Codable {
```

```
    var reason: String
    var error_code: Int
    var result: Result
}

struct Result: Codable {
    var stat: String
    var data: [DataItem]
}

// 实现Hashable, List中的数据必须实现
struct DataItem: Codable, Hashable {
    var title: String
    var date: String
    var category: String
    var author_name: String
    var url: String
}
```

```
let url = URL(string: "http://v.juhe.cn/toutiao/index?type=top")
let request = URLRequest(url: url!)
let session = URLSession.shared
let backgroundQueue = DispatchQueue.global()

let dataPublisher = session.dataTaskPublisher(for: request)
    .retry(5)
    .timeout(5, scheduler: backgroundQueue)
    .map{$0.data}
    .decode(type: NewsModel.self, decoder: JSONDecoder())
    .subscribe(on: backgroundQueue)
    .eraseToAnyPublisher()

let subscription = dataPublisher.receive(on: DispatchQueue.main)
    .sink(receiveCompletion: {_ in }) {
        newsModel in
        print(newsModel.result.data)
}
```

Notification Publisher

Notification 在 iOS 13 之后也提供了创建 Publisher 的辅助 API。

- 系统通知

```
import UIKit
import Combine

let subscription = NotificationCenter.default.publisher(for: UIApplication.willResignActiveNotification)
    .sink(receiveValue: { _ in
        print("App进入后台")
    })
```

- 自定义通知

```
import UIKit
import Combine

// 自定义通知名
extension Notification.Name{
    static var myNotiName = Notification.Name("YungFan")
}

// 订阅通知
let subscription = NotificationCenter.default.publisher(for: .myNotiName)
    .sink(receiveValue: { notification in
        print(notification.object as? String)
    })

// 创建通知
let noti = Notification(name: .myNotiName, object: "some info", userInfo: nil)
// 发送通知
NotificationCenter.default.post(noti)
```

- SwiftUI 监听 App 进入后台和返回前台。

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("")
            // onReceive捕获通知
            .onReceive(NotificationCenter.default.publisher(for:
                UIApplication.willResignActiveNotification)) { _ in
                print("App进入后台")
        }.onReceive(NotificationCenter.default.publisher(for:
            UIApplication.willEnterForegroundNotification)) { _ in
                print("App进入前台")
        }.onAppear() {
                print("第一次显示")
        }
    }
```

KVO Publisher

任何 NSObject 对象一旦被 KVO 监听，则可以成为一个 Publisher。

```
import Combine
import UIKit

class Person: NSObject {
    @objc dynamic var age: Int = 0
}

let person = Person()

let subscription = person.publisher(for: \.age)
    .sink { newValue in
        print("person的age改成了\(newValue)")
    }

person.age = 10 // 改变时会收到通知
```

开发中常见的 KVO Publisher 操作:

```
let scrollView = UIScrollView()
scrollView.publisher(for: \.contentOffset)

let avPlayer = AVPlayer()
avPlayer.publisher(for: \.status)

let operation = Operation()
operation.publisher(for: \.queuePriority)
```

Timer Publisher

当 Subscriber 开始订阅后，大部分的 Publisher 会立即提供数据， 如 `Just` 。但有一种遵守 `ConnectablePublisher` 协议的 Publisher，它需要某种机制来启动数据流。 `Timer Publisher` 就是这种类型的 Publisher。 `ConnectablePublisher` 不同于普通的 Publisher，需要明确地对其调用 `connect()` 或者 `autoconnect()` 方法，它才会开始发送数据。

- autoconnect()

```
import UIKit
import Combine

// every: 间隔时间 on: 在哪个线程 in: 在哪个Runloop
let subscription = Timer.publish(every: 1, on: .main, in: .default)
    .autoconnect()
    .sink { _ in
        print("Hello")
}

// 可以取消
```

```
// subscription.cancel()
```

- connect()

```
import UIKit
import Combine

// every: 间隔时间 on: 在哪个线程 in: 在哪个Runloop
let timerPublisher = Timer.publish(every: 1, on: .main, in: .default)

let cancellablePublisher = timerPublisher
    .sink { _ in
        print("World")
}

let subscription = timerPublisher.connect()

// 可以取消
// subscription.cancel()
```

SwiftUI中使用

- 案例一。

```
struct ContentView: View {
    // Combine Timer
    @State private var timer = Timer.publish(every: 1, on: .main, in: .default).autoconnect()
    // 倒计时
    @State private var count = 5

    var body: some View {
        Text("倒计时\(count)")
            .onReceive(timer) { _ in // 订阅
                count = count - 1
                if count == 0 {
                    // 取消
                    timer.upstream.connect().cancel()
                }
            }
    }
}
```

- 案例二。

```
import Combine
import SwiftUI

class TimerViewModel: ObservableObject {
    // Combine Timer
    private let timer = Timer.publish(every: 1, on: .main, in: .common).autoconnect()
    // 订阅
    private var timerSubscription: Cancellable?
    // 计数
    @Published var count = 0

    // 开始计时
    func startTimer() {
        timerSubscription = timer.sink(receiveValue: { _ in
            self.count = self.count + 1
        })
    }

    // 停止计时
    func stopTimer() {
        timerSubscription = nil
    }
}

struct ContentView: View {
    @ObservedObject var viewModel: TimerViewModel = TimerViewModel()

    var body: some View {
        VStack {
```

```
            Text(viewModel.count.description)

            HStack {
                Button("开始") {
                    viewModel.startTimer()
                }

                Button("停止") {
                    viewModel.stopTimer()
                }
            }
        }
    }
}
```

@Published

这个注解是一个属性包装（Property Wrapper），可以为任何一个属性生成其对应类型的 Publisher，这个 Publisher 会在属性值发生变化时发送消息。`@Published` 广泛应用于 UIKit 与 SwiftUI 中，用 `@Published` 修饰属性以后，通过 `$属性名` 即可得到该属性对应的 Publisher。

- 类中使用。

```
import Combine

class Student {
    @Published var name: String = "zhangsan"
    @Published var age: Int = 20
}

let stu = Student()

// Publisher: $name
let subscription1 = stu.$name.sink {
    print($0)
}
stu.name = "lisi"

// Publisher: $age
let subscription2 = stu.$age.sink {
    print($0)
}
stu.age = 30


/* 输出
zhangsan
lisi
20
30
*/
```

- 协议中使用。

```
// 定义Protocol，通过Published将实际类型包裹起来。
protocol modelProtocol {
    var namePublisher: Published<String>.Publisher { get }
}

// 遵守协议，将name的值返回给namePublisher。
class Student: modelProtocol {
    @Published var name: String

    var namePublisher: Published<String>.Publisher { $name }

    init(name: String) {
        self.name = name
    }
}

let student = Student(name: "zhangsan")

let subscription = student.namePublisher
    .sink {
```

```
        print("hello \($0)")
    }

student.name = "lisi"
student.name = "wangwu"

/* 输出
 hello zhangsan
 hello lisi
 hello wangwu
 */
```

Common Operators

转换类

collect

将 Publisher 发出的数据收集到数组中。

```
import Combine

["A", "B", "C", "D", "E"].publisher
    .collect()
    .sink(
        receiveCompletion: {
            print($0)
    }, receiveValue: {
        print($0)
    }
)

/*输出
["A", "B", "C", "D", "E"]
finished
*/
```

还可以指定收集的元素个数

```
import Combine

["A", "B", "C", "D", "E"].publisher
    .collect(2)
    .sink(
        receiveCompletion: {
            print($0)
    }, receiveValue: {
        print($0)
    }
)

/*输出
["A", "B"]
["C", "D"]
["E"]
finished
*/
```

scan

- 第一个参数是初始值。
- 第二参数是尾随闭包，接受两个参数：
  - 参数1：闭包最后一次返回的值。
  - 参数2：Publisher 当前发出的值。

```
import Combine

// 对序列进行累加，并输出每次的值
[1,  2,  3,  4,  5].publisher
    .scan(0) { $0 + $1}
    .sink(receiveValue: {
        print($0)
    })

/* 输出
1
3
6
10
15
*/
```

flatMap

- 将多个 Publisher 扁平化为一个 Publisher。
- 案例：顺序执行多个网络请求。

```
import UIKit
import Combine

let subscription = URLSession.shared.dataTaskPublisher(for: URL(string: "https://www.example1.com")!)
    .flatMap { data, response in
        URLSession.shared.dataTaskPublisher(for: URL(string: "https://www.example2.com")!)
}
    .flatMap { data, response in
        URLSession.shared.dataTaskPublisher(for: URL(string: "https://www.example3.com")!)
}
    .sink(receiveCompletion: {_ in print("receiveCompletion") },
      receiveValue: { value in  })
```

过滤类

filter

接收一个返回 `Bool` 类型的闭包，返回 `false` 时，传入的值就会被过滤掉。

```
import Combine
```

```
(1...10).publisher
    .filter { $0.isMultiple(of: 3) }
    .sink(receiveValue: { print($0) })

/* 输出
3
6
9
*/
```

replaceNil

将 Publisher 中的 `nil` 的值替换成指定的值。

```
import Combine

["A", nil, "C"].publisher
    .replaceNil(with: "*_*")
    .map { $0! }
    .sink(receiveValue: { print($0) })

/*输出
A
*_*
C
*/
```

removeDuplicates

过滤掉**连续重复**的数据。

```
import Combine

let userInput = ["aaa", "aaa", "bbbb", "ccc", "bbbb"].publisher
let subscription = userInput
    .removeDuplicates()
    .sink(receiveValue: { print($0) })

/* 输出
aaa
bbbb
ccc
bbbb
*/
```

ignoreOutput

如果只想知道 Publisher 什么时候结束，但不关心它发出的数据，可以使用 ignoreOutput。

```
import Combine

(1...10).publisher
    .ignoreOutput()
    .sink(receiveCompletion: {
        print("Completed with: \($0)")
    },receiveValue: {
        print($0)
    })

/* 输出
Completed with: finished
*/
```

合规类

reduce

用法和 scan 类似，只不过 reduce 会发出最后一次闭包的运算结果。

```
import Combine

// Publisher将发布五个output值，当序列中值耗尽时，它将发布finished。而经过reduce变形后，新的Publisher只会在接到上游发出的finished事件后，
才会将reduce后的结果发布出来
[1,  2,  3,  4,  5].publisher
    .reduce(0) { $0 + $1}
    .sink(receiveValue: {
        print($0)
    })

/* 输出
15
*/
```

序列类

min

找出 Publisher 所发出的全部数据中的最小值。

```
import Combine

(1...10).publisher
    .min()
    .sink(receiveValue: {
        print($0)
    })

/* 输出
1
*/
```

first

找出 Publisher 的第一个数据，然后就马上结束，并取消对 Publisher 的订阅。

```
import Combine

(5...10).publisher
    .first()
    .sink(receiveValue: {
        print($0)
    })

/* 输出
5
*/
```

count

计算 Publisher 发出的所有数据的个数。

```
import Combine

(1...10).publisher
    .count()
    .sink(receiveValue: {
        print($0)
    })

/* 输出
10
*/
```

调整类

switchToLatest

从一个 Publiser 切换到另一个，这会停止接收之前 Publiser 的数据而改成接收新切换的 Publiser 中的数据。

```
import Combine

let publisher1 = PassthroughSubject<Int, Never>()
let publisher2 = PassthroughSubject<Int, Never>()
let publisher3 = PassthroughSubject<Int, Never>()

let publishers = PassthroughSubject<PassthroughSubject<Int, Never>, Never>()

let subscription = publishers
    .switchToLatest()
    .sink(receiveCompletion: { _ in print("Completed!") },
          receiveValue: { print($0) })

publishers.send(publisher1)
publisher1.send(1)
publisher1.send(2)

publishers.send(publisher2)
publisher1.send(3)
publisher2.send(4)
publisher2.send(5)

publishers.send(publisher3)
publisher2.send(6)
publisher3.send(7)
publisher3.send(8)
publisher3.send(9)

/* 输出
1
2
4
5
7
8
9
*/
```

资源管理类

share

- 将值类型的 Publisher 包装为引用类型。
- 对于网络等资源密集型操作进行 share 可避免因大量不必要的请求导致的内存问题。
- 案例：只执行一次网络请求的情况下想要多个 Subscriber 接收到数据。

```
import UIKit
import Combine

// 默认情况下dataTaskPublisher是struct
let shared = URLSession.shared
    .dataTaskPublisher(for: URL(string: "https://www.baidu.com")!)
    .share() // 通过share()转成引用类型

print("=====subscribing first=====")

let subscription1 = shared
    .sink(receiveCompletion: {
        print("subscription1 \($0)")
    }, receiveValue: {
        print("subscription1 received: '\($0)'")
    }
)

print("=====subscribing second=====")
```

```
let subscription2 = shared
    .sink(receiveCompletion: {
        print("subscription2 \($0)")
    }, receiveValue: {
        print("subscription2 received: '\($0)'")
    }
)

/* 输出
=====subscribing first=====
=====subscribing second=====
subscription1 received: '(data: 2443 bytes...)'
subscription2 received: '(data: 2443 bytes...)'
subscription1 finished
subscription2 finished
*/
```

可以看出：

- 第一次 `sink` 调用触发了订阅。
- 第二次 `sink` 并没有触发什么，而 Publisher 继续执行。
- 请求完成后，两个 Subscriber 都收到了数据。

存在的问题

share 没有任何缓冲系统，这意味着如果第 2 个订阅发生在请求完成之后，则它将仅接收完成事件。

```
import UIKit
import Combine

// 默认情况下dataTaskPublisher是struct
let shared = URLSession.shared
    .dataTaskPublisher(for: URL(string: "https://www.baidu.com")!)
    .share() // 通过share()转成引用

print("=====subscribing first=====")

let subscription1 = shared
    .sink(receiveCompletion: {
        print("subscription1 \($0)")
    }, receiveValue: {
        print("subscription1 received: '\($0)'")
    }
)

print("=====subscribing second=====")

DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
    let subscription2 = shared
        .sink(receiveCompletion: {
            print("subscription2 \($0)")
        }, receiveValue: {
            print("subscription2 received: '\($0)'")
        }
    )
}

/* 输出
=====subscribing first=====
=====subscribing second=====
subscription1 received: '(data: 2443 bytes...)'
subscription1 finished
subscription2 finished
*/
```

multicast

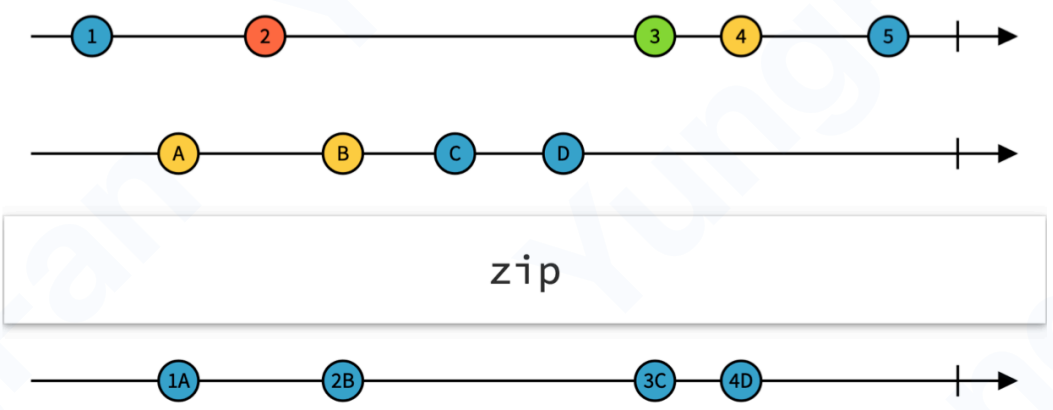
- share 存在的问题可以使用 multicast 解决。
- 返回一个 `ConnectablePublisher` 。在主动调用 `connect()` 之前，它不会向上游 Publisher 发出订阅。
- 必须提供一个 `Subject` 类型的参数。

```swift
import UIKit
import Combine

var cancellables: Set<AnyCancellable> = []

let subject = PassthroughSubject<Data, URLError>()
// 默认情况下dataTaskPublisher是struct
let multicast = URLSession.shared
    .dataTaskPublisher(for: URL(string: "https://www.baidu.com")!)
    .map(\.data)
    .multicast(subject: subject)

print("=====subscribing first=====")

multicast
    .sink(receiveCompletion: {
        print("subscription1 \($0)")
    }, receiveValue: {
        print("subscription1 received: '\($0)'")
    }
).store(in: &cancellables)

print("=====subscribing second=====")

DispatchQueue.main.asyncAfter(deadline: .now() + 2.0, execute: {
    multicast
        .sink(receiveCompletion: {
            print("subscription2 \($0)")
        }, receiveValue: {
            print("subscription2 received: '\($0)'")
        }
    ).store(in: &cancellables)

    multicast.connect().store(in: &cancellables)
})

/*输出
=====subscribing first=====
=====subscribing second=====
subscription1 received: '2443 bytes'
subscription2 received: '2443 bytes'
subscription1 finished
subscription2 finished
*/
```

组合类

zip

通过传入两个 Publisher（要求**Failure类型一致**），输出组合的 Publisher。当组合的每一个 Publisher 都产生数据的时候，才会取出 `index` 相同的数据数据合并成**元组**发送给 Subscriber。除此以外，还有 3 个参数和 4 个参数的 zip 用于更多 Publisher 的组合。




```swift
import Combine

let publisher1 = PassthroughSubject<Int, Never>()
let publisher2 = PassthroughSubject<String, Never>()

let subscription = publisher1
    .zip(publisher2)
    .sink(receiveCompletion: { _ in print("Completed") },
          receiveValue: { print("P1: \($0), P2: \($1)") })
```

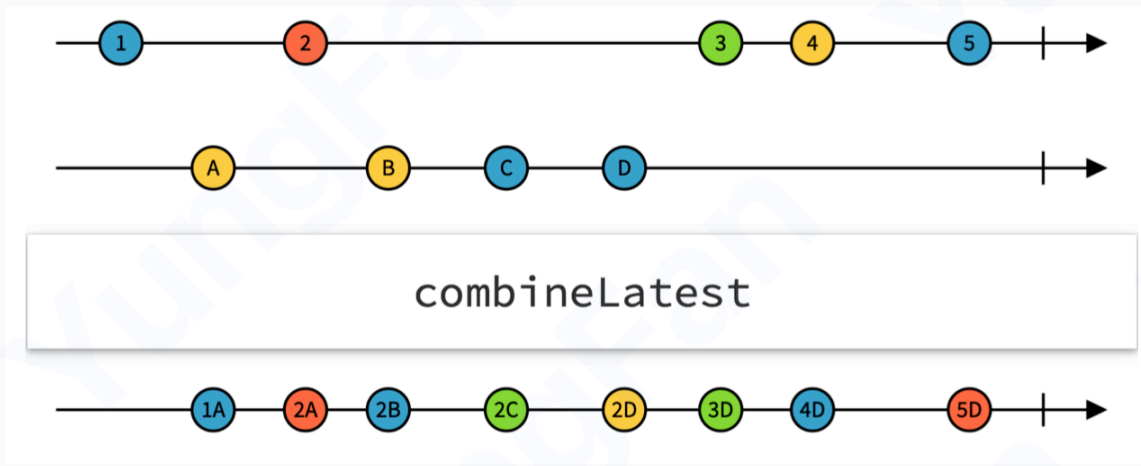
```
publisher1.send(1)
publisher1.send(2)

publisher2.send("a")
publisher2.send("b")

publisher1.send(3)
publisher2.send("c")

publisher1.send(completion: .finished)
publisher2.send(completion: .finished)

/* 输出
P1: 1, P2: a
P1: 2, P2: b
P1: 3, P2: c
Completed
*/
```

应用：并行执行多个网络请求

```
import UIKit
import Combine

let combined = Publishers.Zip3(
    URLSession.shared.dataTaskPublisher(for: URL(string: "https://www.example1.com")!),
    URLSession.shared.dataTaskPublisher(for: URL(string: "https://www.example2.com")!),
    URLSession.shared.dataTaskPublisher(for: URL(string: "https://www.example3.com")!)
)

let subscription = combined.sink(receiveCompletion: {_ in print("receiveCompletion") },
              receiveValue: { (value1, value2, value3) in  })
```

combineLatest

当 Publisher1 发布数据或者 Publisher2 发布数据时，将两个数据合并，作为新的数据发布出去。不论是哪个 Publisher，只要发布了新的数据，combineLatest 就把最新的数据和另一个 Publisher 中的最新的数据合并。除此以外，还有 3 个参数和 4 个参数的 combineLatest 用于更多 Publisher 的组合。




```
import Combine

let publisher1 = PassthroughSubject<Int, Never>()
let publisher2 = PassthroughSubject<String, Never>()

let subscription = publisher1
    .combineLatest(publisher2)
    .sink(receiveCompletion: { _ in print("Completed") },
          receiveValue: { print("P1: \($0), P2: \($1)") })

publisher1.send(1)
publisher1.send(2)

publisher2.send("a")
publisher2.send("b")

publisher1.send(3)
publisher2.send("c")

publisher1.send(completion: .finished)
publisher2.send(completion: .finished)
```

```
/* 输出
P1: 2, P2: a
P1: 2, P2: b
P1: 3, P2: b
P1: 3, P2: c
Completed
*/
```

merge

传入两个 Publisher（要求**Output和Failure类型一致**），输出混合后的 Publisher。它会把两个 Publisher 发出的数据根据发出的时间顺序合并后发送给 Subscriber。除此以外，还有 3 ～ 8 个参数的 merge 用于更多 Publisher 的组合。

```
import Combine

let publisher1 = PassthroughSubject<Int, Never>()
let publisher2 = PassthroughSubject<Int, Never>()

let subscription = publisher1
    .merge(with: publisher2)
    .sink { print($0) }

publisher1.send(1)
publisher1.send(2)

publisher2.send(11)
publisher2.send(22)

publisher1.send(3)
publisher2.send(33)

publisher1.send(completion: .finished)
publisher2.send(completion: .finished)

/* 输出
1
2
11
22
3
33
*/
```

异常处理类

Combine 中也会出现异常，比如 Publisher 与 Subscriber 类型不匹配，Scheduler 不匹配等，一旦出现异常应该如何处理呢？Combine 提供了几种异常处理机制。在下面的案例中，使用 `tryMap` 操作符来抛出异常。

catch与replaceError

- catch：用一个新 Publisher 替换失败的 Publisher。

```
import Combine

struct SomeError: Error {
}

let subscription = Just(1)
    .tryMap { _ in throw SomeError() }
    .catch { _ in Just(2) } // 创建新的Publisher
    .sink { print($0) }

/*输出
2
*/
```

- replaceError：行为类似于 catch，只是它用某个数据替换异常，而不是创建新的 Publisher。

```
import Combine
```

```
struct SomeError: Error {
}

let subscription = Just(1)
    .tryMap { _ -> in throw SomeError() }
    .replaceError(with: 2)
    .sink { print($0) }

/*输出
2
*/
```

retry

在发生异常的情况下，重新向先前的 Publisher 发送给定次数的请求，当所有尝试都用尽之后才往数据流的下游传播异常，这种方式在网络开发中常用。

```
import UIKit
import Combine

let url = URL(string: "https://www.baidu.com")

let subscription = URLSession.shared.dataTaskPublisher(for: url!)
   .retry(3) // 尝试3次
   .sink(receiveCompletion: { print($0) },
         receiveValue: { value in print(value) })
```

调试类

Combine 提供如下的 Operator 进行程序的调试。

print

```
import Combine

let subscription = [1, 2, 3].publisher
    .print("debug info")
    .sink { _ in }

/* 输出
debug info: receive subscription: ([1, 2, 3])
debug info: request unlimited
debug info: receive value: (1)
debug info: receive value: (2)
debug info: receive value: (3)
debug info: receive finished
*/
```

handleEvents

```
import Combine

let subscription = [1, 2, 3].publisher
    .handleEvents(receiveSubscription: { print("Receive subscription: \($0)") },
                  receiveOutput: { print("Receive output: \($0)") },
                  receiveCompletion: { print("Receive completion: \($0)") },
                  receiveCancel: { print("Receive cancel") },
                  receiveRequest: { print("Receive request: \($0)") })
    .sink { _ in }

/* 输出
Receive request: unlimited
Receive subscription: [1, 2, 3]
Receive output: 1
Receive output: 2
Receive output: 3
```

```
Receive completion: finished
*/
```

Scheduler

介绍

内置Scheduler

`Scheduler` 在是一个协议，遵守了该协议的内置 Scheduler 有：

- `DispatchQueue` 。
- `OperationQueue` 。
- `RunLoop` 。
- `ImmediateScheduler` ： 立即执行同步操作， 如果使用它执行延迟的工作，会报错。

使用 `RunLoop.main` 、 `DispatchQueue.main` 和 `OperationQueue.main` 来执行与 UI 相关的操作。

receive与subscribe

默认情况下，当前的 Scheduler 与**最初产生数据**的 Publisher 所在的 Scheduler 相同。但是实际情况往往是在整个数据流中需要切换 Scheduler，所以 Combine 提供了两个函数来设置 Scheduler。

receive(on:)

定义了在哪个 Scheduler 完成 Publisher 的订阅。（**在哪里接收数据**）

```
import Combine

let subscription = Just(1)
    .map { _ in print(Thread.isMainThread) }
    .receive(on: DispatchQueue.global())
    .sink { print(Thread.isMainThread) }

/* 输出
true
false
false
*/
```

subscribe(on:)

定义了 Publisher 在哪个 Scheduler 来发布数据。（**在哪里发布数据**）

```
import Combine

let subscription = Just(1)
    .subscribe(on: DispatchQueue.global())
```

```
    .map { _ in print(Thread.isMainThread) }
    .sink { print(Thread.isMainThread) }

/* 输出
false
false
*/


let subscription = Just(1)
    .subscribe(on: DispatchQueue.global())
    .map { _ in print(Thread.isMainThread) }
    .receive(on: DispatchQueue.main)
    .sink { print(Thread.isMainThread) }

/* 输出
false
true
*/
```

案例

```
import UIKit
import Combine

// 默认情况
let subscription = URLSession.shared
    .dataTaskPublisher(for: URL(string: "https://www.baidu.com")!)
    .map{ $0.data }
    .sink(receiveCompletion: { print($0)
    }) { _ in
        print(Thread.isMainThread)
}

/* 输出
false
finished
*/


// 加上receive(on:)
let subscription = URLSession.shared
    .dataTaskPublisher(for: URL(string: "https://www.baidu.com")!)
    .map{ $0.data }
    .receive(on: DispatchQueue.main)
    .sink(receiveCompletion: { print($0)
    }) { _ in
        print(Thread.isMainThread)
}

/* 输出
true
finished
*/
```

相关Operator

很多 Operator 的参数都含有一个 Scheduler，包括`delay、timeout、debounce、throttle`。

delay

延迟 Publisher 在某个 Scheduler 上数据的发送。

```
import UIKit
import Combine

let subject = PassthroughSubject<String, Never>()
let subscription = subject
```

```
            .delay(for: 3, scheduler: DispatchQueue.main)
            .sink { data in
                print("delay:" + data)
            }

// 输入Hello
subject.send("Hello")

/* 3秒后输出
 delay:Hello
*/
```

timeout

超时，如果上游 Publisher 超过指定的时间间隔而没有生成数据，则终止发布。

```
import UIKit
import Combine

let subject = PassthroughSubject<String, Never>()
// delay时间超过timeout 不会有输出
let subscription = subject
            .delay(for: 2, scheduler: DispatchQueue.main)
            .timeout(4, scheduler: RunLoop.main)
            .sink { data in
                print("timeout:" + data)
            }

// 发送Hello
subject.send("Hello")

/* 2秒后输出
 delay:Hello
*/
```

debounce

翻译为 “防抖”，Publisher 在接收到第一个数据后，并不是立即将它发布出去，而是会开启一个内部计时器，当一定时间内没有新的数据来到，再将这个数据进行发布。如果在计时期间有新的数据，则重置计时器并重复上述等待过程。（**时间间隔很重要**）

```
import UIKit
import Combine

let subject = PassthroughSubject<String, Never>()

// 主要理解1s的意思：它指的是当前Publisher发出的最后一个值的时间到当前时间间隔为1s时，并且Publisher还未结束，debounced才会发出Publisher当前的最后一个值。
let subscription = subject
            .debounce(for: .seconds(1), scheduler: DispatchQueue.main)
            .sink { data in
                print("debounce:" + data)
            }

let typingHelloWorld: [(TimeInterval, String)] = [
    (0.0, "H"),
    (0.1, "He"),
    (0.2, "Hel"),
    (0.3, "Hell"),
    (0.4, "Hello"),
    (1.6, "HelloC"),
    (1.7, "HelloCo"),
    (2.8, "HelloCom"),
    (2.9, "HelloComb"),
    (3.0, "HelloCombi"),
    (3.1, "HelloCombin"),
    (3.2, "HelloCombine")
]

//模拟输入：HelloCombine
typingHelloWorld.forEach { (delay, str) in
    DispatchQueue.main.asyncAfter(deadline: .now() + delay) {
        subject.send(str)
```

```
    }
}

/* 输出
debounce:Hello
debounce:HelloCo
debounce:HelloCombine
*/
```

分析

1. `(0.4, "Hello")` 与 `(1.6, "HelloC")` 之间间隔大于1s， `debounced` 会在这中间发出 `Hello` ，并且设置新的时间点为 `0.4+1=1.4s` 。
2. `(1.7, "HelloCo")` 与 `(2.8, "HelloCom")` 之间间隔大于1s， `debounced` 会在这中间发出 `HelloCo` ，并且设置新的时间点为 `1.7+1=2.7s` 。
3. 最后在 3.2s 发出 `HelloCombine` 之后， `PassthroughSubject` 没有结束也没有发出任何值了，所以 `debounced` 会在 `3.2+1=4.2s` 发出 `HelloCombine` 。

throttle

翻译为 “节流”，在固定时间内只发出一个数据，过滤掉其他数据，可以选择最后一个或第一个数据发出。它会在收到一个数据后开始计时，并忽略计时周期内的后续输入。（**时间区间很重要**）

```
import UIKit
import Combine

let subject = PassthroughSubject<String, Never>()

// 这里的1s指的是：throttled每隔1s就会发出最近的一个1s区间内Publisher发出的第一个值。
// latest设置为false，意思是从每1秒的区间发出的值中取第一个值，如果是true就取最后一个值
let subscription = subject
            .throttle(for: .seconds(1), scheduler: DispatchQueue.main, latest: false)
            .sink { data in
                print("throttle:" + data)
            }

let typingHelloWorld: [(TimeInterval, String)] = [
    (0.0, "H"),
    (0.1, "He"),
    (0.2, "Hel"),
    (0.3, "Hell"),
    (0.4, "Hello"),
    (1.0, "HelloC"),
    (1.2, "HelloCo"),
    (1.5, "HelloCom"),
    (2.0, "HelloComb"),
    (2.1, "HelloCombi"),
    (3.2, "HelloCombin"),
    (3.3, "HelloCombine")
]

//模拟输入：HelloCombine
typingHelloWorld.forEach { (delay, str) in
    DispatchQueue.main.asyncAfter(deadline: .now() + delay) {
        subject.send(str)
    }
}

/* 输出
throttle:H
throttle:HelloC
throttle:HelloComb
throttle:HelloCombin
*/
```

分析

1. 在 0 ~ 1s 中， `PassthroughSubject` 发出的第一个值是 `H` ，所以 `throttled` 在 1.0s 的时候发出 `H` 。
2. 在 1 ~ 2s 中， `PassthroughSubject` 发出的第一个值是 `HelloC` ，所以 `throttled` 在 2.0s 的时候发出 `HelloC` 。
3. 在 2 ~ 3s 中， `PassthroughSubject` 发出的第一个值是 `HelloComb` ，所以 `throttled` 在 3.0s 的时候发出 `HelloComb` 。
4. 在 3 ~ 4s 中， `PassthroughSubject` 发出的第一个值是 `HelloCombin` ，所以 `throttled` 在 4.0s 的时候发出 `HelloCombin` 。

Future

介绍

前面讲解了很多 Publisher 如 `Just` 等，那些 Publisher 其数据的发布和订阅是同步行为。但是如果希望数据的发布和订阅是异步的，需要使用 `Future` 。 `Future` 表示异步操作的最终完成或失败，定义如下：

```
final public class Future<Output, Failure>: Publisher where Failure: Error {

    public typealias Promise = (Result<Output, Failure>) -> Void

    public init(_ attemptToFulfill: @escaping (@escaping Future<Output, Failure>.Promise) -> Void)

    final public func receive<S>(subscriber: S) where Output == S.Input, Failure == S.Failure, S : Subscriber
}
```

- Future 是一个类，实现了 Publisher 协议。
- **Future 会在初始化时立刻执行闭包，在该闭包里完成异步的操作。所以需要存储异步处理的结果，然后发送给一个或多个 Subscriber。**
- 无论有多少 Subscriber 订阅，Future 的异步操作只会执行一次，执行完就结束。

Promise

从上面的定义可以看出，其本质是 `(Result<Output, Failure>) -> Void` 的类型别名，它表示 Future 中异步操作的最终结果。

成功的处理

```
import UIKit
import Combine

let future = Future<Int, Never> { promise in
    DispatchQueue.main.asyncAfter(deadline: .now() + 3) {
        promise(.success(100))
    }
}

let subscription = future.sink(receiveValue: { value in
    print(value)
})
```

失败的处理

```
import UIKit
import Combine

struct SomeError: Error {
}

let future = Future<Int, SomeError> { promise in
    DispatchQueue.main.asyncAfter(deadline: .now() + 3) {
        promise(.failure(SomeError()))
    }
}
```

```
let subscription = future.sink(receiveCompletion: { completion in
    if case .failure(let error) = completion {
        // 失败的处理
        print()
    }
}, receiveValue: { _ in
    // 成功的处理
})
```

基本使用

```
import Combine

// 返回一个Future对象且会产生一个Int类型的值
func createFuture() -> Future<Int, Never> {
    // 返回一个Future，它是一个闭包
    // 在该闭包里执行异步操作，只会执行一次
    return Future { promise in
        // 异步操作

        // 最后必须调用promise完成工作
        promise(.success(100))
    }
}

createFuture().sink(receiveValue: { value in
    print(value)
})

/* 输出
100
*/
```

说明

1. 当创建一个 Future 时，它会立即开始执行。
2. Future 将只运行一次提供的闭包。
3. 多次订阅同一个 Future 将返回同一个结果。

案例

```
import Combine

// 构造模型
struct Model {
    let name: String
}

// 异步获取模型数据
enum ModelService {
    static func getModels() -> Future<[Model], Error> {
        return Future { promise in
            print("执行Future")
            DispatchQueue.global().async {
                let models = [Model(name: "ZhangSan"), Model(name: "LiSi"), Model(name: "WangWu")]
                promise(.success(models))
                print("\(Thread.current)获取数据完成")
            }
        }
    }
}

// 订阅
let subscription = ModelService.getModels().sink(
    receiveCompletion: { completion in
        switch completion {
        case .finished:
            print("数据处理完成")
```

```
        case let .failure(error):
            print("程序出现了错误\(error.localizedDescription)")
        }
    },
    receiveValue: { models in
        print("\(Thread.current)得到数据，进行处理")
        models.forEach {
            print($0.name)
        }
    }
)
/* 输出
 执行Future
 <NSThread: 0x6000027ec600>{number = 3, name = (null)}获取数据完成
 <NSThread: 0x6000027cc840>{number = 1, name = main}得到数据，进行处理
 ZhangSan
 LiSi
 WangWu
 数据处理完成
 */
```

应用—包装异步操作

```
import UIKit
import Combine

var cancellables: Set<AnyCancellable> = []

func fetchData(from url: URL) -> Future<Data, URLError> {
    return Future<Data, URLError> { promise in
        URLSession.shared
            .dataTaskPublisher(for: url)
            .map(\.data)
            .print("Future")
            .sink(receiveCompletion: { completion in
                if case .failure(let error) = completion {
                    promise(.failure(error))
                }
            }, receiveValue: {
                promise(.success($0))
            }).store(in: &cancellables) // 存储订阅者维持较长生命周期
    }
}


fetchData(from: URL(string: "https://www.baidu.com")!)
    .sink(receiveCompletion: { completion in
        if case .failure(let error) = completion {
            // 失败的处理
            print("程序出现了错误\(error.localizedDescription)")
        }
    }, receiveValue: { _ in
        // 成功的处理
        print("得到数据，进行处理")
    }).store(in: &cancellables) // 存储订阅者维持较长生命周期

/* 输出
Future: receive subscription: (DataTaskPublisher)
Future: request unlimited
Future: receive value: (2443 bytes)
得到数据，进行处理
Future: receive finished
*/
```

Practice

介绍

Combine 既可以在 SwiftUI 中使用，也可以在 UIKit 中使用。在使用 Combine 进行编程的时候，需要思考的问题是：

1. 最初的 Publisher 从何而来? — **如何构建 Publisher**
2. 最终的 Publisher 到哪里去? — **在哪订阅 Publisher**

SwiftUI

声明式UI + 响应式编程是未来移动开发的趋势，所以 Combine 对于 SwiftUI 来说是不可或缺的一部分，这也是为什么 Combine 会随着 SwiftUI 一起发布。在 SwiftUI 中任何一个 View 都可以作为 Subscriber。 SwiftUI 中的 View 协议定义了一个 `onReceive()` 的函数可以将 View 变成 Subscriber。 `onReceive()` 函数接收一个 Publisher，然后跟上一个类似于 `sink` 的闭包，可以在其中操作 `@State` 或 `@Binding` 修饰的属性数据。

```
import SwiftUI

struct ContentView: View {
    @State private var currentValue = "😊"

    var body: some View {
        VStack{
            Text(currentValue)

            Text(currentValue)
                .onReceive(Just("SwiftUI + Combine")) { value in
                    self.currentValue = value
                }
        }
    }
}
```

UIKit

虽然 SwiftUI + Combine 是一对黄金搭档，但是在 UIKit 中 Combine 也可以发挥重要作用。如下图的案例，当开关打开（关闭）的时候，按钮可以（不能）点击，点击发送通知按钮，蓝色的标签显示发送的通知内容。


```
import UIKit
import Combine

extension Notification.Name{
    static var newMessage = Notification.Name("YungFan")
}

class ViewController: UIViewController {
    @IBOutlet weak var allowMessageSwitch: UIButton!
    @IBOutlet weak var sendButton: UIButton!
    @IBOutlet weak var messageLabel: UILabel!

    @Published var canSendMessage: Bool = false
    private var cancellables: Set<AnyCancellable> = []

    override func viewDidLoad() {
        super.viewDidLoad()

        // canSendMessage的改变绑定到Button的isEnabled上
        $canSendMessage
            .receive(on: DispatchQueue.main)
            .assign(to: \.isEnabled, on: sendButton)
            .store(in: &cancellables)

        // 通知需要绑定到messageLabel的text
        NotificationCenter.default.publisher(for: .newMessage)
            .map{ notification -> String in
                notification.object as? String ?? "default Value"
        }
            .assign(to: \.text, on: messageLabel)
            .store(in: &cancellables)

        /*
         // 上面的写法等于下面3句
        let messagePublisher = NotificationCenter.Publisher(center: .default, name: .newMessage)
        let messageSubscriber = Subscribers.Assign(object: messageLabel, keyPath: \.text)
        messagePublisher
            .map{ notification -> String in
                notification.object as? String ?? ""
        }.subscribe(messageSubscriber)
        */
    }

    @IBAction func switchChanged(_ sender: UISwitch) {
        // canSendMessage的改变随开关改变
        self.canSendMessage = sender.isOn
    }

    @IBAction func buttonClicked(_ sender: UIButton) {
        // 发送通知
        NotificationCenter.default.post(name: .newMessage, object: "UIKit + Combine")
    }
}
```

应用

以注册界面为例，控件由 3 个输入框，3个提示图片和 1 个按钮组成，需要满足以下条件：

1. 用户名必须满足验证要求；
2. 密码的不能少于 6 位；
3. 密码和确认密码必须相同；
4. 注册按钮只有在 1–3 条件成立时才可以点击。

分析

1. 最初的 Publisher 从何而来?
   最初的 Publisher 都来自输入框的输入内容。
2. 最终的 Publisher 到哪里去?
   最终的 Publisher 被订阅完成图片的显隐与按钮能否点击的切换。

UIKit实现

拖拽界面

构建4个Publisher（开发的核心）

1. 检验用户名的 Publisher
2. 检验密码长度的 Publisher
3. 检验两次密码是否一致的 Publisher
4. 检验用户名和密码同时有数据的 Publisher

```
// 1.检验用户名的Publisher
extension ViewController {
    // 1.1 提交给服务器判断用户名是否合法，网络请求等异步行为
    func usernameChecked(_ username: String, completion: @escaping ((Bool) -> ())) {
        // 模拟网络验证的过程
        DispatchQueue.global().asyncAfter(deadline: .now() + 1.0) {
            if username == "123456" {
                completion(true)
            }
            else {
                completion(false)
            }
        }
    }

    // 1.2 第一个：验证用户名是否合法
    var validatedUsername: AnyPublisher<String?, Never> {
        // 限制产生数据的频率
        return $username.debounce(for: 0.5, scheduler: RunLoop.main)
            .removeDuplicates() // 去重，重复的不需要再次检验
            // map用于元素的转换，flatMap用与Publisher的转换
            .flatMap { username in
                // 使用 Future 包装已有的异步操作
                return Future { promise in
                    self.usernameChecked(username) { available in
                        promise(.success(available ? username : nil))
                    }
                }
        }.eraseToAnyPublisher()
    }
}

// 2.检验密码的Publisher
extension ViewController {
    // 2.1 第二个：验证密码长度是否大于6
    var valiatedPassword: AnyPublisher<String?, Never> {
        return $password.flatMap{ password in
```

```
            return  Just(password.count > 6 ? password : nil)
        }.eraseToAnyPublisher()
    }

    // 2.2 第三个：验证两次密码是否一致
    var valiatedRepassword: AnyPublisher<String?, Never> {
        // 注意这里合并的不是$password而是上一步的valiatedPassword
        return valiatedPassword.combineLatest($repassword).flatMap{ (password, repassword) in
            return Just(password == repassword ? password : nil)
        }.eraseToAnyPublisher()
    }
}

// 3.检验用户名和密码的Publisher
extension ViewController {
    // 第四个：验证用户名和密码
    var validatedAccount: AnyPublisher<(String, String)?, Never> {
        // 合并第一步和第二步产生的Publisher
        validatedUsername.combineLatest(valiatedRepassword).map({ (username, password) -> (String, String)? in
            guard let uname = username, let pwd = password else { return nil }
            return (uname, pwd)
        }).eraseToAnyPublisher()
    }
}
```

将输入的内容与Publisher绑定

用 @Published 修饰属性成为最初的 Publisher，通过 UITextField 的 `Editing Changed` 事件将用户输入的数据实时绑定到属性上。

```
// 属性发布者
@Published var username: String = ""
@Published var password: String = ""
@Published var repassword: String = ""

// 三个输入框的Editing Changed
@IBAction func usernameChanged(_ sender: UITextField) {
    username = sender.text ?? ""
}

@IBAction func passwordChanged(_ sender: UITextField) {
    password = sender.text ?? ""
}

@IBAction func repasswordChanged(_ sender: UITextField) {
    repassword = sender.text ?? ""
}
```

在 `viewDidLoad` 中完成 4 个 Publisher 的订阅，实现图片的显隐与按钮能否点击的切换。

```
// 按钮，默认是不能点击的
@IBOutlet var loginBtn: UIButton!
// 显示用户名的合法时的提示
@IBOutlet weak var usernameInfo: UIImageView!
// 显示密码的合法时的提示
@IBOutlet weak var passwordInfo: UIImageView!
// 显示2次密码的一致时的提示
@IBOutlet weak var repasswordInfo: UIImageView!

// 存储订阅者防止释放
var cancellables: Set<AnyCancellable> = []

override func viewDidLoad() {
    super.viewDidLoad()

    // 1.验证用户名合法
    validatedUsername.map{ $0 == nil }
        .receive(on: RunLoop.main)
        .assign(to: \.isHidden, on: usernameInfo)
        .store(in: &cancellables)

    // 2.验证密码的合法性
    valiatedPassword.map{ $0 == nil }
        .assign(to: \.isHidden, on: passwordInfo)
        .store(in: &cancellables)
```

```
    // 3.验证两次数输入的密码是否一致
    valiatedRepassword.map{ $0 == nil }
        .assign(to: \.isHidden, on: repasswordInfo)
        .store(in: &cancellables)

    // 4.检查用户名和密码
    validatedAccount.map{ $0 != nil }
        .receive(on: RunLoop.main)
        // 使用 Assign 订阅者改变 UI 状态
        .assign(to: \.isEnabled, on: loginBtn)
        .store(in: &cancellables)
}
```

完整代码

```
import UIKit
import Combine

class ViewController: UIViewController {
    // 属性发布者
    @Published var username: String = ""
    @Published var password: String = ""
    @Published var repassword: String = ""

    // 按钮，默认是不能点击的
    @IBOutlet var loginBtn: UIButton!
    // 显示用户名的合法时的提示
    @IBOutlet weak var usernameInfo: UIImageView!
    // 显示密码的合法时的提示
    @IBOutlet weak var passwordInfo: UIImageView!
    // 显示2次密码的一致时的提示
    @IBOutlet weak var repasswordInfo: UIImageView!

    // 存储订阅者防止释放
    var cancellables: Set<AnyCancellable> = []

    // 三个输入框的Editing Changed，每次输入后进行赋值
    @IBAction func usernameChanged(_ sender: UITextField) {
        username = sender.text ?? ""
    }

    @IBAction func passwordChanged(_ sender: UITextField) {
        password = sender.text ?? ""
    }

    @IBAction func repasswordChanged(_ sender: UITextField) {
        repassword = sender.text ?? ""
    }

    override func viewDidLoad() {
        super.viewDidLoad()

        // 1.验证用户名合法
        validatedUsername.map{ $0 == nil }
            .receive(on: RunLoop.main)
            .assign(to: \.isHidden, on: usernameInfo)
            .store(in: &cancellables)

        // 2.验证密码的合法性
        valiatedPassword.map{ $0 == nil }
            .assign(to: \.isHidden, on: passwordInfo)
            .store(in: &cancellables)

        // 3.验证两次数输入的密码是否一致
        valiatedRepassword.map{ $0 == nil }
            .assign(to: \.isHidden, on: repasswordInfo)
            .store(in: &cancellables)

        // 4.检查用户名和密码
        validatedAccount.map{ $0 != nil }
            .receive(on: RunLoop.main)
            // 使用 Assign 订阅者改变 UI 状态
            .assign(to: \.isEnabled, on: loginBtn)
```

```
            .store(in: &cancellables)
    }
}

// 1.检验用户名的Publisher
extension ViewController {
    // 1.1 提交给服务器判断用户名是否合法，网络请求等异步行为
    func usernameChecked(_ username:String, completion: @escaping ((Bool) -> ())) {
        // 模拟网络验证的过程
        DispatchQueue.global().asyncAfter(deadline: .now() + 1.0) {
            if username == "123456" {
                completion(true)
            }
            else {
                completion(false)
            }
        }
    }

    // 1.2 第一个：验证用户名是否合法
    var validatedUsername: AnyPublisher<String?, Never> {
        // 限制产生值的频率
        return $username.debounce(for: 0.5, scheduler: RunLoop.main)
            .removeDuplicates() // 去重，重复的不需要再次检验
            .flatMap { username in
                // 使用 Future 包装已有的异步操作
                return Future { promise in
                    self.usernameChecked(username) { available in
                        promise(.success(available ? username : nil))
                    }
                }
        }.eraseToAnyPublisher()
    }
}

// 2.检验密码的Publisher
extension ViewController {
    // 2.1 第二个：验证密码长度是否大于6
    var valiatedPassword: AnyPublisher<String?, Never> {
        return $password.flatMap{ password in
            return  Just(password.count > 6 ? password : nil)
        }.eraseToAnyPublisher()
    }

    // 2.2 第三个：验证两次密码是否一致
    var valiatedRepassword: AnyPublisher<String?, Never> {
        // 注意这里合并的不是$password而是上一步的valiatedPassword
        return valiatedPassword.combineLatest($repassword).flatMap{ (password, repassword) in
            return Just(password == repassword ? password : nil)
        }.eraseToAnyPublisher()
    }
}

// 3.检验用户名和密码的Publisher
extension ViewController {
    // 第四个：验证用户名和密码
    var validatedAccount: AnyPublisher<(String, String)?, Never> {
        // 合并第一步和第二步产生的Publisher
        validatedUsername.combineLatest(valiatedRepassword).map({ (username, password) -> (String, String)? in
            guard let uname = username, let pwd = password else { return nil }
            return (uname, pwd)
        }).eraseToAnyPublisher()
    }
}
```

效果图

SwiftUI实现

创建ObservableObject（开发的核心）

SwiftUI 由于出现了 `Form` 这种新的 View，对于表单禁止提供了原生的支持，所以实现起来可以比 UIKit 少 1 个 Publisher，构建如下的 3 个 Publisher。

1. 检验用户名的 Publisher
2. 检验密码长度的 Publisher
3. 检验两次密码是否一致的 Publisher

```
class UserAccount: ObservableObject {
    @Published var username: String = ""
    @Published var password: String = ""
    @Published var repassword: String = ""

    // 提交给服务器判断用户名是否合法，网络请求等异步行为
    func usernameChecked(_ username:String, completion: @escaping ((Bool) -> ())) {
        // 模拟网络验证的过程
        DispatchQueue.global().asyncAfter(deadline: .now() + 1.0) {
            if username == "123456" {
                completion(true)
            }
            else {
                completion(false)
            }
        }
    }

    // 第一个：验证用户名是否合法
    var validatedUsername: AnyPublisher<String?, Never> {
        // 限制产生数据的频率
        return $username.debounce(for: 0.5, scheduler: RunLoop.main)
            .removeDuplicates() // 去重，重复的不需要再次检验
            .flatMap { username in
                // 使用 Future 包装已有的异步操作
                return Future { promise in
                    self.usernameChecked(username) { available in
                        promise(.success(available ? username : nil))
                    }
                }
            }.receive(on: RunLoop.main)
                .eraseToAnyPublisher()
    }

    // 第二个：验证密码长度是否大于6
    var valiatedPassword: AnyPublisher<String?, Never> {
        return $password.flatMap{ password in
            return  Just(password.count > 6 ? password : nil)
        }.eraseToAnyPublisher()
    }

    // 第三个：验证两次密码是否一致
    var valiatedRepassword: AnyPublisher<String?, Never> {
        // 注意这里合并的不是$password而是上一步的valiatedPassword
        return valiatedPassword.combineLatest($repassword).flatMap{ (password, repassword) in
            return Just(password == repassword ? password : nil)
        }.eraseToAnyPublisher()
    }
```

```
}
```

输入绑定

将 ObservableObject 中的 @Published 属性绑定到输入的内容上，这样输入的内容就成为了最初的 Publisher。

```
TextField("用户名", text: self.$userAccount.username)
SecureField("密码", text: self.$userAccount.password)
SecureField("确认密码", text: self.$userAccount.repassword)
```

构造界面

采用 From 构造界面，由于 Image 是动态控制的，所以在 Text 上完成 Publisher 的订阅，并根据 3 次订阅的数据设置 Image 的透明度来控制显隐。同时，需要将 3 个订阅数据组合绑定到 Section 的 `disabled` 上来控制按钮能否点击。

```
var body: some View {
    Form {
        Section {
            HStack {
                TextField("用户名", text: self.$userAccount.username)
                    .onReceive(userAccount.validatedUsername.map{ $0 != nil }) {
                        valid in
                        self.unameCondition = valid
                    }

                Image(systemName: "hand.thumbsup.fill")
                    .foregroundColor(Color.green)
                    .opacity(unameCondition ? 1 : 0)
            }

            HStack {
                SecureField("密码", text: self.$userAccount.password)
                    .onReceive(userAccount.valiatedPassword.map{ $0 != nil }) {
                        valid in
                        self.pwdCondition = valid
                    }

                Image(systemName: "hand.thumbsup.fill")
                    .foregroundColor(Color.green)
                    .opacity(pwdCondition ? 1 : 0)
            }

            HStack {
                SecureField("确认密码", text: self.$userAccount.repassword)
                    .onReceive(userAccount.valiatedRepassword.map{ $0 != nil }) {
                        valid in
                        self.rePwdCondition = valid
                    }

                Image(systemName: "hand.thumbsup.fill")
                    .foregroundColor(Color.green)
                    .opacity(rePwdCondition ? 1 : 0)
            }

            Section {
                Button("注册") {
                }
            }.disabled(validation) // 只要不满足条件按钮点击不了
        }
    }
}
```

完整代码

```
import SwiftUI
import Combine

class UserAccount: ObservableObject {
    @Published var username: String = ""
    @Published var password: String = ""
    @Published var repassword: String = ""
```

```swift
// 提交给服务器判断用户名是否合法，网络请求等异步行为
func usernameChecked(_ username:String, completion: @escaping ((Bool) -> ())) {
    // 模拟网络验证的过程
    DispatchQueue.global().asyncAfter(deadline: .now() + 1.0) {
        if username == "123456" {
            completion(true)
        }
        else {
            completion(false)
        }
    }
}

// 第一个：验证用户名是否合法
var validatedUsername: AnyPublisher<String?, Never> {
    // 限制产生数据的频率
    return $username.debounce(for: 0.5, scheduler: RunLoop.main)
        .removeDuplicates() // 去重，重复的不需要再次检验
        .flatMap { username in
            // 使用 Future 包装已有的异步操作
            return Future { promise in
                self.usernameChecked(username) { available in
                    promise(.success(available ? username : nil))
                }
            }
    }.receive(on: RunLoop.main)
        .eraseToAnyPublisher()
}

// 第二个：验证密码长度是否大于6
var valiatedPassword: AnyPublisher<String?, Never> {
    return $password.flatMap{ password in
        return  Just(password.count > 6 ? password : nil)
    }.eraseToAnyPublisher()
}

// 第三个：验证两次密码是否一致
var valiatedRepassword: AnyPublisher<String?, Never> {
    // 注意这里合并的不是$password而是上一步的valiatedPassword
    return valiatedPassword.combineLatest($repassword).flatMap{ (password, repassword) in
        return Just(password == repassword ? password : nil)
    }.eraseToAnyPublisher()
}
}


struct ContentView: View {
    @ObservedObject var userAccount: UserAccount = UserAccount()

    // 保存三个Publisher的订阅数据
    @State private var unameCondition: Bool = false
    @State private var pwdCondition: Bool = false
    @State private var rePwdCondition: Bool = false

    // 按钮的验证条件
    var validation: Bool {
        return !unameCondition || !pwdCondition || !rePwdCondition
    }

    var body: some View {
        Form {
            Section {
                HStack {
                    TextField("用户名", text: self.$userAccount.username)
                        .onReceive(userAccount.validatedUsername.map{ $0 != nil }) {
                            valid in
                            self.unameCondition = valid
                        }

                    // 根据条件控制Image的透明度来达到显隐
                    Image(systemName: "hand.thumbsup.fill")
                        .foregroundColor(Color.green)
                        .opacity(unameCondition ? 1 : 0)
```

```
        }

        HStack {
            SecureField("密码", text: self.$userAccount.password)
                .onReceive(userAccount.valiatedPassword.map{ $0 != nil }) {
                    valid in
                    self.pwdCondition = valid
            }

            Image(systemName: "hand.thumbsup.fill")
                .foregroundColor(Color.green)
                .opacity(pwdCondition ? 1 : 0)
        }

        HStack {
            SecureField("确认密码", text: self.$userAccount.repassword)
                .onReceive(userAccount.valiatedRepassword.map{ $0 != nil }) {
                    valid in
                    self.rePwdCondition = valid
                }

                Image(systemName: "hand.thumbsup.fill")
                    .foregroundColor(Color.green)
                    .opacity(rePwdCondition ? 1 : 0)
            }

            Section {
                Button("注册") {
                }
            }.disabled(validation) // 只要不满足条件按钮点击不了
        }
    }
}
```

效果图

Conclusion

Combine 与 SwiftUI 同时在 WWDC 2019 推出，是一门非常新的技术，二者是未来 iOS/iPadOS/macOS 开发的一对黄金搭档。由于新技术变化比较快，所以随着 Apple 官方的不断更新，本教程也会同步更新。

iOS开发课程

Combine 学习的时候，需要掌握 Swift 语法、熟悉基于 UIKit 或 SwiftUI 的 iOS 开发知识，针对这些知识作者也发布了相应的视频教程，详情请查看 iOS 开发系列教程。

源代码

本教程配套源代码下载地址

Appendix I

常见Publisher的使用

```
import Combine
import UserNotifications

var subscriptions = Set<AnyCancellable>()

// MARK: - Just
Just(1)
    .sink(
        receiveCompletion: { print("completion: \($0)") },
        receiveValue: { print("value: \($0)") })
    .store(in: &subscriptions)

print("----------------------------------")

Just(2)
    .prepend(Just(1))
    .append(Just(3))
    .sink(
        receiveCompletion: { print("completion: \($0)") },
        receiveValue: { print("value: \($0)") })
    .store(in: &subscriptions)

print("----------------------------------")

// MARK: - Empty
// completeImmediately: true
Empty<String, Never>()
    .sink(
        receiveCompletion: { print("completion: \($0)") },
        receiveValue: { print("value: \($0)") })
    .store(in: &subscriptions)

Just(1)
    .append(Empty(completeImmediately: false)) // 任何Publisher追加都不会正常结束
    .sink(
        receiveCompletion: { print("completion: \($0)") },
        receiveValue: { print("value: \($0)") })
    .store(in: &subscriptions)

print("----------------------------------")

// MARK: - Fail
enum MyError: Error {
    case fail
}

Fail<String, Error>(error: MyError.fail)
    .sink(
        receiveCompletion: { print("completion: \($0)") },
        receiveValue: { print("value: \($0)") })
    .store(in: &subscriptions)

print("----------------------------------")

// MARK: - Optional
Optional.Publisher(1)
    .sink(
        receiveCompletion: { print("completion: \($0)") },
        receiveValue: { print("value: \($0)") })
    .store(in: &subscriptions)

// nil时不会接收任何值
Optional.Publisher(nil)
    .sink(
        receiveCompletion: { print("completion: \($0)") },
        receiveValue: { print("Combine Optional") })
```

```
    .store(in: &subscriptions)

print("--------------------------------")

// MARK: - Sequence
(1 ... 10).publisher
    .sink(
        receiveCompletion: { print("completion: \($0)") },
        receiveValue: { print("value: \($0)") })
    .store(in: &subscriptions)

print("--------------------------------")

// MARK: - Deferred
// 一般与Future配合使用
Deferred {
    Future<Bool, Error> { promise in
        UNUserNotificationCenter
            .current()
            .requestAuthorization(options: [.alert, .sound, .badge]) { granted, error in
                if let error = error {
                    promise(.failure(error))
                } else {
                    promise(.success(granted))
                }
            }
    }
}
.sink(
    receiveCompletion: { print("completion: \($0)") },
    receiveValue: { print("value: \($0)") })
.store(in: &subscriptions)

print("--------------------------------")

// MARK: - Record
// 方式一
Record<Int, Error>(output: [1, 2, 3], completion: .finished)
    .sink(
        receiveCompletion: { print("completion: \($0)") },
        receiveValue: { print("value: \($0)") })
    .store(in: &subscriptions)

// 方式二
Record<Int, Error> { record in
    record.receive(1)
    record.receive(2)
    record.receive(3)
    record.receive(completion: .failure(MyError.fail))
}
.sink(
    receiveCompletion: { print("completion: \($0)") },
    receiveValue: { print("value: \($0)") })
.store(in: &subscriptions)

// 方式三
Record(recording: Record<Int, Error>(output: [1, 2, 3], completion: .finished).recording)
    .sink(
        receiveCompletion: { print("completion: \($0)") },
        receiveValue: { print("value: \($0)") })
    .store(in: &subscriptions)

print("--------------------------------")

// MARK: - Share
let scan = Timer
    .publish(every: 1, on: .main, in: .default)
    .autoconnect()
    .scan(0) { count, _ in // 累加,count为闭包最后一次返回的值
        count + 1
    }.share()

scan
    .sink {
        print("a:", $0)
```

```
    }
    .store(in: &subscriptions)

DispatchQueue.main.asyncAfter(deadline: .now() + 3) {
    scan
        .sink {
            print("b:", $0)
        }
        .store(in: &subscriptions)
}

print("--------------------------------")

// MARK: - Multicast
let scan2 = Timer
    .publish(every: 1, on: .main, in: .default)
    .autoconnect()
    .scan(0) { count, _ in // 累加,count为闭包最后一次返回的值
        count + 1
    }.multicast(subject: PassthroughSubject<Int, Never>())

scan2
    .sink {
        print("c:", $0)
    }
    .store(in: &subscriptions)

DispatchQueue.main.asyncAfter(deadline: .now() + 3) {
    scan2
        .sink {
            print("d:", $0)
        }
        .store(in: &subscriptions)

    scan2
        .connect()
        .store(in: &subscriptions)
}

// MARK: - Result
Result<String, Error>.Publisher(.success("Success"))
    .sink(
        receiveCompletion: { print("completion: \($0)") },
        receiveValue: { print("value: \($0)") })
    .store(in: &subscriptions)

Result<Void, Error>.Publisher(.failure(MyError.fail))
    .sink(
        receiveCompletion: { print("completion: \($0)") },
        receiveValue: { print("value: \($0)") })
    .store(in: &subscriptions)

print("--------------------------------")
```