Apple 官方异步编程框架：Swift Combine 简介

06/25/2022 21:13 下午 posted in *apple (apple.html)*

> WWDC19 Session 722 – Introducing Combine (https://developer.apple.com/videos/play/wwdc2019/722/)

引言

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），根据事件去让界面变化。而对异步事件的处理，会让代码和状态变得复杂。而现有的 Cocoa 框架中，异步编程的接口主要有以下这几种：

- Target/Action
- NotificationCenter
- KVO
- Callbacks

而在实际情况中，由于不同的第三方库，系统框架，业务代码可能采用不一样的方式处理异步事件，会导致对事件的处理分散且存在差异。苹果为了帮助开发者简化异步编程，发布了 Swift 的异步编程框架 – Combine。

What is Combine

> “A unified, declarative API for processing values over time”

统一、声明式、为处理变化的值而生的 API。

Combine 作用是将异步事件通过组合`事件处理操作符`进行自定义处理。

关注如何处理变化的值，正是响应式编程所考虑的。也可以说，Combine 是一个苹果官方的 Swift `响应式`框架。

> 响应式编程（Reactive Programming）：面向异步数据流的编程思想。业界比较知名的响应式框架是 ReactiveX 系列。Rx 也有 Swift 版本的 RxSwift。

Combine 特性

由于 Combine 是一个 Swift 编写的框架，所以 Combine 可以受益于 Swift 的一些语言特性。

泛型支持

Combine 享受 Swift 泛型带来的便利性。泛型可以帮助开发者提取更多模板代码，这也意味着我们可以让异步操作的代码支持泛型，然后适配到各个种类的异步操作中。

类型安全

同样受惠于 Swift，可以让编译器和 Runtime 帮助我们检查类型安全问题。

组合优先

Combine 的主要设计理念，使用组合。组合的优点是可以将核心设计得简单又便于理解，但当放在一起使用时，能产生 1 + 1 > 2 的效果。

请求驱动

> 请求驱动(Request Driven)：基于请求和响应的设计思想，消费者向生产者请求某个事务的变化，当变化时生产者给消费者对应的响应。

> 事件驱动(Event Driven)：基于事件通知的设计思想。在事务发生变化时，生产者将通知提交给事件管道进行分发，而不关心谁去消费事件。消费者需要到事件管道中订阅关心的通知。

Combine 是基于请求和响应的设计思想的，这允许你更精准的控制 App 的内存使用和性能。(这一块苹果没有详细解释)

Combine 核心

Combine 框架有三个核心概念

- 发布者(Publisher)
- 订阅者(Subscriber)
- 操作符(Operator)

发布者(Publisher)

发布者在 Combine 框架中是一个协议：

```
public protocol Publisher {

    /// 产生的值的类型
    associatedtype Output

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

    /// 实现这个方法，将调用 `subscribe(_:)` 订阅的订阅者附加到发布者上
    func receive<S>(subscriber: S) where S : Subscriber, Self.Failure == S.Fa
}

extension Publisher {
    /// 将订阅者附加到发布者上，供外部调用，不直接使用 `receive(_:)` 方法
    public func subscribe<S>(_ subscriber: S) where S : Subscriber, Self.Failu
}
```

发布者定义了*如何描述产生的值和错误*，通过定义关联类型 `Output` 和 `Failure` 的实际类型（当发布者不产生错误时，可以使用 `Never`）。由于发布者不需要实际产生值和错误，所以我们可以用值类型来定义它，也就是 Swift 里的结构体。发布者提供让订阅者注册的能力，通过实现 receive 方法。

发布者可以适配到现有的很多异步操作接口中。

官方 Cocoa 框架中 `NotificationCenter` 接口适配发布者的例子（~~截止至 beta1 版本的 Xcode，此 API 尚未开放~~ beta2 版本已支持）：

```
extension NotificationCenter {
        struct Publisher: Combine.Publisher {
                typealias Output = Notification
                typealias Failure = Never
                init(center: NotificationCenter, name: Notification.Name, ob
        }
}
```

订阅者（Subscriber）

和发布者配对的，就是订阅者。同样的，在 Combine 框架中是一个协议：

```
public protocol Subscriber : CustomCombineIdentifierConvertible {

    /// 接受到的值的类型
    associatedtype Input

    /// 可能接受到的错误的类型
    associatedtype Failure : Error

    /// 告诉订阅者，它在发布者上被成功订阅，可以请求值了
    func receive(subscription: Subscription)

    /// 告诉订阅者，发布者产生值了
    func receive(_ input: Self.Input) -> Subscribers.Demand

    /// 告诉订阅者，发布者已经终止产生值了，不管是正常情况还是由于错误情况
    func receive(completion: Subscribers.Completion<Self.Failure>)
}
```

订阅者定义了*如何描述接受的值和错误*，类似的，通过定义关联类型 `Input` 和 `Failure`。由于订阅者在接受到值之后，经常会影响和修改某些状态，所以我们使用引用类型来定义它，也就是 Swift 里的类类型。

订阅者有三个核心方法：

- 接收到订阅的消息
- 接收到产生的值的消息
- 接收到产生已经终止的消息

> 订阅消息(Subsciption):描述如何控制发布者到订阅者的数据流动，用于表达发布者和订阅者之间的连接。

官方的提供 Assign 的订阅者例子（可以在文档中找到更详细的定义）：

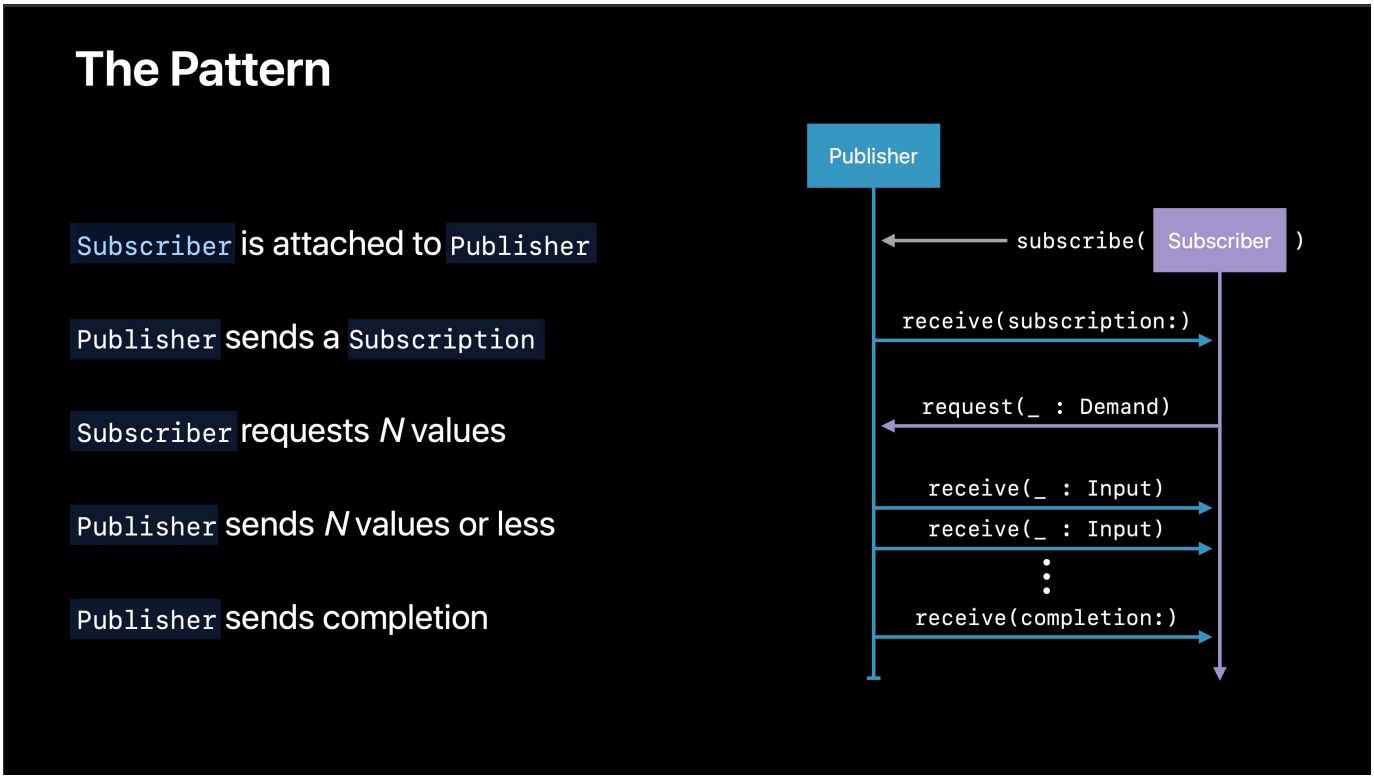
```
extension Subscribers {

    final public class Assign<Root, Input> : Subscriber, Cancellable {

        public typealias Failure = Never
        public init(object: Root, keyPath: ReferenceWritableKeyPath<Root, In
                        //...
    }
}
```

运算符（Operator）

当有了发布者和订阅者后，它们之间是怎么配合起来运作的呢，如下图所示：



- 首先订阅者通过 subsribe 方法附加到发布者上
- 发布者发送一个订阅消息给订阅者，订阅者通过 receive 方法响应
- 发布者通过定义在订阅消息中的 request 方法发送请求需要的值
- 当请求的值产生时，发布者将值发送给订阅者，订阅者通过 receive 方法收到响应

- 最后当发布者终止产生值时，发送完成消息给订阅者，订阅者通过 receive 方法收到响应

看起来一切都配合的很美好，好像这两个概念已经可以实现我们的功能了。当你开开心心敲下这个例子的时候：

```
class Foo {
    var name:String
    init(name:String) {
        self.name = name
    }
}

let object = Foo(name:"Test")
let publisher = NotificationCenter.Publisher(center: .default, name: "Send",
let subsciber = Subscribers.Assign(object: object, keyPath: \.name)

publisher.subscribe(subsciber)
```

你会发现编译不过，因为 NotificationCenter 发布者产生的值类型是 `Notification`，而订阅者需要接受 name 的 String 类型。这时候，操作符就应运而生了。

操作符是一个桥梁，定义一些便捷的操作，让发布者和订阅者能搭配使用，而这样就可以让发布者和订阅者的耦合程度降低，通用程度变高。而复用的发布者和订阅者可以通过一到多个操作符进行适配。

操作符定义如何进行*值的转换*。操作符是遵循 Publisher 协议的，从上游的发布者订阅值，生成新的发布者，处理后并发送给下游的订阅者。

官方提供的 Map 操作符的例子：

```
extension Publishers {
    public struct Map<Upstream, Output> : Publisher where Upstream : Publish
        public typealias Failure = Upstream.Failure
        public let upstream: Upstream
        public let transform: (Upstream.Output) -> Output
        public func receive<S>(subscriber: S) where Output == S.Input, S : S
    }
}
```

但在一般使用时，我们不直接生成操作符发布者，而是使用官方提供的声明式操作符 API。

比如 Map 操作符的方法如下：

```
extension Publishers.Map {
    public func map<T>(_ transform: @escaping (Output) -> T) -> Publishers.M
}
```

Combine 框架中，有以下几类声明式操作符 API：

1. 函数式转换

比如 `map`、`filter`、`reduce` 等函数式思想里的常见的高阶函数的操作符。

2. 列表操作

比如 `first`、`drop`、`append` 等在产生值序列的中使用便捷方法的操作符。

3. 错误处理

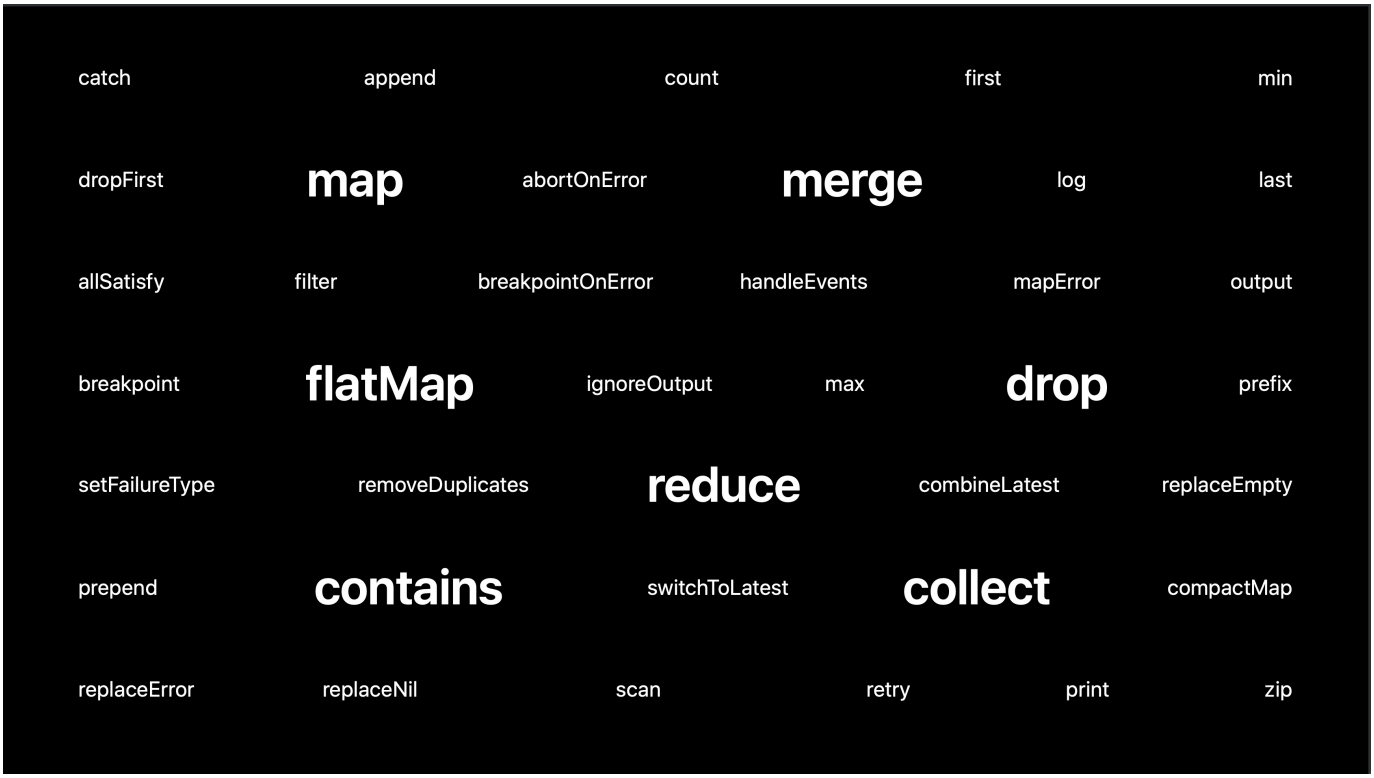
比如 `catch`，`retry` 等进行错处理的操作符。

4. 线程/队列行为

比如 `subscribeOn`，`receiveOn` 等对订阅和接受时线程进行指定的操作符。

5. 调度和时间处理

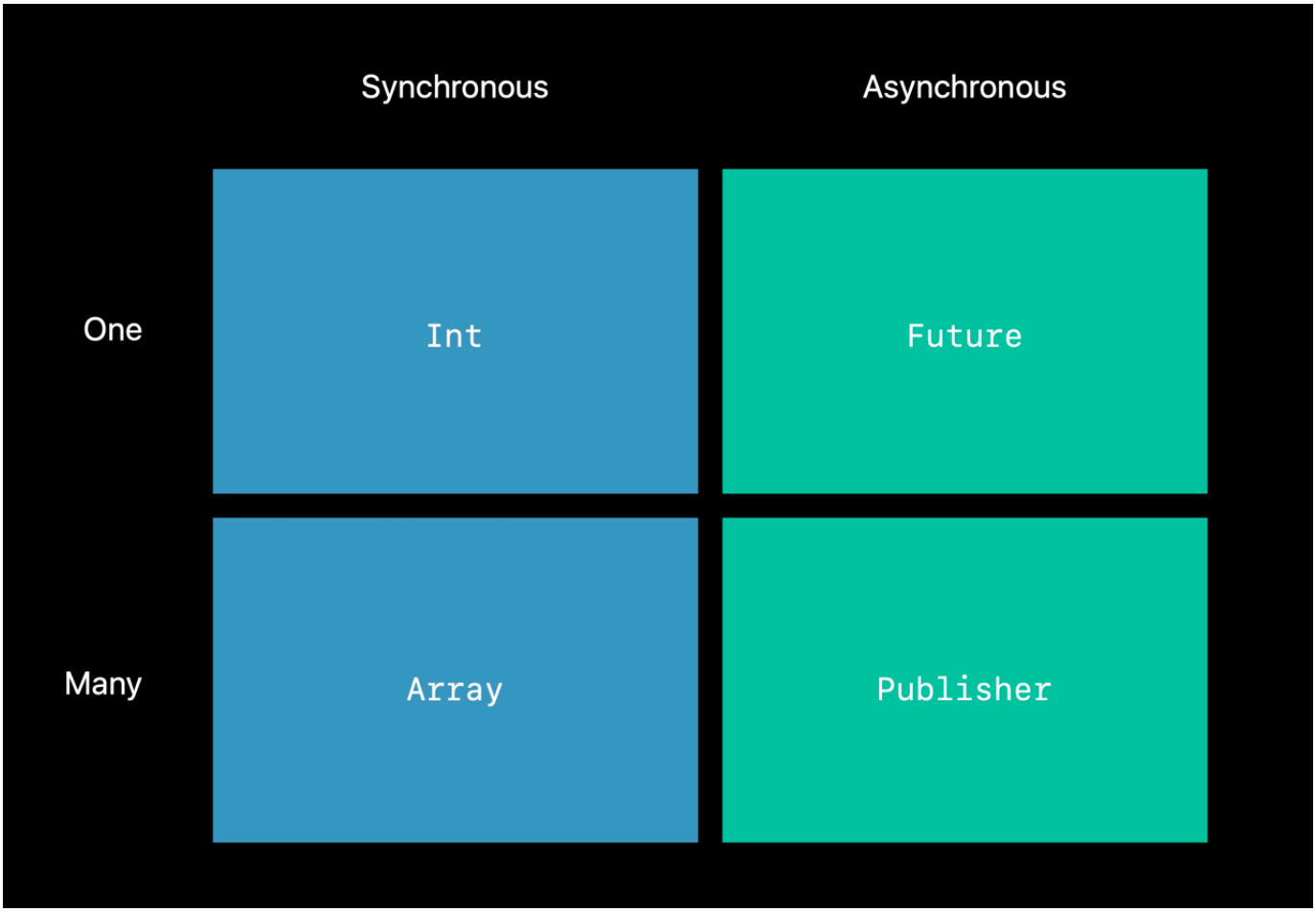
比如 `delay`，`debounce`（去抖动），`throttle`(节流) 等操作符。



Combine 设计理念 - 组合优先

Combine 的设计理念就是优先是使用*组合*。

如果把 Combine 的思想对应到同步编程里面的概念，就是这样下面的图。

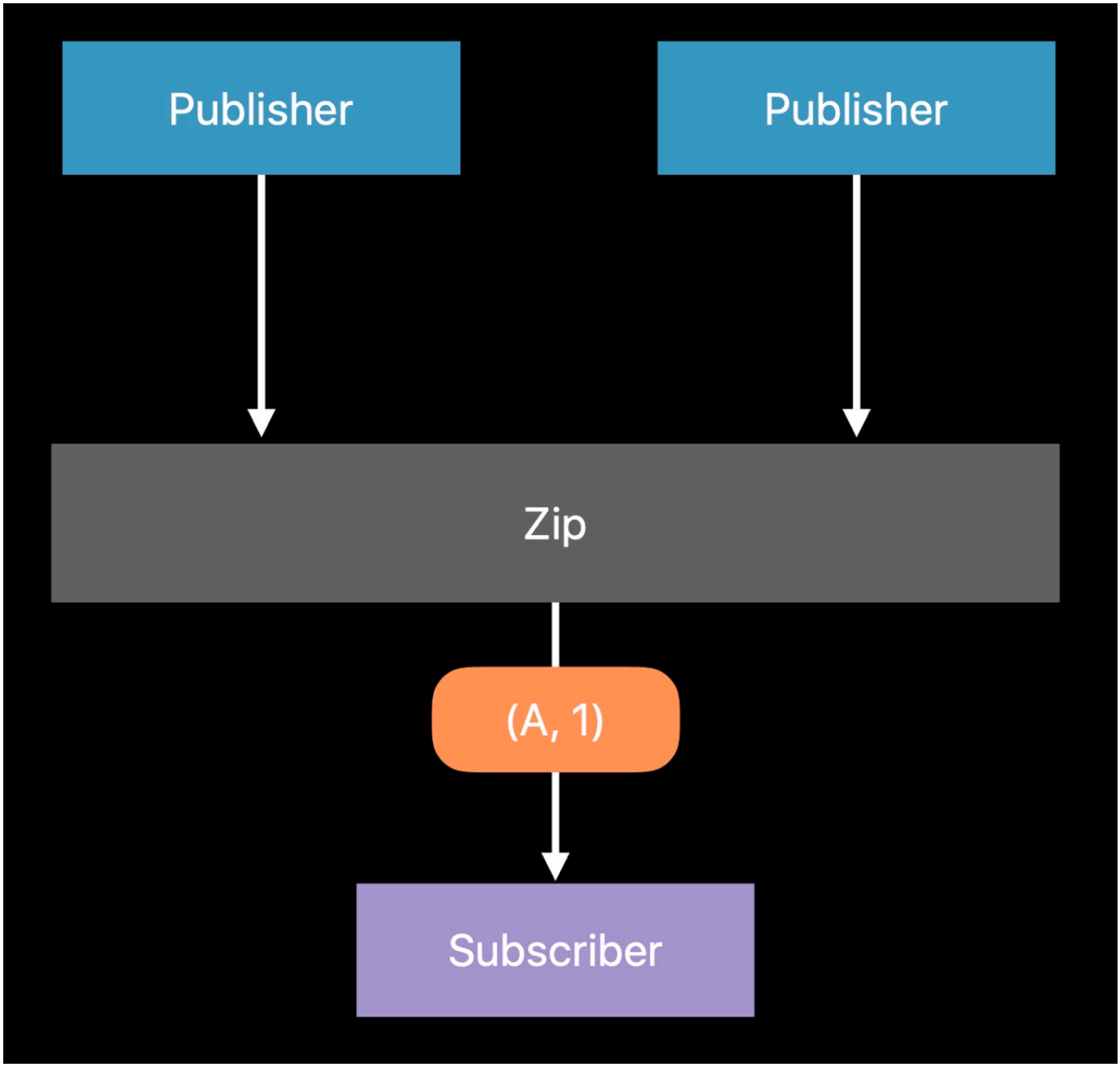


也就是发布者可以使用的操作符，大部分和 Foundation 框架里 Collection 类型的操作符相似，举个例子：我们知道可以对集合类型进行 filter 操作，那么 Combine 操作符里也有对应的 filter 方法。通过方法链（Method Chaning），组合大量的操作符，可以让异步操作更加统一，代码更加整洁。

组合多个发布者

有时候，我们希望将一些异步操作汇合，并统一处理响应。组合多个发布者有两种方式：Zip 和 CombineLastest。

Zip

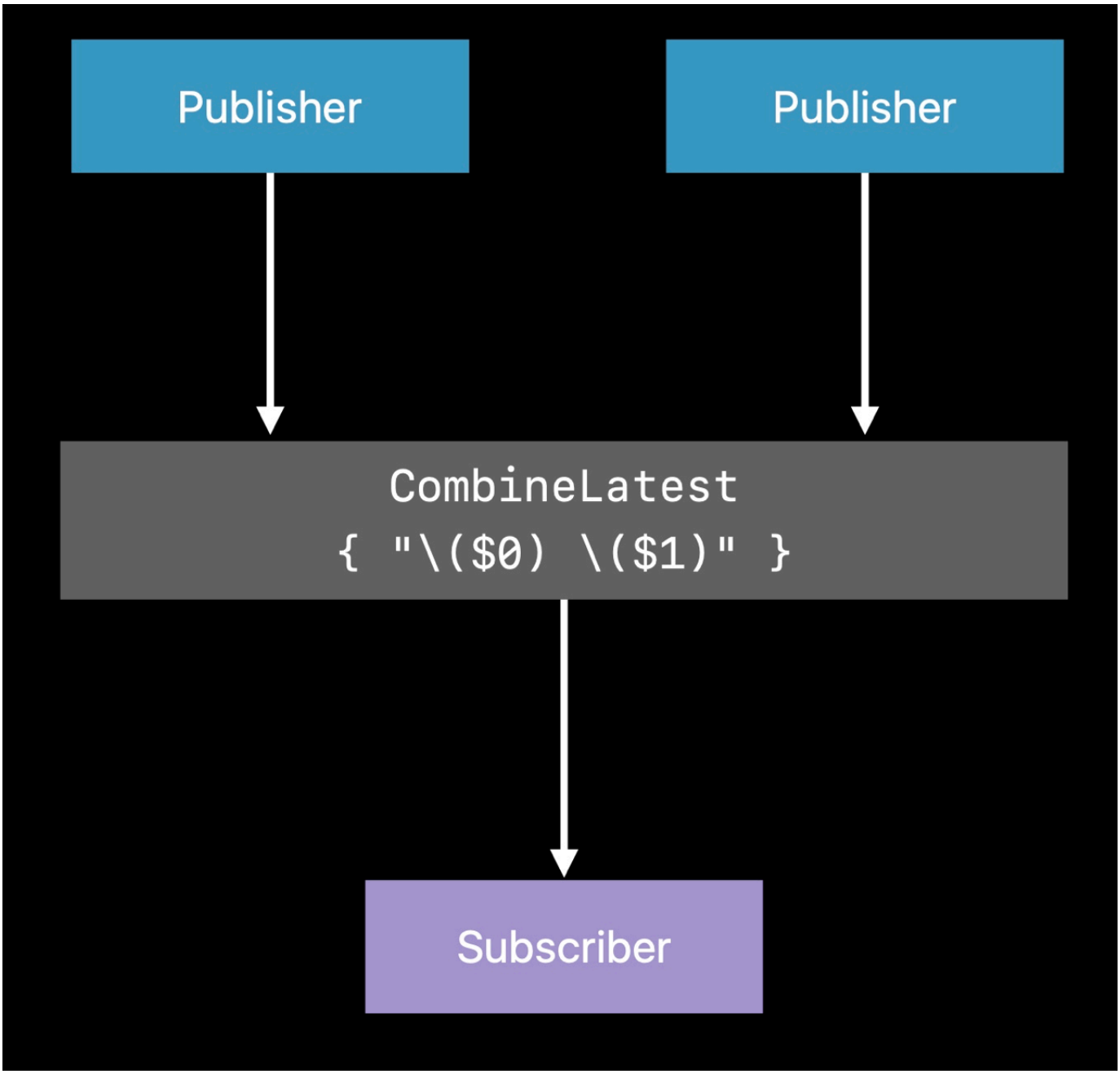


Zip 在 Combine 框架里面是一个结构体(同时还有 Zip3，Zip4，用于更多数量的组合)

```
extension Publishers {
    public struct Zip<A, B> : Publisher where A : Publisher, B : Publisher,
        public typealias Output = (A.Output, B.Output)
        public typealias Failure = A.Failure
        public let a: A
        public let b: B
        public init(_ a: A, _ b: B)
        public func receive<S>(subscriber: S) where S : Subscriber, B.Failur
    }
}
```

Zip 可以通过传入两个发布者进行初始化，要求多个组合的发布者的的错误类型一致，而输出是多个组合的发布者合并起来的元组。Zip 的意思是，当组合的每一个发布者都产生值的时候，才会将值合并成元组发送给订阅者。

Combine Lastest



类似的是，CombineLastest 在 Combine 框架里面也是一个结构体(同理还有 CombineLatest3，CombineLatest4，用于更多数量的组合)

```
extension Publishers {
    public struct CombineLatest<A, B, Output> : Publisher where A : Publishe
        public typealias Failure = A.Failure
        public let a: A
        public let b: B
        public let transform: (A.Output, B.Output) -> Output
        public init(_ a: A, _ b: B, transform: @escaping (A.Output, B.Output
        public func receive<S>(subscriber: S) where Output == S.Input, S : S
    }
}
```

`CombineLatest` 使用两个发布者加上一个 transform 的转换闭包(在闭包中将两个产生的值处理并返回)进行初始化(~~截止目前 beta1 版本的 Xcode，苹果还没将声明公开 init 方法，预计可能还会调整，这一部分来自官方幻灯片中的例子，有待订正~~beta 2 版本已支持)，同样也要求多个组合发布者的错误类型一致，输出是 transform 闭包里的 Output 类型。CombineLastest 的意思是，当多个发布者中任意一个发布者产生值时，都会执行 transform 闭包的操作并将结果发送给订阅者。

总结

Combine 作为官方出品的响应式框架，无疑是令人惊喜的。整个框架的思想虽然现有的响应式框架(如 RxSwift）差不多，但无论是接口上隐藏掉序列，冷热信号等理解难度较大的概念，还是更方便的事件设计，清晰简洁的命名规范，都觉得 Apple 很用心的在打磨这个框架。同时，最具杀伤力的的是，Apple 可以让他和 Cocoa 框架更紧密的结合，打造出类似 UIKit+Combine，Foundation+Combine 的官方支持，这也意味着 Combine 有望成为类似 JavaScript 里面 Promise 这样的规范特性。如果你错过了 ReactiveCocoa，也错过了 RxSwift，也没使用过响应式编程，那么我强烈建议你了解一下今年的 Combine 框架。而剩下最大的问题，是国内什么时候能够 iOS 13 Only 和迁移到 Swift 了。

延伸阅读

Apple 官方异步编程框架：Swift Combine 应用 (https://nemocdz.github.io/post/apple-%E5%AE%98%E6%96%B9%E5%BC%82%E6%AD%A5%E7%BC%96%E7%A8%8B%E6%A1%86%E6%9E%B6swift-combine-%E5%BA%94%E7%94%A8/)

RXSwift 和 Combine 对应关系速查表 (https://github.com/freak4pc/rxswift-to-combine-cheatsheet)

WWDC 19 专栏文章目录 (https://xiaozhuanlan.com/topic/8362954017)

« Combine之自定义Publisher (165622363640 57.html)

Apple 官方异步编程框架：Swift Combine 应用 » (16561663057076.html)

COCOA开发者

Cocoa开发者是一个拾遗补缺的iOS移动端开发博客，记录一些疑难问题和日记。

CATEGORIES

Linux (Linux.html) **BlockChain** (BlockChain.html) **HTTP** (HTTP.html) **UIKit** (UIKit.html)
apple (apple.html) **杂七杂八** (mark.html) **Markdown&MWeb** (Markdown&MWeb.html)
Cocoapods (cocoapods.html) **Application** (Application.html)
ReactiveCocoa (reactiveCocoa.html) **Xcode** (Xcode.html) **Runtime** (runtime.html)
内核编程 (advancedProgram.html) **HTML** (Hybrid.html) **GCD** (GCD.html)
Thread (thread.html) **TextKit** (TextKit.html) **DLNA** (DLNA.html)
Foundation (Foundation.html) **Program** (Program.html) **VPN** (VPN.html)

Filter (Filter.html) **BlockChain** (BlockChain-1.html) **Aliyun** (Aliyun.html) **Git** (Git.html)
RunLoop (RunLoop.html) **Python** (Python.html) **Crypto** (Crypto.html)
Raspbian (raspbian.html) **Flutter** (Flutter.html)

RECENT POSTS

Swift 中 Protocol 和 泛型 (16826799395306.html)

动态库转静态库 (16821712527050.html)

必须由子类重写的Swift类方法 (16724512448053.html)

『ios』不常用的__attribute__ (16656479283288.html)

代码自动格式化 (16655678716160.html)