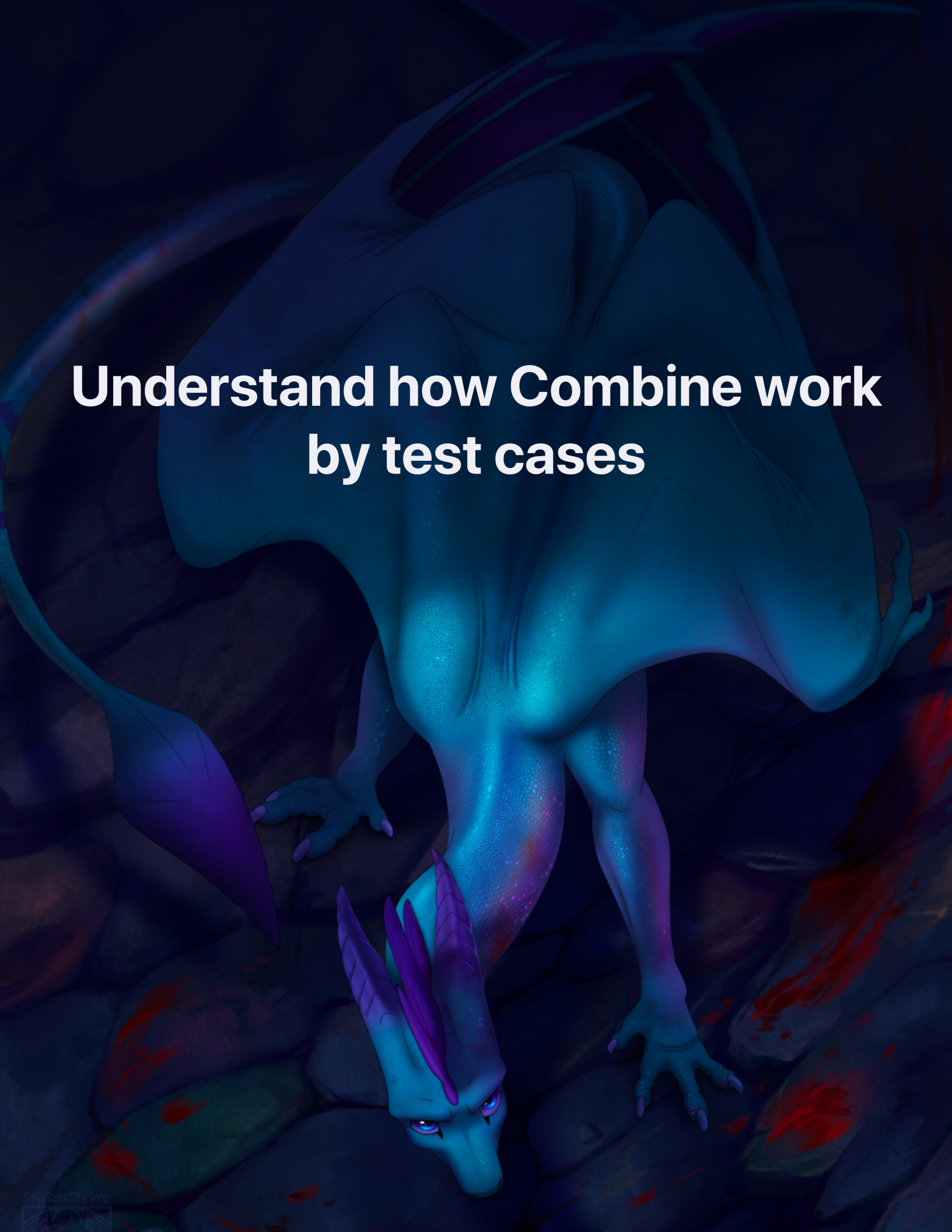
Understand how Combine work by test cases



用测试用例探索 Combine 的工作机制

Mars

這本書的網址是 http://leanpub.com/combine-research-by-test-cases

此版本發布於 2019-12-04

Contents

写在前面

欢迎下载并阅读这本小册子。

在这里，我们会尝试用测试用例探寻 Combine[1] 的工作机制。而整理这些内容的动力，则源于开发泊学[2] App 的时候，把代码迁移到 Combine 过程中时遇到的一个 Bug。至于这个 Bug 本身，我们会在这个册子的最后才向大家揭晓，因为只有到那个时候，你可能才真的能理解我在说什么。相信我，如果你希望尝试使用 Combine 开发一些应用，理解我们接下来要讨论的东西真的可以为你节约很多时间。

当然，这不是一个 Combine 的入门教程，所以，确保你在继续之前，至少要体验过 Combine，如果你还有 RxSwift[3] 的开发经验，就更好了:)

关于代码和影片

这本册子中用到的全部代码，都在这里[4]。

除了文字内容之外，我们还为每一章内容录制了影片，它们也是完全免费的。影片的链接，会放在每一章的开始，大家可以直接点击观看（无需注册）。

---

[1] https://developer.apple.com/documentation/combine
[2] boxueio.com
[3] https://github.com/ReactiveX/RxSwift
[4] https://github.com/puretears/CustomCombine

Chapter One Combine 中的订阅和发布模型

影片地址

观察 Combine 的一些边缘情况[5]

Combine 中的发布者和订阅者

而我们的探索过程，就从理解事件发布和订阅，这个最基本的模型开始。在 Combine 里，表达事件发布概念的是一个叫做 `Publisher` 的协议，经验上，这应该就是一个可以用不同方式不断产生值的对象。但事实上，并不如此。因为它并没有约束任何和“发送事件”这个概念有关的接口。它唯一约束的，就是一个接受“订阅者”的方法：

```
protocol Publisher {
  associatedtype Output
  associatedtype Failure : Error

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

这是 Combine 中第一个和我们响应式编程过往经验不同的地方，**`Publisher`** 仅仅是某种可以接受订阅者的对象。那么，在 `Publisher` 的定义里，那个表达订阅者的 `Subscriber` 又是什么呢？它约束了一组接收各种不同类型事件的接口：

[5] https://boxueio.com/series/build-boxue-app-in-mvvm/episode/618

```
protocol Subscriber : CustomCombineIdentifierConvertible {
  associatedtype Input
  associatedtype Failure : Error

  func receive(_ input: Self.Input) -> Subscribers.Demand
  func receive(completion: Subscribers.Completion<Self.Failure>)
  func receive(subscription: Subscription)
}
```

从功能上说，`receive(_ input:)` 负责从 `Publisher` 接收事件值。`receive(completion:)` 负责从 `Publisher` 接收完成事件。至于 `receive(subscription:)` 我们一会儿再说。

看到这，我们不难猜想，既然 `Publisher` 只有接受订阅者的接口，而 `Subscriber` 又约束了接收事件的接口，Combine 中的事件流最有可能的实现方式，就是发布者直接以“点对点”的形式把事件发送到订阅者。而这个过程，应该和最后的 `receive(subscription:)` 方法有着密切的联系。

观察 Combine 的事件订阅模型

了解了这两个类型之后，我们再从订阅 `Publisher` 的结果进一步观察下 Combine 中发布和订阅模型。

准备工作

首先，做一些准备工作。稍后，我们会自己实现一些概念版的 Combine 基础组件。为此，我们可以创建一个 Mac OS framework 项目，叫做 CustomCombine，并让它包含单元测试。在这个项目里，添加一个 Subject+Send.swift，在这里给 `Subject` 添加一个扩展，它可以把一个序列的元素，变成 `Subject` 发送的一系列事件。这个方法主要用于方便我们在单元测试中生成测试事件：

```
public extension Subject {
  func send<S: Sequence>(
    sequence: S,
    completion: Subscribers.Completion<Self.Failure>? = nil)
    where S.Element == Self.Output {
    for v in sequence {
      send(v)
    }

    if completion != nil {
      send(completion: completion!)
```

```
    }
  }
}
```

接下来，再创建一个 Subscribers+Event.swift。在这里，添加一个 `enum Event`，它是 Combine 中“事件”这个概念的封装，把订阅者可能接收的事件，统一成了一个类型。之所以要这样做，也是为了稍后在测试用例中，方便观察和比较订阅者接收到的事件。

```
public extension Subscribers {
  enum Event<Value, Failure: Error> {
    case value(Value)
    case complete(Subscribers.Completion<Failure>)
  }
}

extension Subscribers.Event: Equatable
  where Value: Equatable, Failure: Equatable {}
```

然后，在这个文件里，给 `Sequence` 添加一个扩展方法 `asEvents`，让它可以把 `Sequence` 中的元素类型，转换成 `Event` 数组，我们用这个方法，生成单元测试的期望结果：

```
public extension Sequence {
  func asEvents<Failure>(
    failure: Failure.Type,
    completion: Subscribers.Completion<Failure>? = nil)
    -> [Subscribers.Event<Element, Failure>] {
    let values = map(Subscribers.Event<Element, Failure>.value)
    guard let completion = completion else { return values }
    return values + [Subscribers.Event<Element, Failure>.complete(completion)]
  }
}
```

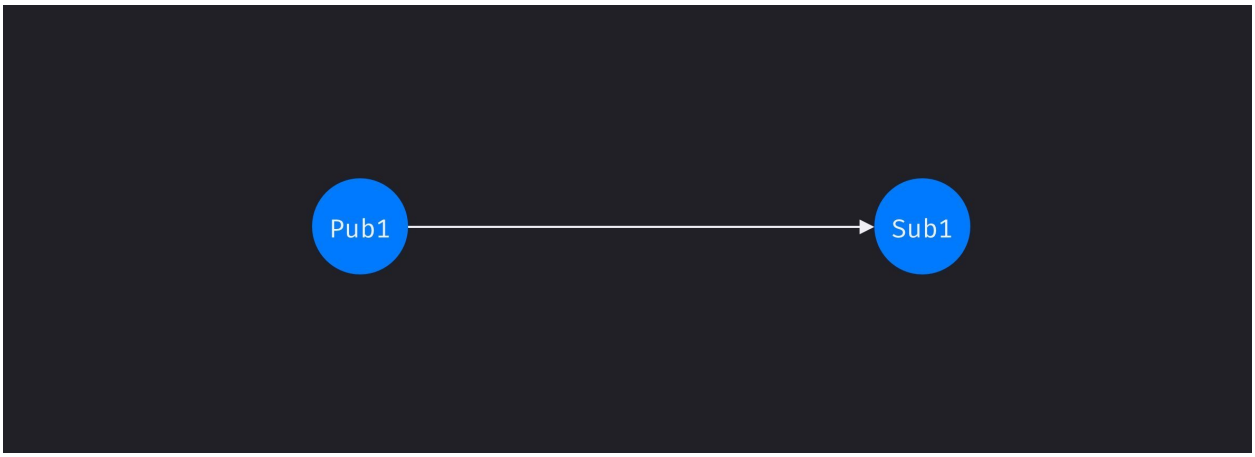
这里，一个很有意思的用法就是，我们可以把 `enum` 中带有关联值的 `case` 直接作为参数传递给 `map`，编译器会把这个 `case` 认为是 `(Value) -> EnumType` 这种类型的函数，在我们的例子中，也就是 `(Value) -> Event<Value, Failure>`。有了这个方法之后，再重载一个错误事件是 `.Never` 的版本。因为在单元测试中，我们主要观察的是整个事件流，而错误的类型通常并不重要。

```
func asEvents(completion: Subscribers.Completion<Never>? = nil)
  -> [Subscribers.Event<Element, Never>] {
  return asEvents(failure: Never.self, completion: completion)
}
```

有了这些方法之后，为这一节内容做的准备工作就完成了。

观察一个最基础的发布订阅

接下来，我们从观察一个最基本的发布订阅模型开始：

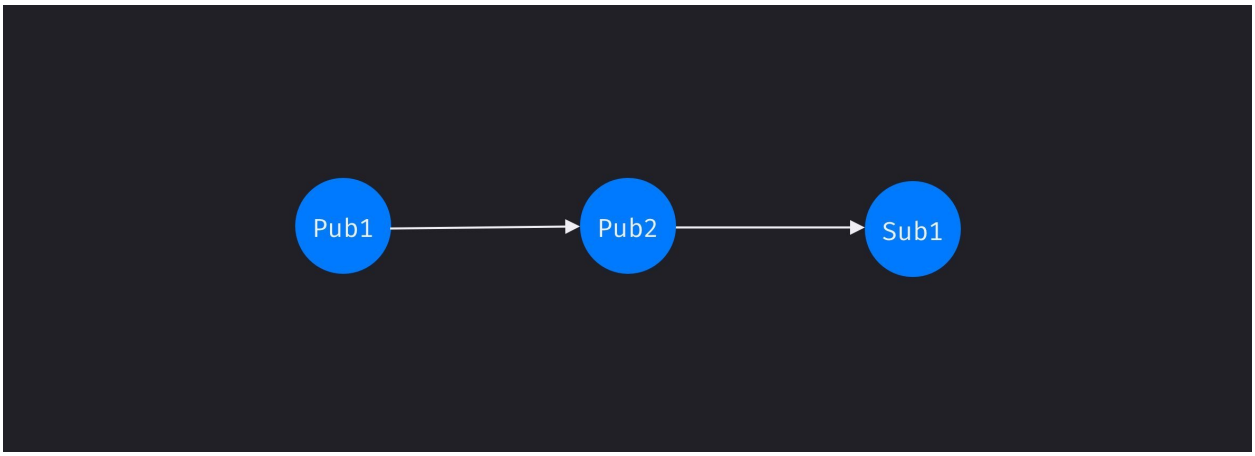


为了表达这个关系，在 CustomCombineTest.swift 里，创建一个测试用例：

```
func testSubjectSink() {
  let subject = PassthroughSubject<Int, Never>()
  var received = [Subscribers.Event<Int, Never>]()
  let sink = Subscribers.Sink<Int, Never>(
    receiveCompletion: {
      received.append(.complete($0))
    },
    receiveValue: {
      received.append(.value($0))
    })

  subject.subscribe(sink)
  subject.send(sequence: 1...3, completion: .finished)

  XCTAssertEqual(received, (1...3).asEvents(completion: .finished))
}
```

这里，事件发布者是 subject，订阅者是 sink。在 subject.subscribe(sink) 之后，我们通过 subject 发布了 1,2,3,.finished 事件（这里用的就是之前给 Sequence 添加的扩展方法）。显然，sink 也应该全数收到这些事件，而这，就是我们在 XCTAssertEqual 中判断的依据。

并且，即便我们像下面这样修改上游事件流，整个事件的发布订阅模型也不会受到影响：



这种关系，在 Combine 里可以用一个 transformer 表示：

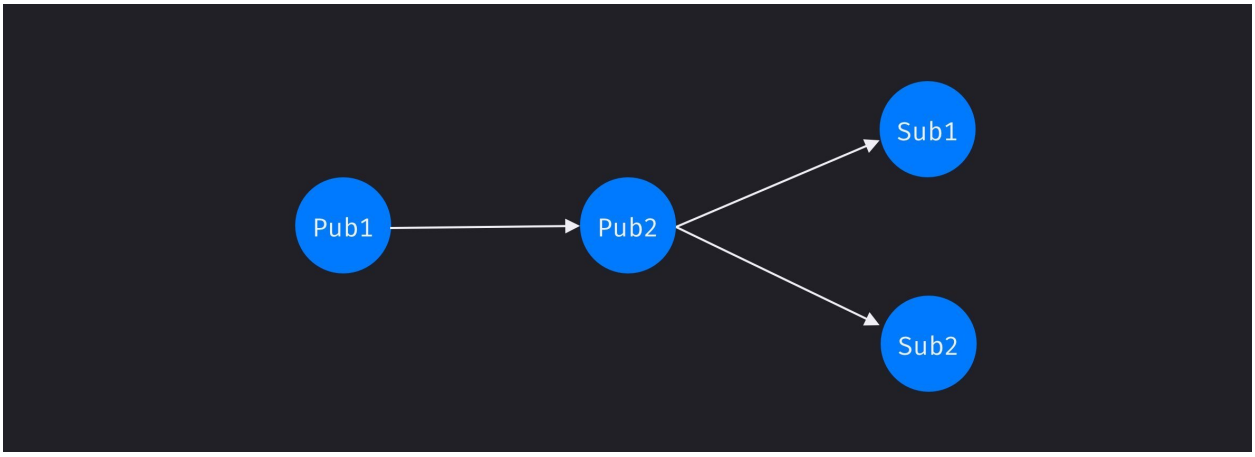
```
func testScan() {
  let subjectA = PassthroughSubject<Int, Never>()
  let scanB = Publishers.Scan(
    upstream: subjectA,
    initialResult: 10, nextPartialResult: +)

  var received = [Subscribers.Event<Int, Never>]()
  let sink = Subscribers.Sink<Int, Never>(
    receiveCompletion: {
      received.append(.complete($0))
    },
    receiveValue: {
      received.append(.value($0))
    })

  scanB.subscribe(sink)
  subjectA.send(sequence: 1...3, completion: .finished)

  XCTAssertEqual(received, [11, 13, 16].asEvents(completion: .finished))
}
```

这里，我们用 Scan 加工了原始数据流，并订阅了加工后的结果。当 subjectA 生成了原始事件之后，我们还是可以全数接受到所有加工后的事件。

至此，“订阅了一个 Publisher 之后，就可以订阅到它的所有事件”这个结论，仍旧是成立的。

半路杀出来个订阅者又如何呢

接下来，我们把订阅模型变成下面这样，在 Pub1 的生命周期里，又多出来一个订阅者会如何呢?



对于这种情况，Sub2 可以订阅到的内容，响应式框架通常会给我们一些选择，例如：

- 订阅到所有缓存的历史事件；
- 订阅到最近的一次事件；
- 只订阅下一次生成的事件；
- 重新触发整个事件订阅流程；

在 Combine 里，默认使用的就是最后一种方式，看似也最不容易理解。我们用下面这个测试用例表达这个模型：

```
1  func testSequenceABCD() {
2   let subjectA = PassthroughSubject<Int, Never>()
3   let scanB = Publishers.Scan(
4    upstream: subjectA, initialResult: 10, nextPartialResult: +)
5
6   var receivedC = [Subscribers.Event<Int, Never>]()
7   let sinkC = Subscribers.Sink<Int, Never>(
8    receiveCompletion: {
9     receivedC.append(.complete($0))
10   },
11   receiveValue: {
12    receivedC.append(.value($0))
```

```
    })

  var receivedD = [Subscribers.Event<Int, Never>]()
  let sinkD = Subscribers.Sink<Int, Never>(
    receiveCompletion: {
      receivedD.append(.complete($0))
    },
    receiveValue: {
      receivedD.append(.value($0))
    })

  scanB.subscribe(sinkC)
  scanB.subscribe(sinkD)
  subjectA.send(sequence: 1...3, completion: .finished)

  XCTAssertEqual(receivedC, [11, 13, 16].asEvents(completion: .finished))
  XCTAssertEqual(receivedD, [11, 13, 16].asEvents(completion: .finished))
}
```

这次，我们给 `scanB` 添加了两个订阅者 `sinkC` 和 `sinkD`。所谓“重新触发整个事件订阅流程”就是指，当 `sinkD` 订阅 `scanB` 时，会驱使 `scanB` 重新对 `subjectA` 中的所有元素进行一次计算，并把结果发送到 `sinkD`，而不是 `sinkC` 消费了 `scanB` 的所有事件之后，`sinkD` 就订阅不到事件了。

因此，“订阅了一个 `Publisher` 之后，就可以订阅到它的所有事件”这个结论，仍旧是成立的。

考虑上时间维度之后呢

当然你可能会觉得，上面的例子不太有说服力，毕竟，`sinkC` 和 `sinkD` 是处在同一个时间点订阅的。因此，我们很难观察到所谓“半路杀出来的订阅者”真实的订阅行为。

emm...没错，我完全认同。而当我们把这两个订阅的时间错开，得到的结论，也让我们可以发现一些有意思的结论。为了演示这个过程，我们创建一个 `Deferred` 对象，并用 `Scan` 加工其中的事件：

```
func testDeferredSubjects() {
  var subjects = [PassthroughSubject<Int, Never>]()
  let deferred = Deferred { () -> PassthroughSubject<Int, Never> in
    let request = PassthroughSubject<Int, Never>()
    subjects.append(request)

    return request
  }

  let scanB = Publishers.Scan(
    upstream: deferred, initialResult: 10, nextPartialResult: +)
}
```

这里，我们用 `subjects` 保存了 `Deferred` 中生成的 `PassthroughSubject` 对象的引用。这是为了稍后方便在其中生成事件。接下来，还是创建两个订阅者 `sinkC` 和 `sinkD`，这部分和上个测试是一样的：

```
func testDeferredSubjects() {
  /// ...

  var receivedC = [Subscribers.Event<Int, Never>]()
  let sinkC = Subscribers.Sink<Int, Never>(
    receiveCompletion: {
      receivedC.append(.complete($0))
    },
    receiveValue: {
      receivedC.append(.value($0))
    })

  var receivedD = [Subscribers.Event<Int, Never>]()
  let sinkD = Subscribers.Sink<Int, Never>(
    receiveCompletion: {
      receivedD.append(.complete($0))
    },
    receiveValue: {
      receivedD.append(.value($0))
    })
}
```

最后，我们先让 `sinkC` 订阅 `scanB`，并让 `scanB` 开始生成事件。然后，再让 `sinkD` 也加入订阅，并通过 `subject[0]` 和 `subject[1]` 继续让 `scanB` 生成事件：

```
scanB.subscribe(sinkC)
subjects[0].send(sequence: 1...2, completion: nil)

scanB.subscribe(sinkD)
subjects[0].send(sequence: 3...4, completion: .finished)
subjects[1].send(sequence: 1...4, completion: .finished)
```

现在，sinkD 订阅了 scanB 之后，它会订阅到通过 subjects[0] 生成的后续事件么？说到这，事情就有点儿矛盾了。按照我们之前说过的“订阅了一个 Publisher 之后，就可以订阅到它的所有事件”这个结论，显然 sinkD 应该可以订阅到 subjects[0] 生成的后续 16,20,.finsihed 事件。但如果这样，它还会订阅到通过 subjects[1] 生成的 11,13,16,20,.finished 事件，一个订阅者可以订阅到两次 .finished 事件显然是不正确的。

而正确的结果是，sinkD 只能订阅到 subject[1] 通过 scanB 生成的事件：

```
XCTAssertEqual(receivedC, [11, 13, 16, 20].asEvents(completion: .finished))
XCTAssertEqual(receivedD, [11, 13, 16, 20].asEvents(completion: .finished))
```

What’s next?

当然，我们的确是通过一些小伎俩刻意放大了我们想达到的效果，即：订阅同一个 Publisher 不一定可以订阅到它后续的所有事件。而我们之前用图片表达的发布者和订阅者的关系，也不一定真实的反应了事件的传递路径，在它们背后，还有着我们没有发觉的内容。而继续探寻的线索，就是 Subscriber 协议中的 func receive(subscription: Subscription) 方法。除了 Publisher 和 Subscriber 之外，这个 Subscription 是做什么的呢？

模拟 Subscription 实现 - I

影片地址

模拟 Subscription 实现 - I[6]

一个完整的订阅发布模型

这一节，我们继续之前的话题。Subscription 究竟是什么呢？它又是如何影响 Combine 的事件发布订阅模型的呢？为了搞清楚这个问题，我们可以从两个方向入手。

- 首先，通过文档和已经公开的代码，整理和补充 Subscription 在事件发布订阅模型中承担的功能，当然这里也需要有一些脑补的环节；
- 其次，用代码自定义一个 Subscription 对象并把它用于 Combine 的单元测试，验证我们的理解；

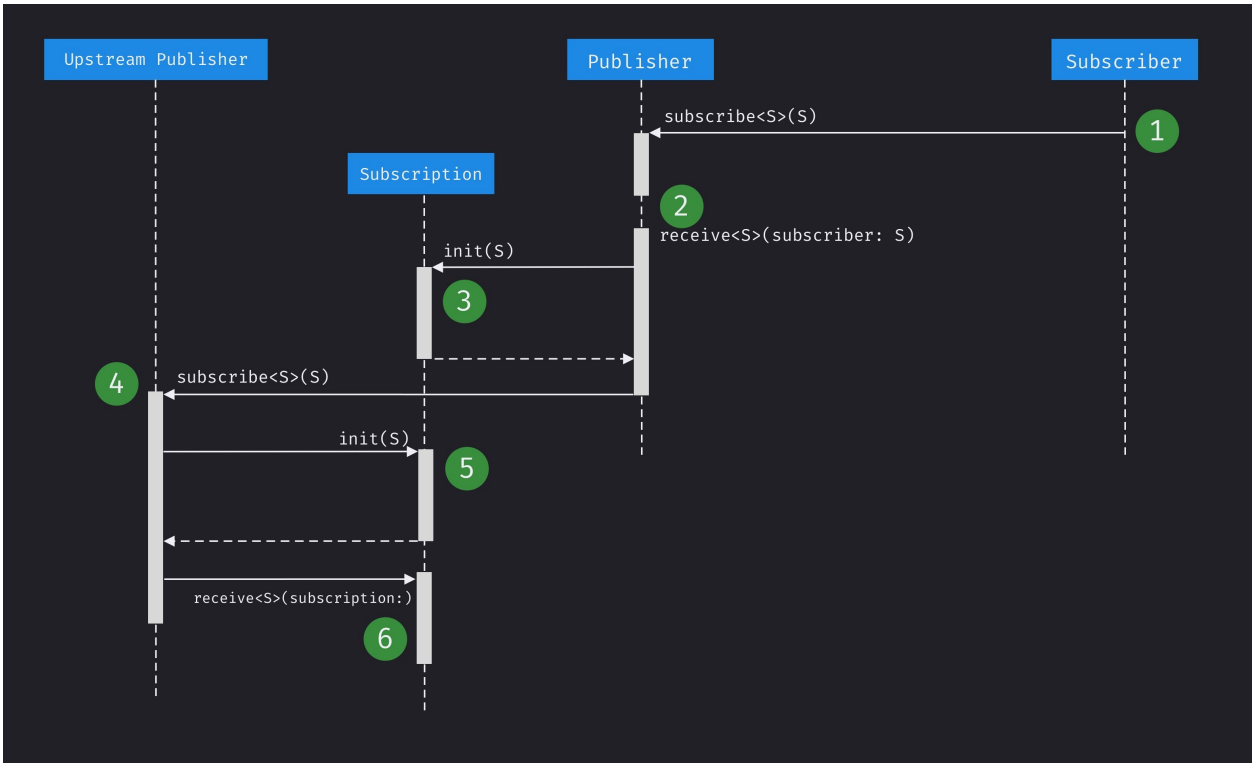
如果直接去翻翻 Subscription 的代码，就会发现，它只约束了一个 request 方法：

```
1  public protocol Subscription
2   : Cancellable, CustomCombineIdentifierConvertible {
3     /// Tells a publisher that it may send more values to the subscriber.
4     func request(_ demand: Subscribers.Demand)
5  }
```

再想想我们上一节整理出来的内容，Subscriber 约束了一个接受 Subscription 参数的 receive 方法，而 Subscription 只约束了一个接受事件订阅数量的 request 方法。它们到底在事件发布订阅模型中，承担了什么角色呢？

为了搞清楚这个问题，在不考虑额外调度的前提下，我们先把一个完整的事件发布订阅模型陈列出来：

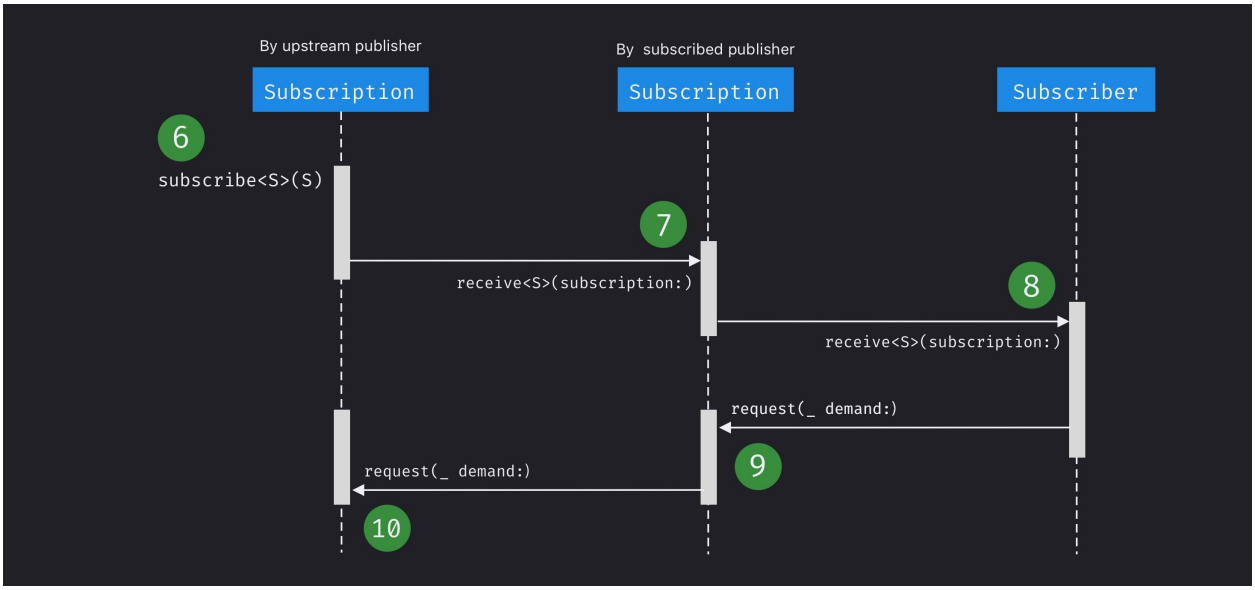
[6] https://boxueio.com/series/build-boxue-app-in-mvvm/episode/619



- 首先，订阅者调用发布者的 `subscribe` 方法，把自己作为参数传递给发布者表示要订阅事件；
- 其次，在 `subscribe` 方法的实现里，会调用 `Publisher` 约束的 `receive<S>(subscriber: S)` 方法处理传入的订阅者；
- 第三，在 `receive` 的实现里，会把传入的订阅者封装成一个 `Subscription` 对象，这个对象会持有原始订阅者的引用；
- 第四，如果被订阅的发布者存在上游发布者，它就会把封装的 `Subscription` 作为订阅者，继续调用上游发布者的 `subscribe` 方法。为了让这一步成立，`Subscription` 也应该是一个实现了 `Subscriber` 的类型；
- 第五，上游发布者会把下游传入的 `Subscription` 对象进一步封装成表示订阅其自身的 `Subscription` 对象，并让它持有指向传入的 `Subscription` 对象的引用；
- 第六，事件流源头的发布者，会调用下级订阅者的 `receive<S>(subscription:)` 方法，传入它封装好的 `Subscription` 对象；

至此，整个发布订阅模型的前半部分就说完了。在整个事件流发布和订阅的链条里，只有最末端的订阅者，是通过 `Subscriber` 表示的，而其余的环节里，维持链条完整的“中间订阅者”都是通过 `Subscription` 表示的。尽管从公开的代码和文档中没有并没有说明 `Subscription` 是一个 `Subscriber`，仅仅说了 `Subscription` is a protocol representing the connection of a subscriber to a publisher。但从要实现的结果，以及 `Publisher` 和 `Subscriber` 各自的接口声明来看，要想维系这种关系，`Subscription` 的确需要有 `Subscriber` 的行为特征才行。

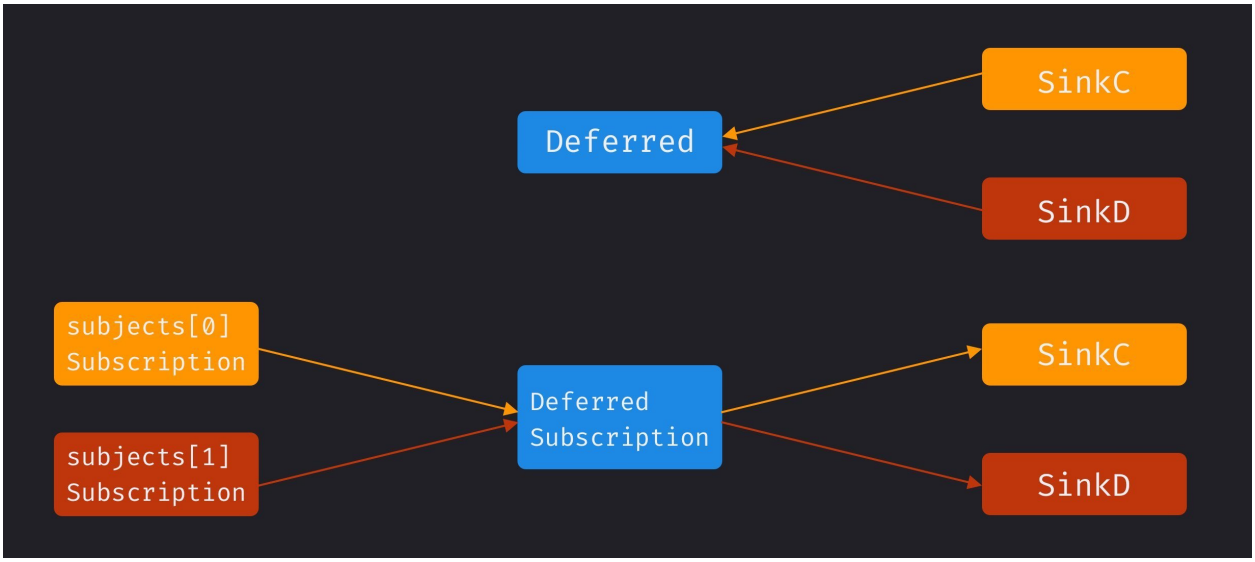
为了进一步巩固这个想法，我们继续来看发布订阅模型的后半部分，把这部分用一张图表示，是这样的：



这次，我们仅仅列出了原始的订阅者，直接订阅的 Publisher 封装的 Subscription，以及上游 Publisher 封装的 Subscription。这个过程紧接者刚才说过的“第六”。

- 第七，上游 Publisher 封装了自己的 Subscription 对象之后，把下游 Publisher 封装的 Subscription 对象当成了一个订阅者，调用了它的 receive<S>(subscription:) 方法；
- 第八，下游 Publisher 再调用原始订阅者的 receive<S>(subscription:)，传递它自己的 Subscription 对象；
- 第九，在原始订阅者的 receive<S>(subscription:) 方法里，调用传入的 Subscription 对象的 request(_ demand:) 方法，表明自己期望订阅到的事件数量；
- 最后，Publisher 继续调用自己上游 Publisher 传入的 Subscription 的 request(_ demand:) 方法，这样在整条事件传递链条上，最终到订阅者的，就只有特定数量的事件了；

以上，就是一个完整的 Combine 事件发送订阅模型。通过第二个图片可以看出，在 Combine 里，通过代码呈现的发布者和订阅者关系的背后，真正发挥作用的，其实是一个 Subscription 链条。说到这，也就不难理解为什么上一节最后一个测试用例中，sinkC 和 sinkD 不能共享 scanB 中的事件了，因为对于这两个订阅者来说，scanB 的上游事件并不相同，把它们的订阅关系转换成 Subscription 对象的关系是这样的：



自定义 Subscription 实现

看到这，你可能会想，虽然从道理上整个逻辑能说得通，但 Combine 中真就如此么？毕竟这里有我们根据文档和方法签名推测出来的东西。而验证这个推测最好的方式，就是自定义一个 `Publisher` 和 `Subscription` 对象，并把它们用在上一节的测试用例中，如果可以通过测试，关于整个模型的推测就得到印证了。

定义 `Subscription` “应有”的行为

首先，我们要自定义一个协议，让它拥有之前我们推测的各种行为。为此，在上一节创建的 framework 项目里，新建个 CustomSubscription.swift。在这里，按照我们对 `Subscription` “需求”定义一个协议：

```
1  public protocol SubscriptionBehavior
2   : class, Cancellable, CustomCombineIdentifierConvertible {
3   associatedtype Input
4   associatedtype Failure: Error
5   associatedtype Output
6   associatedtype OutputFailure: Error
7
8   var demand: Subscribers.Demand { get set }
9   var upstream: Subscription? { get set }
10  var downstream: AnySubscriber<Output, OutputFailure> { get }
11
12  func request(_ d: Subscribers.Demand)
```

```
13   func receive(_ input: Input) -> Subscribers.Demand
14   func receive(completion: Subscribers.Completion<Failure>)
15 }
```

这里，之所以让 SubscriptionBehavior 是一个 class 才能遵守的协议，是因为 Swift 会自动为类生成 CustomCombineIdentifierConvertible 要求的 combineIdentifier 属性，省得我们自己去实现它了，用起来方便一些。

其次，来看关联类型和属性的部分：

- Input 和 Failure 是上游 Publisher 中的普通和错误事件类型；
- Output 和 OutputFailure 是下游 Subscriber 的普通和错误事件类型；
- demand 是下游 Subscriber 向上传递的要订阅的事件数量；
- upstream 是上游 Publisher 下发的 Subscription 对象，由于不一定存在上游 Publisher，因此这是一个 optional；
- downstream 是下游订阅者的引用；

最后，是接口的部分：

- request 就是 Combine 中 Subscription 约束的 request 接口，它收集订阅者期望订阅的事件数量；
- 剩下的两个 receive 方法，让 SubscriptionBehavior 具有 Subscriber 的行为特征，既可以让它接收到 Publisher 的普通和完成事件；

以上，就是我们推测的 Subscription 应该具备的全部行为特征了。然后，根据之前整理的事件发布订阅模型中 Subscription 的功能，我们来给这些接口添加默认实现。

首先是 request，它累计下游订阅者的订阅需求，并把这个需求传到上游发布者：

```
1  public extension SubscriptionBehavior {
2   func request(_ d: Subscribers.Demand) {
3     demand += d
4     upstream?.request(demand)
5   }
6  }
```

其次，是 Cancellable 约束的 cancel 方法，我们这个用于演示的 SubscriptionBehavior 无需做额外的操作，因此直接通知上游发布者就好了：

```
public extension SubscriptionBehavior {
  func cancel() {
    upstream?.cancel()
  }
}
```

第三，是把正常事件转发到订阅者的 `receive` 方法。它执行的逻辑就是 `demand` 有配额就调用订阅者的 `receive` 方法，并更新 `demand`，这里要特别注意 `demand` 的计算方法。它是和下游订阅者的 `receive` 返回值累计的。否则，就返回 `Demand.none`：

```
public extension SubscriptionBehavior
  where Input == Output, Failure == OutputFailure {
  func receive(_ input: Input) -> Subscribers.Demand {
    if demand > 0 {
      let newDemand = downstream.receive(input)
      demand = newDemand + (demand - 1)

      return demand
    }

    return Subscribers.Demand.none
  }
}
```

最后，是接受完成事件的 `receive` 方法。由于我们不需要做额外的处理，这次，就直接把事件转发给下游订阅者就好了：

```
public extension SubscriptionBehavior
  where Failure == OutputFailure {
  func receive(completion: Subscribers.Completion<Failure>) {
    downstream.receive(completion: completion)
  }
}
```

至此，按照我们的推断，一个 `Subscription` 对象应有的行为，就都实现完了。这里，我们只是让 `SubscriptionBehavior` 看上去像一个 `Subscription` 而已，并没有给它一个正式的“名分”，至于为什么这样做，接着往下看就好了。

自定义 `Subscription`

接下来，利用 `SubscriptionBehavior`，我们自定义一个实现了 `Subscription` 的类型。为了线程安全，我们把 `SubscriptionBehavior` 的所有操作，都用一个互斥锁保护起来。还是在 CustomSubscription.swift 里，先来看这个类型的定义：

```
public struct CustomSubscription<Content: SubscriptionBehavior>
  : Subscriber, Subscription {
  public typealias Input = Content.Input
  public typealias Failure = Content.Failure

  public var combineIdentifier: CombineIdentifier {
    return content.combineIdentifier
  }

  let recursiveMutext = NSRecursiveLock()
  let content: Content

  init(behavior: Content) {
    self.content = behavior
  }
}
```

按照之前的推测，一个实现了 `Subscription` 的类型也应该是一个实现了 `Subscriber` 的类型。因此，`CustomSubscription` 是在我们的框架中，有正式名分的 `Subscription` 对象。而之所以要把 `SubscriptionBehavior` 放在泛型参数里，是为了让它专注于处理事件发布和订阅的具体逻辑。而 `CustomSubscription` 则专注于让这些方法保持线程安全性。如果你还有点迷糊，没关系，继续往下看就好了。

首先，我们来实现 `CustomSubscription` 中 `Subscription` 的部分：

```
public func request(_ demand: Subscribers.Demand) {
  recursiveMutex.lock()
  defer { recursiveMutex.unlock() }
  content.request(demand)
}

public func cancel() {
  recursiveMutex.lock()
  content.cancel()
  recursiveMutex.unlock()
}
```

看到了吧，这就是我们说过的区分开执行逻辑和线程安全性的方式。接下来，我们再来实现 `Subscriber` 的部分。

```
public func receive(subscription upstream: Subscription) {
  recursiveMutext.lock()
  defer { recursiveMutext.unlock() }

  content.upstream = upstream
  content.downstream.receive(subscription: self)
}
```

通过这个 `receive(subscription:)` 的实现，我们就知道 `Subscription` 是如何持有上级 `Subscription` 引用，以及如何把自身对象传到下级订阅者的了。

最后，是接受普通和完成事件的 `receive` 方法。它们的实现都很简单，就是在 `recursiveMutex` 的保护下接收事件而已：

```
public func receive(_ input: Input) -> Subscribers.Demand {
  recursiveMutex.lock()
  defer { recursiveMutex.unlock() }
  return content.receive(input)
}

public func receive(completion: Subscribers.Completion<Failure>) {
  recursiveMutex.lock()
  defer { recursiveMutex.unlock() }
  content.receive(completion: completion)
}
```

What's next?

至此，按照我们的推测，一个实现了 `Subscription` 的类型该有的样子，就完成了。在继续之前，回顾下我们对 Combine 事件发布订阅模型的描述，以及 `CustomSubscription` 是如何与这个模型中对应的环节关联起来的。如果你确定已经没问题了，就可以进入下一节的内容了。

模拟 Subscription 实现 - II

影片地址

模拟 Subscription 实现 - II[7]

自定义 Publisher

继续之前的工作，这一节，我们自定义一个 Publisher，并把它用于之前的单元测试，如果一切顺利，测试应该可以通过，这样，我们对 Combine 中事件发布和订阅模型的理解，也就得到了验证。

在之前的项目中，新建一个 CombineSubject.swift。在这里，为了方便稍后测试，我们创建一个类似 PassthroughSubject 行为的类：

```
1  public class CustomSubject<Output, Failure: Error>: Subject {
2   var subscribers: Dictionary<
3     CombineIdentifier, CustomSubscription<Behavior>> = [:]
4  }
```

由于 Subject 是一个从 Publisher 派生而来的协议，因此，我们就不用再声明 CustomSubject 实现 Publisher 了。CustomSubject 有一个 subscribers 属性，它用一个字典保存了所有的订阅者。其中，Key 是当前 Publisher 事件流的标识，Value 是封装了订阅者的 Subscription 对象。

定制 SubscriptionBehavior

那么，上面代码中的 Behavior 是什么呢？按照上一节的描述，它定义的是把 CustomSubject 的订阅者封装成 Subscription 时完成的操作。这里唯一需要定制的，就是当有订阅者取消订阅时，我们要从 CustomSubject.subscribers 删掉对应的 CustomSubscription 对象。可能光看文字有点晕，我们直接来看代码：

[7] https://boxueio.com/series/build-boxue-app-in-mvvm/episode/620

```swift
public class CustomSubject<Output, Failure: Error>: Subject {
  class Behavior: SubscriptionBehavior {
    // 1
    typealias Input = Output

    // 2
    var demand: Subscribers.Demand = .none
    var upstream: Subscription? = nil
    let downstream: AnySubscriber<Output, Failure>

    // 3
    let subject: CustomSubject<Output, Failure>
    init(subject: CustomSubject<Output, Failure>,
      downstream: AnySubscriber<Output, Failure>) {
      self.subject = subject
      self.downstream = downstream
    }

    // 4
    func request(_ d: Subscribers.Demand) {
      demand += d
      upstream?.request(d)
    }

    // 5
    func cancel() {
      subject.subscribers.removeValue(forKey: combineIdentifier)
    }
  }
}
```

按照代码注释中数字的顺序，我们逐个看一下这些段代码。

第一部分，按照我们的想法，`CustomSubject` 不会对事件进行变化，因此，`Input` 和 `Output` 的类型是相同的，我们把 `Output` 定义成 `Input` 的别名，就不用再额外指定 `Input` 的类型了。

第二部分，是 `SubscriptionBehavior` 约束的属性。把它们定义成属性，也就有各自的 `get` 和 `set` 方法了。

第三部分，是 `CustomSubject` 对象的引用。通过这个属性，`Subscription` 才能完成修改 `CustomSubject` 的订阅需求，或者取消订阅的操作。

第四部分，按说，这个 `request` 方法是不需要在这里重新定义一遍的。因为它和 `SubscriptionBehavior` 中的默认实现一样。但在 Xcode 11.2.1 这个版本自带的 Swift 版本上，不定义这个方法会导致 Swift 编译器发生段错误，因此，它和我们要实现的逻辑没什么

关系，纯粹是一个临时的解决方案。

第五部分，就是我们要为 CustomSubject 自定义 Behavior 的根本原因。不同于 Subscription-Behavior 中的默认实现，这里，有订阅者要取消订阅时，我们要把它从 subject.subscribers 中删除。

实现 CustomSubject

定义好了 Behavior 之后，就可以实现 CustomSubject 了。我们先来实现 Publisher 的部分：

```
public func receive<S>(subscriber: S)
  where S : Subscriber, Failure == S.Failure, Output == S.Input {
  let behavior = Behavior(subject: self, downstream: AnySubscriber(subscriber))
  let subscription = CustomSubscription(behavior: behavior)

  subscribers[subscription.combineIdentifier] = subscription
  subscription.receive(subscription: Subscriptions.empty)
}
```

通过这段代码，就能了解当 Publisher 被订阅的时候，究竟发生什么了：

- 首先，通过用 subscriber 创建 Behavior 对象，我们就定义了如何向订阅者“递交”发生的普通和完成事件；
- 其次，通过创建 CustomSubscription 对象，我们演示了如何用 Subscription 封装下有订阅者；
- 最后，调用 receive(subscription:) 方法。由于 CustomSubject.upstream 被我们设置成了 nil，这里，我们无需再向上游事件发布者封装自己的 Subscription 对象。因此，直接传递了 Subscriptions.empty。进而会导致订阅者的 request(_ demand:) 被调用，从而完成整个订阅过程；

完成后，我们来实现 Subject 的部分，首先是一个我们不太常见的 send 方法：

```
public func send(subscription: Subscription) {
  subscription.request(.unlimited)
}
```

坦白讲，我并不清楚这个方法究竟用在什么场景里，我们在之前推测的事件发布订阅模型中也没有提到它，Apple 在官方文档中对这个方法的描述也很模糊。大致的意思，感觉像和 Subscriber.receive(subscription) 是类似的。大家如果有对这个方法更确切的理解，欢迎写邮件给我。既然不了解它，我们就一直订阅 subscription 生成的事件就好了。

其次，是发送普通事件的 send 方法：

```
public func send(_ value: Output) {
  for (_, sub) in subscribers {
    _ = sub.receive(value)
  }
}
```

这个实现很好理解，就是遍历 subscribers 数组，然后通过 receive 方法让订阅者们接收事件就好了。而这种形式，也就是在之前我们猜测的 Publisher 中的事件是“点对点”发送到订阅者的表现形式。

最后，是发送完成事件的 send 方法。大体上，和发送普通事件的 send 逻辑是一样的，只不过，给所有订阅者发送了完成事件之后，要清空 subscribers，这样，就不会再向这些订阅者发送任何消息了：

```
public func send(completion: Subscribers.Completion<Failure>) {
  for (_, sub) in subscribers {
    _ = sub.receive(completion: completion)
  }

  subscribers.removeAll()
}
```

What’s next?

至此，CustomSubject 也就完成了，它当前的问题，就是对 subscribers 的访问并不是线程安全的，因此，还不能在多线程环境里使用共享的 CustomSubject 对象。不过，这个问题并不影响当前的研究，为了保持精力集中，我们稍后再花点时间专门处理线程安全性的问题。

现在，稍微让自己休息一会儿，回顾一下这一节我们为 CustomSubject 自定义 Behavior 的目的，以及 CustomSubject 自身的实现是如何对应到 Combine 事件发布订阅模型的。下一节，我们将仿照这个模式，自定义 Publishers.Scan 的实现。这是在使用单元测试验证整个模型之前，我们要实现的最后一个 Combine 组件。

模拟 Subscription 实现 - III

影片地址

模拟 Subscription 实现 - III[8]

自定义 CustomScan

希望你还记得，之前我们通过测试用例研究 Combine 事件发布订阅模型的时候，创建过一个叫做 `Publishers.Scan` 的对象对上游事件序列中的事件值进行累计。这一节，我们就通过自定义一个类似的 `CustomScan`，来了解这类类型的工作原理。

同样作为一个 `Publisher`，其实实现的过程和上一节的 `CustomSubject` 是一样的。首先，根据它的订阅者接收事件的方式定义一个实现了 `SubscriptionBehavior` 的类型；其次，根据这个类型，实现 `Publisher` 约束的 `receive<S>(subscriber: S)` 就好了。

为此，在项目中，我们创建了一个 CustomScan.swift，在其中添加 `CustomScan` 的实现。先来看 `CustomScan` 的定义：

```
1  public class CustomScan<Upstream: Publisher, Output>: Publisher {
2   public typealias Failure = Upstream.Failure
3   let reducer: (Output, Upstream.Output) -> Output
4   let upstream: Upstream
5   let initialState: Output
6
7   public init(upstream: Upstream,
8    initialResult: Output,
9    nextPartialResult: @escaping (Output, Upstream.Output) -> Output) {
10   self.initialState = initialResult
11   self.reducer = nextPartialResult
12   self.upstream = upstream
13  }
14 }
```

其中，泛型参数 `Upstream` 表示上游事件发布者的类型，`Output` 表示 `CustomScan` 自身转换出来的新事件类型。由于 `CustomScan` 不转换错误事件的类型，所以我们一开始，让 `Failure` 就是上游事件序列中的错误类型。接下来：

[8] https://boxueio.com/series/build-boxue-app-in-mvvm/episode/621

- `reducer` 是累计事件值的计算方法；
- `upstream` 是上游事件序列的引用，由于 `CustomScan` 要依赖于上游事件进行计算，因此这个属性不是一个 optional；
- `initialState` 是计算的初始值；

在 memberwise init 方法里，逐个对这些属性初始化就好了。

自定义 SubscriptionBehavior

接下来，和上一节创建 `CustomSubject` 一样，我们也要给 `CustomScan` 自定义一个 `CustomBehavior` 对象。因为我们要在接收到上游事件之后，进行累加计算。还是先来看它的定义：

```
1  public class CustomScan<Upstream: Publisher, Output>: Publisher {
2   class Behavior: SubscriptionBehavior {
3    typealias Input = Upstream.Output
4    var upstream: Subscription? = nil
5    let downstream: AnySubscriber<Output, Upstream.Failure>
6    var demand: Subscribers.Demand = .none
7    let reducer: (Output, Upstream.Output) -> Output
8    var state: Output
9
10   init(downstream: AnySubscriber<Output, Upstream.Failure>,
11     reducer: @escaping (Output, Upstream.Output) -> Output,
12     initialState: Output) {
13     self.downstream = downstream
14     self.reducer = reducer
15     self.state = initialState
16   }
17
18   func receive(_ input: Upstream.Output) -> Subscribers.Demand {
19     state = reducer(state, input)
20     return downstream.receive(state)
21   }
22  }
23 }
```

可以看到，除了“沿袭”自 `SubscriptionBehavior` 的内容之外，`Behavior` 还持有了 `CustomScan` 的计算方法和初始状态，通过这个定义，我们可以进一步领会在 `Publisher` 背后，`Subscription` 对象的功能。在我们的实现里，上游事件值的累计，是在递交给订阅者的时候，计算出来的。

实现 Publisher 约束的方法

有了 `Behavior` 之后，就可以实现 `CustomScan` 的剩余方法了。也就是 `Publisher` 约束的 `receive<S>(subscriber:)`：

```
public func receive<S>(subscriber: S)
  where S: Subscriber, Failure == S.Failure, Output == S.Input {
  let downstream = AnySubscriber(subscriber)
  let behavior = Behavior(
    downstream: downstream, reducer: reducer, initialState: initialState)
  let subscription = CustomSubscription(behavior: behavior)
  upstream.subscribe(subscription)
}
```

这里，和 `CustomSubject` 唯一不同的就是，我们把包含了 `CustomScan` 事件转换逻辑的 `Subscription` 对象作为一个订阅者，订阅的上游事件发布者。每一个中间环节都像这样订阅订阅上游发布者，整个事件链就通过 `Subscription` 连接起来了。

通过单元测试验证模型

说到这里，用于验证 Combine 事件发布订阅模型需要的核心组件，就都完成了。
为了验证这个模型，我们基于之前写过的 `testDeferredSubjects`，创建一个新的测试用例：

```
func testCustomSubjects() {
  var subjects = [CustomSubject<Int, Never>]()
  let deferred = Deferred { () -> CustomSubject<Int, Never> in
    let request = CustomSubject<Int, Never>()
    subjects.append(request)

    return request
  }

  let scanB = CustomScan(
    upstream: deferred,
    initialResult: 10,
    nextPartialResult: +)

  var receivedC = [Subscribers.Event<Int, Never>]()
  let sinkC = Subscribers.Sink<Int, Never>(
    receiveCompletion: {
      receivedC.append(.complete($0))
```

```
    },
    receiveValue: {
      receivedC.append(.value($0))
    })

  var receivedD = [Subscribers.Event<Int, Never>]()
  let sinkD = Subscribers.Sink<Int, Never>(
    receiveCompletion: {
      receivedD.append(.complete($0))
    },
    receiveValue: {
      receivedD.append(.value($0))
    })

  scanB.subscribe(sinkC)
  subjects[0].send(sequence: 1...2, completion: nil)

  scanB.subscribe(sinkD) /// `sinkD` does not receive events in `subjects[0]`
  subjects[0].send(sequence: 3...4, completion: .finished)
  subjects[1].send(sequence: 1...4, completion: .finished)

  XCTAssertEqual(receivedC, [11, 13, 16, 20].asEvents(completion: .finished))
  XCTAssertEqual(receivedD, [11, 13, 16, 20].asEvents(completion: .finished))
}
```

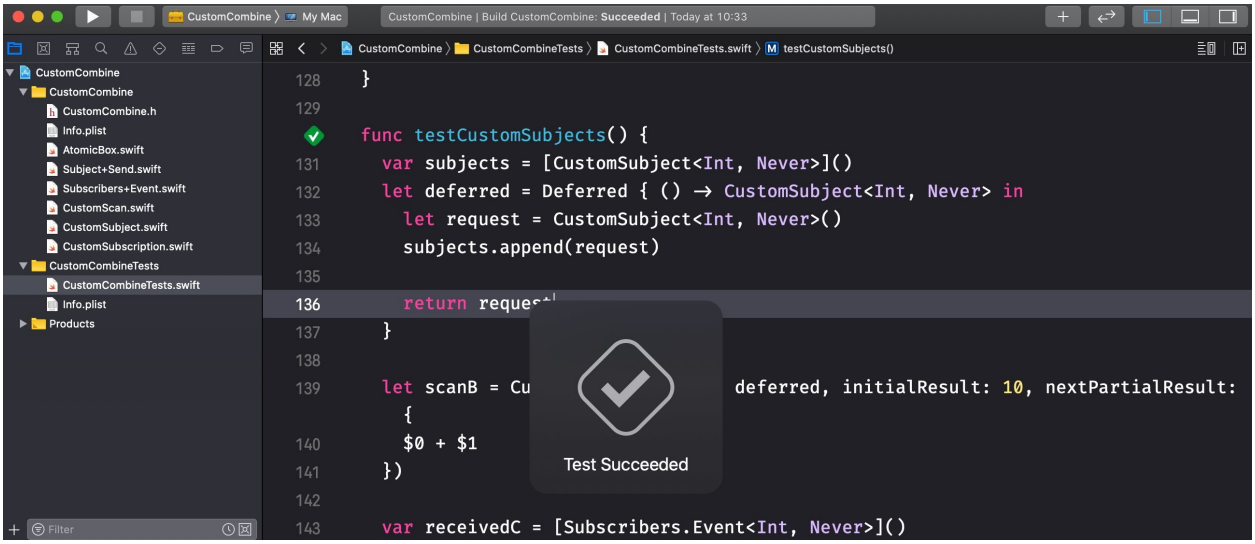
这次，我们让 `deferred` 包含了两个 `CustomSubject` 对象。其余的部分，和 `testDeferredSubjects` 是完全相同的，如果一切顺利，测试的结果也应该和使用 `PassthroughSubject` 是一样的。不过，在开始自行测试之前，先来想个问题：为什么这个测试用例可以验证我们对模型的推测呢?

首先，`CustomSubject` 作为一个 `Publisher`，我们定义了它的 `receive(subscriber:)` 方法。但 `sinkC` 和 `sinkD` 订阅它的时候，`subscribe` 方法仍旧是 Combine 提供的，如果订阅之后 `receive(subscriber:)` 被调用了，就验证了这个方法被调用的时机；

其次，在 `receive(subscriber:)` 的实现里，`receive(subscription: Subscriptions.empty)` 应该会导致封装着原始订阅者的 `Subscription` 中的 `request` 方法被调用，也就是我们在 `CustomSubscription` 中定义的 `request(_ demand:)` 方法；

最后，`CustomSubscription` 是一个值类型，这也就意味着在 `receive<S>(subscriber:)` 方法里，每个订阅者获得的，都是一份属于它自己的 `CustomSubscription` 对象，因此，`sinkC` 和 `sinkD` 订阅到的结果，也应该是独立的；

把这些问题想清楚之后，验证是非常简单的，执行一下这个测试用例就好了，如果一切顺利，它应该是可以 Pass 的：



What's next?

以上，就是对 Combine 基础工作原理方面的研究，现在，对于 Combine 中的事件是如何发送的，如何订阅的，以及 `Publisher`，`Subscriber` 和 `Subscription` 这三个角色在整个模型中的角色，我们应该了解的比较清楚了。

接下来，要探索的，是关于“共享”的话题。就像之前在测试用例分析中说到的，每个订阅者独占的 `Subscription` 对象会导致事件在传递到不同订阅者的时候发生重复计算。但有些时候，我们并不希望如此。如何避免这种问题呢？如何实现类似“一发多收”的效果呢？如何能缓存要下发的事件呢？在 RxSwift 里，我们可以用 `ConnectableObservable<E>`。那么，在 Combine 里，如何实现类似的“共享”功能呢？

让 CustomSubject 线程安全

影片地址

让 CustomSubject 线程安全[9]

AtomicBox

理解了 Combine 的事件发布订阅模型之后，这一节我们对 `CustomSubject` 的实现做一些改进，让它支持在多线程环境中使用。

为了给要访问的对象添加访问保护，在 Swift 中一个常见的做法就是创建一个“箱子”类型，并把要保护的内容作为箱子里的“内容”，这样做的一个好处就是保护逻辑和内容的访问是分开的。为此，在我们的项目中，创建一个 AtomicBox.swift，在其中添加一个 `AtomicBox` 类型：

```
1  public final class AtomicBox<T> {
2   @usableFromInline var mutex = os_unfair_lock()
3   @usableFromInline var unboxed: T
4
5   public init(_ unboxed: T) {
6    self.unboxed = unboxed
7   }
8  }
```

其中：

- `mutext` 是 `AtomicBox` 内部使用的用于线程同步的 mutex 对象；
- `unboxed` 是 `AtomicBox` 中的“内容”，也就是我们要保护的对象；

这里，之所以要把它们用 `@usableFromInline` 修饰，是为了让它们成为这个模块二进制接口的一部分。为了性能方面的考虑，`AtomicBox` 的一部分公开接口会用内联的方式实现，为了在这些接口中使用这些属性，我们应该使用 `@usableFromInline` 修饰它们。

接下来，是 `AtomicBox` 中的读写接口。读取数据是通过 `value` 属性提供的。上锁成功后，返回 `unboxed` 就好了：

[9] https://boxueio.com/series/build-boxue-app-in-mvvm/episode/621

```
@inlinable public var value: T {
  get {
    os_unfair_lock_lock(&mutex)
    defer { os_unfair_lock_unlock(&mutex) }

    return unboxed
  }
}
```

修改 unboxed 的操作，是通过 mutate 方法提供的：

```
@discardableResult @inlinable
public func mutate<U>(_ fn: (inout T) throws -> U) rethrows -> U {
  os_unfair_lock_lock(&mutex)
  defer { os_unfair_lock_unlock(&mutex) }

  return try fn(&unboxed)
}
```

这里，我们实现的逻辑并不是直接对 unboxed 赋值。而是把修改 unboxed 的过程封装成了一个函数 fn。其中，修改 unboxed 的部分用 inout T 表示。这个方法还可以抛出错误，并且返回一个和 T 不同的类型。

当然，错误也好，不同的返回值类型也好，这些都是修改 unboxed 这个动作的某种副作用，并不是指一定会用到它们。因此，我们用 @discardableResult 修饰了 mutate 的返回值，这样，编译器就不会产生未使用返回值的警告了。

最后一个要实现的接口，是判断 AtomicBox 保护的内容是否正在被修改，我们用 os_unfair_lock_trylock 方法尝试上锁，如果锁上了，就表示 unboxed 没人在用，否则，就表示 unboxed 正在被修改：

```
/// A computed property that has conditional statement should not be
/// marked as `@inlinable`
public var isMutating: Bool {
  if os_unfair_lock_trylock(&mutex) {
    os_unfair_lock_unlock(&mutex)
    return false
  }

  return true
}
```

要注意的是，我们并没有使用 @inlinable 修饰 isMutating，这是因为它的实现里包含了 if。通常，我们不对这种带有分之语句的方法使用内联。这样，一个简单的同步读写类就完成了。

CustomSuject 的线程安全性

接下来，我们用它保护 `CustomSubject` 中的资源，也就是它的 `subscribers` 属性。为了尽可能减少对这个属性访问的修改，我们可以先基于 `AtomicBox` 创建一个 property wrapper。

为此，继续在 AtomicBox.swift 里，添加下面的代码：

```
@propertyWrapper
public class Atomic<T> {
 var boxed: AtomicBox<T>
 init(content: T) { self.boxed = AtomicBox<T>(content) }

 public var wrappedValue: T {
  get {
   return boxed.value
  }
 }

 public var projectedValue: AtomicBox<T> {
  get {
   return boxed
  }
 }
}
```

其实，就是 `AtomicBox` 的一层包装，很简单，我们就不多说了。然后，我们就可以用它修饰 `CustomSubject` 中 `subscribers` 的定义了：

```
@Atomic<SubscriberRecords>(content: [:]) var subscribers
```

这样，所有读取 `subscribers` 的代码都无需变动，只需要处理一下修改 `subscribers` 的代码就好了。例如，要把 `cancel` 改成这样：

```
func cancel() {
 subject.$subscribers.mutate {
  $0.removeValue(forKey: self.combineIdentifier)
 }
}
```

也就是说，所有之前直接修改 `subscribers` 的地方，现在要通过 `$subscriber` 先得到内部的 `AtomicBox` 对象，再把修改的逻辑放到 `mutate` 方法的 closure 里。受到影响的代码，还有 `send(completion:)` 和 `receive<S>(subscriber:)`：

```
public func send(completion: Subscribers.Completion<Failure>) {
  for (_, sub) in subscribers {
    _ = sub.receive(completion: completion)
  }

  $subscribers.mutate { $0.removeAll() }
}

// Publisher
public func receive<S>(subscriber: S)
  where S : Subscriber, Failure == S.Failure, Output == S.Input {
  let behavior = Behavior(
    subject: self, downstream: AnySubscriber(subscriber))
  let subscription = CustomSubscription(behavior: behavior)

  $subscribers.mutate {
    $0[subscription.combineIdentifier] = subscription
  }

  subscription.receive(subscription: Subscriptions.empty)
}
```

What's next?

现在，`CustomSubject` 就可以在多线程环境中安全的使用了。下一节，我们继续讨论之前留下的话题：如何在 Combine 中实现共享订阅的话题。

让 CustomSubject 线程安全

影片地址

Combine 中的共享计算[10]

共享 Subject

为了研究 Combine 中的共享订阅，我们要从 `Publisher` 生成事件的两种模式说起。也就是在 RxSwift 中经常被讨论的冷和热。并且，RxSwift 作者在GitHub 上的这篇文档[11]中的描述，也基本上适用于 Combine。

所谓“冷”的 `Publisher`，指的是在有人订阅的时候才生成事件的那一类。每个订阅都会生成一个独立的 `Publisher` 对象，事件发生了也就随之被处理了。这类 `Publisher` 和它们的订阅者的关系是静态的，因此，缓存它们的历史数据是没意义的。

而所谓“热”的 `Publisher`，指的是可以在任何时候，以任何速率主动生成事件的那一类，事件的生成过程与下游订阅者无关。所有订阅者，共享同一个 `Publisher` 对象。这类 `Publisher` 在生命周期内，和它们的订阅者的关系是可以动态变化的。也正因为如此，当新订阅者加入的时候，才有所谓历史事件的处理策略问题。

在之前 `testDeferredSubjects` 中，我们使用了 `Deferred` 演示了“冷” `Publisher` 的行为特性：即每当有订阅者的时候，就创建一个新的 `PassthroughSubject` 生成事件。现在，把这个测试用例改成下面这样，让所有的订阅者共享同一个 `PassthroughSubject`：

```
1  func testSharedSubject() {
2    let subjectA = PassthroughSubject<Int, Never>()
3    let scanB = subjectA.scan(10, +)
4
5    var receivedC = [Subscribers.Event<Int, Never>]()
6    let sinkC = scanB.sink(event: { receivedC.append($0) })
7    subjectA.send(sequence: 1...2, completion: nil)
8
9    var receivedD = [Subscribers.Event<Int, Never>]()
10   let sinkD = scanB.sink(event: { receivedD.append($0) })
11   subjectA.send(sequence: 3...4, completion: .finished)
12
```

[10] https://boxueio.com/series/build-boxue-app-in-mvvm/episode/623
[11] https://github.com/ReactiveX/RxSwift/blob/master/Documentation/HotAndColdObservables.md

```
  XCTAssertEqual(receivedC, [11, 13, 16, 20].asEvents(completion: .finished))
  XCTAssertEqual(receivedD, [13, 17].asEvents(completion: .finished))

  sinkC.cancel()
  sinkD.cancel()
}
```

这次，sinkD 同样是在 subjectA 开始生成事件之后才开始订阅。但从 XCTAssertEqual 的判断条件可以看到，receivedC 中有 4 个事件，receivedD 中只有 2 个。因此，不难推断，sinkC 和 sinkD 共享了同一个 subjectA 事件序列。但如果再仔细看下 receivedD 中的内容，就会发现和我们直觉中的并不相同。sinkD 订阅到的事件为什么不是后续的 16 和 20，而是 13 和 17 呢?

问题就出在了 scan(10, +) 调用上，它返回的 Scan 是一个“冷”Publisher。于是事情就变得拧巴了，尽管 sinkC 和 sinkD 订阅的是同一个事件源头 subjectA，但 Scan 会分别为这两个订阅者生成不同的 Subscription 对象。因此，当 subjectA 生成事件 3 时，给 sinkD 的 Subscription 对象的初始状态仍旧是 10，因此，sinkD 订阅到的第一个事件就是 13 了。简单来说，就是它们共享了同一个 Subject，但是没有共享为它们生成订阅事件的过程。理解了这个道理，你也就对第二个事件是 17 不觉得奇怪了吧:)

怎么样，是不是和我们想象的特别不一样。因此，这种“冷热”的 Publisher 几乎无法直接搭配在一起工作，这是我们特别要注意的事情。为了让 sinkD 得到想象中的结果，我们可以使用 multicast:

```
func testMulticastSubject() {
  let subjectA = PassthroughSubject<Int, Never>()
  let multicastB = subjectA.scan(10, +)
    .multicast { PassthroughSubject() }
    .autoconnect()

  var receivedC = [Subscribers.Event<Int, Never>]()
  let sinkC = multicastB.sink(event: { receivedC.append($0) })
  subjectA.send(sequence: 1...2, completion: nil)

  var receivedD = [Subscribers.Event<Int, Never>]()
  let sinkD = multicastB.sink(event: { receivedD.append($0) })
  subjectA.send(sequence: 3...4, completion: .finished)

  XCTAssertEqual(receivedC, [11, 13, 16, 20].asEvents(completion: .finished))
  XCTAssertEqual(receivedD, [16, 20].asEvents(completion: .finished))

  sinkC.cancel()
  sinkD.cancel()
}
```

这样，就又把 scan 计算的事件值，通过一个共享的 PassthroughSubject 发送了出来。sinkD 收到的，就是我们预期中的 16 和 20 了。这里别忘了对 multicast 的返回结果使用 autoconnect() 方法，这样才可以让 multicastB 开始生成事件。

缓存计算出来的事件

除了共享 Subject 之外，我们也可以共享事件的计算结果。当前版本的 Combine 提供了一个 CurrentValueSubject 完成这个任务。为了了解它的工作方式，我们可以把上一个测试用例中创建 multicastB 的部分改成这样：

```
func testMulticastLatest() {
  let subjectA = PassthroughSubject<Int, Never>()
  let multicastB = subjectA
    .scan(10) { state, next in state + next }
    .multicast { CurrentValueSubject(0) }
    .autoconnect()

  /// ...

  XCTAssertEqual(receivedC, [11, 13, 16, 20].asEvents(completion: .finished))
  XCTAssertEqual(receivedD, [13, 16, 20].asEvents(completion: .finished))
}
```

可以看到，这次 sinkD 接收到的事件中，就包含了订阅 multicastB 时，它当前的最新事件 13。这个值是缓存在 multicastB 里的，并不需要 scan 重新计算。

What's next?

但 CurrentValueSubject 只能缓存一个事件，如果想要缓存多个怎么办呢？你第一个想到的可能是 Publishers.Buffer，但它只能在“热”Publisher 以及它的订阅者之间缓存事件，并不能在它的多个下游订阅者之间缓存事件。也就是说，在我们这种情况里，Buffer 并不能为 sinkD 缓存特定数量的 sinkC 订阅到事件。因此，如果真的需要这样的缓存行为，我们就只能自己实现一个了，好在有了之前实现 CustomSubject 的经验，这并不太困难。

实现自定义的事件缓冲区 - I

影片地址

实现自定义的事件缓冲区 - I[12]

定义 Buffer

接着上一节提出的问题，我们来实现一个叫做 `BufferSubject` 的类型，一方面，它发送事件的行为类似 `PassthroughSubject`，另一方面，它还支持在多个订阅者之间缓存事件。整体的实现套路，和之前实现 `CustomSubject` 基本上是一样的，分成四个部分：

- 首先，实现表达缓存行为的类 `Buffer`；
- 其次，为 `BufferSubject` 定义它自己的 `SubscriptionBehavior`；
- 第三，为 `BufferSubject` 实现 `Subject` 约束的方法；
- 第四，用单元测试验证预期的结果；

这一节，我们先来完成前两部分。

在项目目录中，新建一个 BufferSubject.swift。在这里，创建一个表达缓存行为的类型：

```
1  public struct Buffer<Output, Failure: Error> {
2   var values: [Output] = []
3   var completion: Subscribers.Completion<Failure>? = nil
4   let limit: Int
5   let strategy: Publishers.BufferingStrategy<Failure>
6
7   public var isEmpty: Bool {
8    return values.isEmpty && completion == nil
9   }
10 }
```

其中：

- `values` 表示缓存的普通事件数组；

[12] https://boxueio.com/series/build-boxue-app-in-mvvm/episode/624

- `completion` 表示缓存的完成事件。由于 `Buffer` 只应该缓存一个完成事件，因此，我们把 `values` 和 `completion` 分开定义，而没有统一定义成一个 `Event<Output, Failure>` 数组；
- `limit` 是缓存的上限，它应该是一个大于等于 0 的整数；
- `strategy` 是 Combine 提供的缓存策略，我们一会再说；
- `isEmpty` 用于判断缓存是否为空，它的依据除了 `value` 为空之外，还要判断 `completion` 是否为 `nil`；

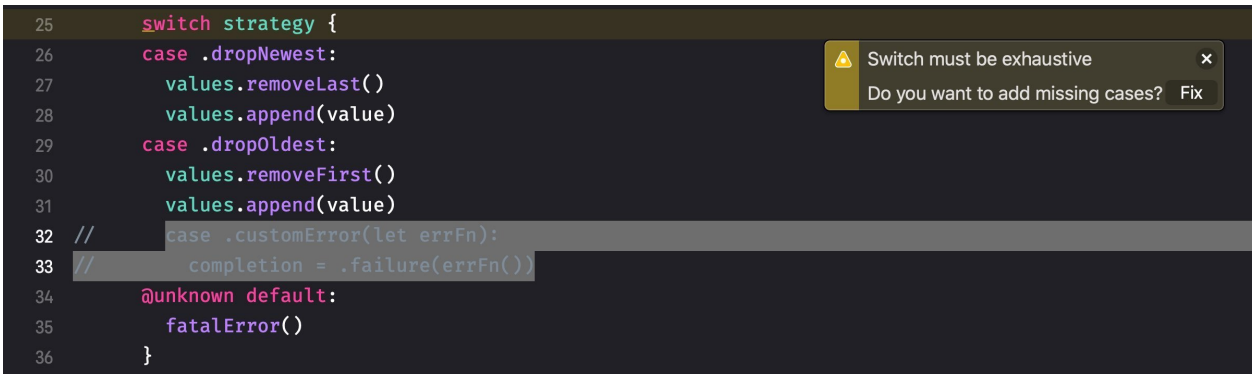
接下来，就是它的支持的接口了。我们先来看向缓存中添加事件的 `push`：

```
1  public mutating func push(_ value: Output) {
2   guard completion == nil else { return }
3   guard values.count < limit else {
4    switch strategy {
5    case .dropNewest:
6     values.removeLast()
7     values.append(value)
8    case .dropOldest:
9     values.removeFirst()
10    values.append(value)
11   case .customError(let errFn):
12    completion = .failure(errFn())
13   @unknown default:
14    fatalError()
15   }
16
17   return
18  }
19
20  values.append(value)
21 }
```

如果 `completion` 不为 `nil` 说明事件序列已经结束了，这时就不应该再缓存任何内容，直接返回。否则，判断下当前是否还可以缓存事件，如果可以，就直接把事件加入 `values` 末尾。否则，就说明缓存已经满了，这时，根据 `strategy` 支持的三种策略，我们要分别定义对应的行为：

- `.dropNewest` 表示放弃掉队列中最新的，因此，我们就用 `removeLast` 去掉 `values` 中最后一个，再把新事件放到数组末尾；
- `.dropOldest` 表示放弃掉队列中最旧的，因此，我们就用 `removeFirst` 去掉 `values` 中第一个，再把新事件放到数组末尾；
- `.customError` 表示用户提供了一个当发生这种情况时生成错误的方法，我们就用这个方法封装一个 `Completion<Failure>` 事件，并把它作为缓冲区的完成事件；

在 Swift 5 之后，由于 `BufferingStrategy<Failure>` 是一个 non-frozen enum，为了更好的处理未来新增的缓存策略，我们使用了 `@unknown default` 处理未知的情况。它和 `default` 唯一的区别就是，`@unknown default` 在我们没有使用 `case` 过完所有已知的情况之前，会显示一个警告信息：



但如果只用 `default` 处理默认情况的话，就没有这个“福利”了。完成后，我们来看向 `Buffer` 中添加完成事件的方法：

```
public mutating func push(
  completion: Subscribers.Completion<Failure>) {
  guard self.completion == nil else { return }
  self.completion = completion
}
```

它的实现就简单多了，只要 `Buffer` 中还没有完成事件，就设置 `self.completion`。最后，是从 `Buffer` 中取出事件的方法 `fetch`：

```
public mutating func fetch()
  -> Subscribers.Event<Output, Failure>? {
  if values.count > 0 {
    return .value(values.removeFirst())
  }
  else if let completion = self.completion {
    values = []
    self.completion = nil
    return .complete(completion)
  }

  return nil
}
```

如果缓冲区中还有事件，我们就把它包装在 `Event.value` 里返回。这里要注意的是，取出事件的顺序是从 `values` 的头元素开始的，因为向新的订阅者“回放”历史事件也要按照它们加入 `Buffer` 的顺序执行。

如果 values 空了，就检查 completion，如果不为 nil，就把 values 清空，completion 设置成 nil，并把之前缓存的完成事件封装成 Event.completion 返回。

最后，如果不是上述两种情况，Buffer 就没有任何内容可提供，我们就直接返回 nil。至此，这个提供事件缓存功能的 Buffer 就定义好了。

自定义 SubscriptionBehavior

接下来，我们基于 Buffer 实现 SubscriptionBehavior。继续在 BufferSubject.swift 里添加下面的代码：

```
public class BufferSubject<Output, Failure: Error>: Subject {
 class Behavior: SubscriptionBehavior {
  typealias Input = Output

  var upstream: Subscription? = nil
  let downstream: AnySubscriber<Input, Failure>
  let subject: BufferSubject<Output, Failure>
  var demand = Subscribers.Demand.none
  var buffer: Buffer<Output, Failure>

  init(subject: BufferSubject<Output, Failure>,
       downstream: AnySubscriber<Input, Failure>,
       buffer: Buffer<Output, Failure>){
   self.subject = subject
   self.downstream = downstream
   self.buffer = buffer
  }
 }

 /// ...
}
```

这里，BufferSubject 是下一节我们最终要实现的具备缓存功能的 Publisher。在 Behavior 的定义里：

- 首先，缓存功能并不对事件值进行类型转换，因此，Input 和 Ouput 的类型仍旧是相同的；
- 其次，在这个例子中，我们用不到 upstream，直接把它设置成 nil 就好；
- 第三，downstream 是下游订阅者的引用；
- 第四，subject 是这个 Behavior 服务的 Subject 对象，也就是我们将要定义的 BufferSubject；

- 第五，是可订阅的事件数量；
- 最后，`buffer` 是要执行的缓存策略；

了解了它的属性之后，我们先来看 `Behavior` 实现的接口。首先，是 `request(_ d: Subscribers.Demand)`:

```
func request(_ d: Subscribers.Demand) {
 demand += d

 while demand > 0, let next = buffer.fetch() {
  demand -= 1

  switch next {
  case .value(let value):
   let newDemand = downstream.receive(value)
   demand += newDemand
  case .complete(let completion):
   downstream.receive(completion: completion)
  }
 }
}
```

它的逻辑，就是只要缓冲区中还有事件，并且数量没超过订阅者发起的订阅请求数量，就不断向下游的订阅者发送缓存的事件。如果普通事件发送完了，就发送完成事件。通过它的实现，我们可以进一步理解 `demand` 的计算方法，它会根据每一次订阅者返回的 `receive` 方法的返回值进行调整。

其次，是 `Behavior` 自身作为订阅者，接收到普通事件的 `receive` 方法:

```
func receive(_ input: Output) -> Subscribers.Demand {
 if demand > 0 && buffer.isEmpty {
  let newDemand = downstream.receive(input)
  demand = newDemand + demand - 1
 }
 else {
  buffer.push(input)
 }

 return .unlimited
}
```

在它的实现里，如果，`demand` 大于 0，并且当前没有缓存任何事件。那我们也无需缓存收到的 `input` 事件，直接把它发送给下游订阅者，并更新 `demand` 就好了。否则，就表示我们

还有历史事件没有发送完，这时，就通过 `push` 把事件保存进 `buffer`。另外，从它返回的 `.unlimited` 我们就知道，`Behavior` 会一直向上游 `Publisher` 订阅事件。

第三，是 `Behavior` 自身作为订阅者，接收到完成事件的 `receive` 方法。它的实现则简单一些，如果缓存为空就直接向下游订阅者发送完成事件，否则就让这个事件进缓存：

```
func receive(completion: Subscribers.Completion<Failure>) {
 if buffer.isEmpty {
  downstream.receive(completion: completion)
 }
 else {
  buffer.push(completion: completion)
 }
}
```

最后，是执行取消订阅的 `cancel` 方法：

```
func cancel() {
 subject.subscribers.mutate {
  $0.removeValue(forKey: self.combineIdentifier)
 }
}
```

这和之前我们实现 `CustomSubject` 时实现的 `cancel` 是一样的，通过 `self.combineIdentifier`，从 `subscribers` 中删掉申请发起 `cancel` 请求的订阅者就好了。

What's next?

至此，这个为 `BufferSubject` 服务的 `Behavior` 就完成了。现在，稍微休息一会儿，下一节，基于这个实现，我们完成 `BufferSubject` 的定义，并编写单元测试来验证它的功能。

实现自定义的事件缓冲区 - I

影片地址

实现自定义的事件缓冲区 - I[13]

BufferSubject

欢迎回来，基于上一节实现的 `Behavior`，我们完成 `BufferSubject` 剩余的部分，也就是 `Subject` 接口的实现。我们从 `BufferSubject` 的属性部分说起：

```
public class BufferSubject<Output, Failure: Error>: Subject {
  /// class Behavior ...
  typealias SubscriberRecords =
    Dictionary<CombineIdentifier, CustomSubscription<Behavior>>
  let subscribers = AtomicBox<SubscriberRecords>([:])
  let buffer: AtomicBox<Buffer<Output, Failure>>

  public init(limit: Int = 1,
    whenFull strategy: Publishers.BufferingStrategy<Failure> = .dropOldest) {
    precondition(limit >= 0)
    buffer = AtomicBox(Buffer(limit: limit, strategy: strategy))
  }
}
```

其中，`subscribers` 的定义，和 `CustomSubject` 是一样的。通过它的定义，我们可以进一步体会在 `CustomSubscription` 的实现里，把 `Subscription` 行为和线程安全性的代码隔离开的好处。另外一个属性 `buffer` 则是 `BufferSubject` 使用的缓冲区，对它的访问也需要是线程安全的。

接下来，是我们还不太清楚的 `send(subscription:)` 方法。老规矩，我们只是保持让 `subscription` 订阅事件：

[13] https://boxueio.com/series/build-boxue-app-in-mvvm/episode/625

```
public func send(subscription: Subscription) {
  subscription.request(.unlimited)
}
```

然后，是向订阅者发送普通事件的 `send(_ value:)` 方法。在它的实现里，我们做了两件事情。一个是把事件缓存，以便将来可以回放给后来的订阅者；另一个，是把事件发送给当前所有的订阅者：

```
public func send(_ value: Output) {
  buffer.mutate { b in
    b.push(value)
  }

  for (_, sub) in subscribers.value {
    _ = sub.receive(value)
  }
}
```

接下来，是向订阅者发送完成事件的 `send(completion:)` 方法。它的实现逻辑，和 `send(_-value:)` 几乎是一样的，只是发送完事件后，要清空 `subscribers`：

```
public func send(completion: Subscribers.Completion<Failure>) {
  buffer.mutate { b in
    b.push(completion: completion)
  }

  for (_, sub) in subscribers.value {
    sub.receive(completion: completion)
  }

  subscribers.mutate { $0.removeAll() }
}
```

这样，`BufferSubject` 中 `Subject` 部分的接口就完成了。最后，我们来实现 `Publisher` 的部分：

```
public func receive<S>(subscriber: S)
  where S: Subscriber, Failure == S.Failure, Output == S.Input {
  let behavior = Behavior(
    subject: self,
    downstream: AnySubscriber(subscriber),
    buffer: buffer.value)

  let subscription = CustomSubscription(behavior: behavior)
  subscribers.mutate {
    $0[subscription.combineIdentifier] = subscription
  }

  subscription.receive(subscription: Subscriptions.empty)
}
```

通过它的实现，我们就能理解当订阅 `BufferSubject` 的时候，究竟发生了什么。首先，创建 `Behavior` 的时候，可以看到所有订阅者共享的都是同一个 `buffer`，这也就意味着，在缓冲区未满的情况下，先来的订阅者和后来的订阅者，应该可以收到同样的消息，这也就达到了之前我们期望的给新订阅者回放所有历史事件的目的。至于 `receive<S>(subscriber: S)` 后半部分的实现，则和 `CustomSubject` 是完全一样的，我们就不重复了。

至此，`BufferSubject` 也就全部完成了。

用单元测试验证结果

接下来，我们把测试 `multicast` 的测试用例改成下面这样，来测试下 `BufferSubject`：

```
func testMulticastBuffer() {
  let subjectA = PassthroughSubject<Int, Never>()
  let multicastB = subjectA.scan(10, +)
    .multicast { BufferSubject(limit: Int.max) }
    .autoconnect()

  /// `sinkC`
  /// `sinkD`

  XCTAssertEqual(receivedC, [11, 13, 16, 20].asEvents(completion: .finished))
  XCTAssertEqual(receivedD, [11, 13, 16, 20].asEvents(completion: .finished))

  sinkC.cancel()
  sinkD.cancel()
}
```

我们只是把 multicast closure 中返回的 PassthroughSubject 替换成了 BufferSubject。订阅的过程以及事件发送的顺序，则和之前是完全一样的。现在，当 sinkD 订阅 multicastB 的时候，它就可以订阅到之前 sinkC 订阅到的所有事件了。从最后 XCTAssertEqual 判断的条件也可以看到，receivedC 和 receivedD 是相等的。

What's next?

说到这里，如果你回头看看之前所有使用了 multicast 方法的测试用例就不难发现，为了让 multicastB 生成事件，我们都使用了 autoconnect() 方法。如果换成手动连接，也可以写成这样：

```
func testMulticastLatest() {
  let subjectA = PassthroughSubject<Int, Never>()
  let multicastB = subjectA.scan(10, +)
    .multicast { CurrentValueSubject(0) }
  let cancelB = multicastB.connect()

  /// The same as before...

  cancelB.cancel()
}
```

当然，这里是否调用 cancel 并不影响我们的单元测试结果。但如果忽略 connect() 的返回值：

```
_ = multicastB.connect()
```

sinkC 和 sinkD 就无法订阅到任何事件了。这也和我们预期中的行为有所差别。凭着直觉，connect 之后，multicastB 就应该开始产生了事件了。究竟是什么导致了这个现象呢？为了搞清楚这个问题，下一节，我们来聊聊和 Combine 中 Subject 生命周期相关的话题。

有些意外的 Subject 生命周期

影片地址

有些意外的 Subject 生命周期[14]

从一个最自然的模型开始

上一节末尾，我们提出了一个问题，为什么必须要保持 `multicastB.connect()` 返回值的引用才可以让 `sinkC` 和 `sinkD` 订阅到事件呢？这似乎又是个有悖于我们直觉的行为。在 Combine 里，我们必须要小心翼翼地保持 `Publisher` 的引用才可以让订阅者正常订阅到事件么？

为了研究这个问题，我们先创建一个新的测试用例：

```
1  func testAnyCancellable() {
2   let subject = PassthroughSubject<Int, Never>()
3   var received = [Subscribers.Event<Int, Never>]()
4
5   weak var weakCancellable: AnyCancellable?
6
7   do {
8    let anyCancellable = subject.sink(event: { received.append($0) })
9    weakCancellable = anyCancellable
10
11   subject.send(1)
12  }
13
14  XCTAssertNil(weakCancellable)
15
16  subject.send(2)
17  XCTAssertEqual(received, [1].asEvents(completion: nil))
18 }
```

这个测试用例的重点，就是中间我们用 `do` 手工创建的作用域。之所以要这样做，是为了尽可能避免不同编译设置下，编译器对代码作用域进行的优化。在这个作用域里，我们自

[14] https://boxueio.com/series/build-boxue-app-in-mvvm/episode/626

已实现的 `sink(event:)` 返回一个 `AnyCancellable` 对象。在 Apple 官方文档[15]里明确说明了，`AnyCancellable` 对象在离开作用域的时候，会自动调用 `cancel()` 方法取消订阅。

于是，`received` 中，应该有在 `do` 作用域内订阅到的事件 `1`。离开作用域后，`anyCancellable` 就被释放了，它会自动取消订阅，进而，`weakCancellable` 就会变成 `nil`，这也是我们要测试的第一个条件。之后，我们又通过 `subject` 生成了事件 `2`，此时 `anyCancellable` 已经不存在了，`received` 中自然也不会有新事件进账，这是我们要测试的第二个条件。

直接执行这个测试，结果应该和我们描述的一模一样，这也是最符合我们预期的行为。

不会自动取消的订阅者

但如果我们换一种订阅方式，直接使用 `Subscribers.Sink`，结果就有些迷惑了。把刚才那个测试用例改成这样：

```
1  func testSinkCancellation() {
2   let subject = PassthroughSubject<Int, Never>()
3   var received = [Subscribers.Event<Int, Never>]()
4
5   weak var weakSink: Subscribers.Sink<Int, Never>?
6
7   do {
8    let sink = Subscribers.Sink<Int, Never>(
9     receiveCompletion: { received.append(.complete($0))},
10    receiveValue: { received.append(.value($0))})
11
12   weakSink = sink
13   subject.subscribe(sink)
14
15   subject.send(1)
16  }
17
18  XCTAssertNotNil(weakSink)
19
20  subject.send(2)
21  weakSink?.cancel()
22  subject.send(3)
23
24  XCTAssertEqual(received, [1, 2].asEvents(completion: nil))
25 }
```

[15] https://developer.apple.com/documentation/combine/anycancellable

在 do 作用域里，我们只是把 anyCancellable 换成了 sink。从测试的条件可以看到，这次，weakSink 不再是 nil 了，这就意味着，尽管离开了作用域，sink 引用的对象仍旧是存活着的！直到我们手工调用了 weakSink?.cancel()，它才会释放自己，进而不会订阅到接下来的事件 3。按照我们之前的分析，之所以 sink 能存活下来，很有可能是 subject 创建的 Subscription 对象持有了 sink 所代表的订阅者的引用。

如果你第一次看到这个场景，想必会对此感到困惑。

一个不需要任何强引用的场景

但是，故事到此还没结束。实际上，不仅我们不需要保持订阅者的引用，就连 subject 的强引用也可以没有，整个订阅还是有机会正常工作。来看下面这个测试用例：

```
func testOwnership() {
  var received = [Subscribers.Event<Int, Never>]()

  weak var weakSubject: PassthroughSubject<Int, Never>?
  weak var weakSink: Subscribers.Sink<Int, Never>?

  do {
    let subject = PassthroughSubject<Int, Never>()
    weakSubject = subject

    let sink = Subscribers.Sink<Int, Never>(
      receiveCompletion: { received.append(.complete($0))},
      receiveValue: { received.append(.value($0))})
    weakSink = sink

    subject.subscribe(sink)
  }

  XCTAssertNotNil(weakSubject)
  XCTAssertNotNil(weakSink)

  weakSubject?.send(1)
  weakSubject?.send(completion: .finished)

  XCTAssertNil(weakSubject)
  XCTAssertNil(weakSink)

  XCTAssertEqual(received, [1].asEvents(completion: .finished))
}
```

这次，在 do 外部，我们只创建了两个弱引用，而创建真正的 Sink 和 PassthroughSubject 对象，以及订阅，都是在 do 作用域内部发生的。按照我们的想象，离开作用域后，subject 应该被销毁，sink 的订阅也应该被取消。

但事实并不如此，从：

```
1 XCTAssertNotNil(weakSubject)
2 XCTAssertNotNil(weakSink)
```

这两个判断就可以看到，subject 和 sink 引用的对象依旧存活，并且，subject 也可以订阅到在 do 作用域之外发生的事件 1 和 .finished。直到 .finished 之后，subject 和 sink 才会自然销毁。

这可是个我们应该绝对提高警惕的场景，甚至绝大多数情况下，这都不是我们预期的行为。至于为什么如此，以及未来这种行为会不会改变，由于 Combine 并不开源，我们也暂时无法得知了。通过这些试验，我们唯一能确定的就是：始终都应该在你的订阅模型中，保持一个 **AnyCancellable** 角色，它可以像我们预期的一样，管理 **Combine** 中订阅者和发布者的生存周期。

What's next?

以上，就是我们对 Combine 中内存管理方式的一些探索。回顾一下之前的各种研究，我们都是围绕着一个 Publisher 存在着多个订阅者的情况。下一节，我们来研究反过来的情况。可以让一个订阅者同时订阅多个 Publisher 么？此时的订阅者会收到来自这些 Publisher 的事件么？

Combine 中的多重订阅 - I

影片地址

Combine 中的多重订阅 - I[16]

什么是多重订阅呢

如果让一个订阅者同时订阅多个 Publisher，它可以同时收到来自这些 Publisher 的事件么？这一节，我们就来研究这个问题。

首先，我们直接创建一个表达这个想法的测试用例：

```
1  func testMultipleSubscribe() {
2    let subject1 = PassthroughSubject<Int, Never>()
3    let subject2 = PassthroughSubject<Int, Never>()
4    var received = [Subscribers.Event<Int, Never>]()
5
6    let sink = Subscribers.Sink<Int, Never>(receiveCompletion: {
7      received.append(.complete($0))
8    }, receiveValue: {
9      received.append(.value($0))
10   })
11
12   subject1.subscribe(sink)
13   subject2.subscribe(sink)
14
15   subject1.send(sequence: 1...2, completion: .finished)
16   subject2.send(sequence: 3...4, completion: .finished)
17
18   XCTAssertEqual(received, (1...2).asEvents(completion: .finished))
19 }
```

最后的判断条件，就完全说明问题了。首先，一个订阅者可以订阅多个 Publisher，上面的代码可以运行；其次，一个订阅者只能订阅到第一个 Publisher 生成的事件，而忽略其余的

[16] https://boxueio.com/series/build-boxue-app-in-mvvm/episode/627

Publisher。按照之前我们对 Combine 事件发布订阅模型的整理，一个 Subscriber 只能持有一个来自 Publisher 的 Subscription 对象，并从中接收事件。

但如果我们就是要同时订阅多个 Publisher 该怎么办呢？在 Combine 里，目前唯一具有这个功能的，就是 Subject。为了演示这个特性，我们把刚才的测试用例改成这样：

```
1  func testMultipleSubjectSubscribe() {
2   let subject1 = PassthroughSubject<Int, Never>()
3   let subject2 = PassthroughSubject<Int, Never>()
4   let multiSubject = PassthroughSubject<Int, Never>()
5
6   let cancellable1 = subject1.subscribe(multiSubject)
7   let cancellable2 = subject2.subscribe(multiSubject)
8
9   var received = [Subscribers.Event<Int, Never>]()
10
11  let sink = multiSubject.sink(receiveCompletion: {
12    received.append(.complete($0))
13  }, receiveValue: {
14    received.append(.value($0))
15  })
16
17  subject1.send(sequence: 1...2, completion: nil)
18  subject2.send(sequence: 3...4, completion: .finished)
19
20  XCTAssertEqual(received, [1, 2, 3, 4].asEvents(completion: .finished))
21
22  cancellable1.cancel()
23  cancellable2.cancel()
24  sink.cancel()
25 }
```

这次，我们先把 multiSubject 当成了一个订阅者，订阅了上游的 subject1 和 subject2。然后，再把它当成了一个 Publisher，并订阅了其中的事件。从测试条件就能看到，这样，就可以订阅到多个上游 Publisher 产生的事件了。

但这个结果也是有条件的，首先，我们必须小心构建 subject1 和 subject2，让它们产生的事件逻辑上，可以形成一个流。因此，我们让 subject1 的完成事件是 nil。好让 multiSubject 可以继续从 subject2 接受事件。如果我们让 subject1 的完成事件也是 .finsihed，multiSubject 就订阅不到 subject2 的事件了。

另外，我们需要手工维护所有用于取消上游订阅的 Cancellable 对象。因为，即便 subject1 和 subject2 离开了作用域，这个订阅也不会默认被取消掉。所以，尽管上面的代码“可以工作”，但附加的条条框框足以让我们对这样的代码敬而远之。

MergeSink

为了解决这些不便之处，我们可以给 PassthroughSubject 写一个包装类改进上面提出的问题。一方面，保留它订阅多个上游 Publisher 的能力；另一方面，让这个包装类自我管理处于中间环节的 Cancellable。为此，我们创建了一个 MergeSink.swift，在其中定义了一个叫做 MergeInput 的类用来合并多个 Publisher 的事件，先来看 MergeSink 的属性：

```
public class MergeInput<I>: Publisher, Cancellable {
  public typealias Output = I
  public typealias Failure = Never

  private let subscriptions = AtomicBox(
    Dictionary<CombineIdentifier, Subscribers.Sink<I, Never>>())
  private let subject = PassthroughSubject<I, Never>()

  public init() {}
}
```

其中：

- subscriptions 用来管理所有订阅上游 Publisher 的订阅者；
- subject 是为下游订阅者服务的 Publisher；

然后，来看它的 Publisher 接口部分的实现。作为一个普通的 Publisher，它的实现也就是把订阅者传递给实际提供订阅功能的 subject 的 receive(subscriber:) 方法就好了。

```
public func receive<S>(subscriber: S)
  where S: Subscriber, S.Failure == Never, S.Input == I {
  subject.receive(subscriber: subscriber)
}
```

接下来，是 Cancellable 接口部分的实现。这里我们需要做的，是调用每一个订阅者的 cancel() 方法取消订阅，然后清空 subscribers 字典：

```
public func cancel() {
  subscriptions.mutate {
    $0.values.forEach { $0.cancel() }
    $0.removeAll()
  }
}
```

为了让 MergeInput 可以在释放的时候，自动取消所有订阅，我们只要在 deinit 中调用 cancel 就好了：

```
deinit { cancel() }
```

最后，是让 `MergeInput` 接受多个 `Publisher` 事件的实现：

```
public func subscribe<P>(_ publisher: P)
 where P: Publisher, P.Output == I, P.Failure == Never {
 var identifier: CombineIdentifier?

 let sink = Subscribers.Sink<I, P.Failure>(
  receiveCompletion: { _ in
   self.subscriptions.mutate {
    _ = $0.removeValue(forKey: identifier!)
   }
  }, receiveValue: {
   self.subject.send($0)
  })

 identifier = sink.combineIdentifier
 subscriptions.mutate { $0[sink.combineIdentifier] = sink }

 publisher.subscribe(sink)
}
```

它的参数 `publisher` 表示要订阅的上游 `Publisher`，这个 `Publisher` 的普通事件类型和 `MergeInput` 相同，错误事件类型也是 `Never`。至于实现的方法，就是在内部创建一个 `publisher` 的订阅者，当收到普通事件的时候，就让它通过 `subject` 转发给下游订阅者。收到完成事件的时候，就把它从 `subscriptions` 中删除。创建完成后，把它纳入到 `subscriptions` 统一管理，并用它订阅 `publisher` 就好了。

有了这个 `MergeInput` 之后，我们给 `Publisher` 添加一个扩展，方便让它接受更多 `Publisher`：

```
public extension Publisher where Failure == Never {
 func merge(into mergeInput: MergeInput<Output>) {
  mergeInput.subscribe(self)
 }
}
```

What's next?

这样，我们就可以用这个扩展，先让 `subject` 合并进多个 `Publisher`，再让这个 `subject` 为下游订阅者提供服务了。下一节，我们将为 `MergeInput` 编写测试用例，验证这个应用场景。并继续讨论另外一种多次订阅的情况。

Combine 中的多重订阅 - II

影片地址

Combine 中的多重订阅 - II[17]

从测试用例开始

欢迎回来，这一节开始，我们先编写一个新的测试用例，验证 `MergeInput` 合并订阅多个 `Publishers` 的能力。在 CustomCombineTests.swift 里，添加下面的代码：

```
1  func testMergeInput() {
2    let subject1 = PassthroughSubject<Int, Never>()
3    let subject2 = PassthroughSubject<Int, Never>()
4    let input = MergeInput<Int>()
5
6    subject1.merge(into: input)
7    subject2.merge(into: input)
8
9    var received = [Subscribers.Event<Int, Never>]()
10
11   let sink = input.sink(receiveCompletion: {
12     received.append(.complete($0))
13   }, receiveValue: {
14     received.append(.value($0))
15   })
16
17   subject1.send(sequence: 1...2, completion: .finished)
18   subject2.send(sequence: 3...4, completion: .finished)
19
20   XCTAssertEqual(received, [1, 2, 3, 4].asEvents(completion: nil))
21
22   sink.cancel()
23 }
```

[17] https://boxueio.com/series/build-boxue-app-in-mvvm/episode/628

大体上，它和 testMultipleSubjectSubscribe 是一样的，只不过我们把订阅上游 Publisher 的 PassthroughSubject 换成了 MergeInput。这次，我们就无需再手工管理用于订阅的 Cancellable 对象了，也不用再刻意构建 subject1 和 subject2 的事件内容了。当 subject1 和 subject2 结束之后，received 收录的就是我们期望的 [1, 2, 3, 4] 事件。

无法“续命”的订阅

了解了同时订阅多个 Publisher 的行为之后，我们再来探讨另外一种情况：当一个订阅者订阅的 Publisher 结束之后，还可以用它继续订阅其它 Publisher 么？为了验证这个过程，我们先来实现这个测试用例的前半部分：

```
1  func testSinkReactivation() {
2   var received = [Subscribers.Event<Int, Never>]()
3   let sink = Subscribers.Sink<Int, Never>(receiveCompletion: {
4    received.append(.complete($0))
5   }, receiveValue: {
6    received.append(.value($0))
7   })
8
9   weak var weakSubject: PassthroughSubject<Int, Never>?
10
11  do {
12   let subject = PassthroughSubject<Int, Never>()
13   weakSubject = subject
14   subject.subscribe(sink)
15
16   subject.send(1)
17  }
18
19  XCTAssertNotNil(weakSubject)
20 }
```

其中，sink 是我们用于测试的订阅者，它订阅了 do 作用域中的 subject。按照之前说过的内容，离开作用域之后，subject 并不会被释放，订阅也依然有效，此时的 weakSubject 并不是 nil。接下来，通过 weakSubject，我们让事件序列结束：

```
func testSinkReactivation() {
 /// The same as before...

 weakSubject?.send(completion: .finished)
 XCTAssertNil(weakSubject)
 XCTAssertEqual(received, [1].asEvents(completion: .finished))
}
```

这次，`weakSubject` 就是 `nil` 了，并且 `received` 中应该可以收录到订阅的事件 `1` 以及 `.finished`。

接下来，按照常理，后面就没有 `subject` 和 `sink` 什么事儿了。但这时，如果我们用 `sink` 重新订阅一个新的 `Publisher` 会如何呢？继续在 `testSinkReactivation` 里添加下面的代码试一下：

```
func testSinkReactivation() {
 /// The same as before...

 let subject2 = PassthroughSubject<Int, Never>()
 subject2.subscribe(sink)
 subject2.send(2)

 XCTAssertEqual(received, [1].asEvents(completion: .finished))
}
```

就像测试中判断依据所展示的，`sink` 已经无法再订阅到 `subject2` 中的事件。也就是说，一旦订阅结束了，就是结束了。不过要说明一下的是，这个行为仅在 **mac OS 10.15 beta6** 及以后的版本才生效。在 beta6 之前，`sink6` 是可以订阅到 `subject2` 的事件的。

What's next?

以上，就是所有关于 Combine 中有关重新订阅这个行为的研究。在众多发现里，最重要的一个，就是订阅代码有可能会把 `Publisher` 和 `Subscriber` 保持在内存里，进而造成意料之外的内存泄漏。使用 `sink` 方法订阅相对安全，因为它返回的 `AnyCancellable` 对象会自动取消订阅。而如果要使用自定义的 `Subscribers.Sink` 对象，就要格外小心，手工处理好 `cancel` 的调用。或者，确保事件序列的正常结束。

说到这，我们对 Combine 中的各种边缘情况，整理的就差不多了。接下来，就是最接近在泊学 App 开发中遇到的一个问题了。你认为一旦 `Subscriber` 订阅了 `Publisher` 之后，就可以收到它的所有事件了么？或者说，一旦 `Subscriber` 取消了订阅，它就再也不会订阅到到事件了么？

让人意外的是，在 Combine 里，这两个问题的答案都是 No。在接下来的内容里，我们就来研究下这些场景，导致问题发生的原因，以及避免的方法。

Combine 事件供给机制的回顾

影片地址

Combine 事件供给机制的回顾[18]

CustomDemandSink

在上一节末尾，我们提出了一些 Combine 中看似“有违常理”的现象。为了搞清楚这些问题，我们要从 `Publisher` 和 `Subscriber` 之间事件的传递方式说起。

之前，实现 `SubscriptionBehavior` 的时候，接收普通事件的 `receive(_ input:)` 方法是这样的：

```
1  func receive(_ input: Input) -> Subscribers.Demand {
2   if demand > 0 {
3    let newDemand = downstream.receive(input)
4    demand = newDemand + (demand - 1)
5
6    return demand
7   }
8
9   return Subscribers.Demand.none
10 }
```

也就是说，只有在 `demand` 大于 0 的时候，`Publisher` 才会把 `input` 递交给订阅者，否则就会丢弃 `input`。通常，在自定义 `Subscriber` 的时候，`receive(_ input:)` 最后都会直接返回 `.unlimited` 表示一直订阅到 `Publisher` 结束。特别是那些订阅用户交互事件的代码，更是如此。因此，依靠 `demand` 决定是否递交事件的模式在绝大多数情况下，并不会产生问题。

CustomDemandSink

接下来，为了更具象地观察到 `demand` 对订阅事件的影响。在项目中，我们创建了一个 CustomDemandSink.swift。在这里定义一个可以动态调整 `demand` 的订阅者类型：`CustomDemandSink`。先来看它的属性：

[18] https://boxueio.com/series/build-boxue-app-in-mvvm/episode/629

```
public struct CustomDemandSink<Input, Failure: Error>
 : Subscriber, Cancellable {
 public let combineIdentifier: CombineIdentifier = CombineIdentifier()
 let activeSubscription =
  AtomicBox<(demand: Int, subscription: Subscription?)>((0, nil))
 let value: (Input) -> Void
 let completion: (Subscribers.Completion<Failure>) -> Void

 public init(
  demand: Int,
  receiveValue: @escaping ((Input) -> Void),
  receiveCompletion: @escaping ((Subscribers.Completion<Failure>) -> Void)) {
  activeSubscription.mutate { $0.demand = demand }
  self.value = receiveValue
  self.completion = receiveCompletion
 }
}
```

其中：

- `combineIdentifier` 是来自 `Subscriber` 的要求。由于 `CustomDemandSink` 是一个结构，Swift 无法自动为它合成这个属性，因此我们直接用 Combine API 定义了一个；
- `activeSubscription` 是一个用 `AtomicBox` 保护的 tuple。`demand` 部分表示允许我们动态进行调整的订阅需求，`subscription` 表示当前订阅的 `Publisher`；
- `value` 和 `completion` 是订阅到普通和完成事件后，调用的方法；

接下来，我们定义一个调整订阅数量的方法 `increaseDemand(_ value:)`，它把 `activeSubscription` 内部的 `subscription` 订阅事件的最大值，调整到 `value`。之所以要有这个方法，是为了稍后方便我们观察 `activeSubscription` 的订阅情况：

```
public func increaseDemand(_ value: Int) {
 activeSubscription.mutate {
  $0.subscription?.request(.max(value))
 }
}
```

然后，是 `Cancellable` 协议的实现。我们调用 `subscription` 的 `cancel` 方法取消订阅，并重置 `activeSubscription` 的状态就好了：

```
public func cancel() {
  activeSubscription.mutate {
    $0.subscription?.cancel()
    $0 = (0, nil)
  }
}
```

最后，是 `Subscriber` 的实现，也就是它约束的三个 `receive` 方法：

```
public func receive(subscription: Subscription) {
  activeSubscription.mutate {
    if $0.subscription == nil {
      $0.subscription = subscription

      if $0.demand > 0 {
        $0.demand -= 1
        subscription.request(.max(1))
      }
    }
  }
}

public func receive(_ input: Input) -> Subscribers.Demand {
  value(input)

  var demand = Subscribers.Demand.none
  activeSubscription.mutate {
    if $0.demand > 0 {
      $0.demand -= 1
      demand = .max(1)
    }
  }

  return demand
}

public func receive(completion c: Subscribers.Completion<Failure>) {
  completion(c)
  activeSubscription.mutate {
    $0 = (0, nil)
  }
}
```

先来看 `receive(subscription:)`。如果 `activeSubscription.subscription` 为 `nil`，就把接收到的参数设置成当前的 `Subscription` 对象，并向这个对象表示的 `Publisher` 订阅 1 个事件。

再来看 `receive(_ input:)`。它先调用我们指定的 `value` 处理事件 `input`。后半部分，则和 `receive(subscription:)` 是基本相同的。只是当 `activeSubscription.demand` 为 0 的时候，它会向上游 `Publisher` 返回 `.none` 表示不再接收事件。

最后，是 `receive(completion c:)`。它先调用我们指定的 `completion` 方法处理完成事件。然后重置 `activeSubscription` 的值。至此，这个可以在外部手工调整订阅需求的订阅者就完成了。

编写测试用例

接下来，我们编写一个测试用例，观察下 `demand` 对订阅事件的影响：

```
1  func testDemand() {
2   var received = [Subscribers.Event<Int, Never>]()
3   let subject = PassthroughSubject<Int, Never>()
4   let sink = CustomDemandSink<Int, Never>(
5    demand: 2,
6    receiveValue: { received.append(.value($0)) },
7    receiveCompletion: { received.append(.complete($0)) }
8   )
9
10  subject.subscribe(sink)
11
12  subject.send(sequence: 1...3, completion: nil)
13  sink.increaseDemand(2)
14  subject.send(sequence: 4...6, completion: .finished)
15  sink.increaseDemand(2)
16
17  XCTAssertEqual(received, [1, 2, 4, 5].asEvents(completion: .finished))
18 }
```

首先，我们创建了一个可订阅 2 个事件的 `CustomDemandSink` 对象。当 `subject` 产生 `1...3` 事件的时候，显然 `sink` 只应该订阅到 1 和 2，3 会被 `subject` 丢弃掉。

之后，我们把 `sink` 的 `demand` 又加了 2。于是，`sink` 可以订阅到接下来的 4 和 5，6 会被 `subject` 丢弃掉。因此，在最后的测试条件里，我们判断的事件，就是 `[1, 2, 4, 5]`。

通过这个例子，我们至少已经呈现了上一节提出的一个现象：订阅了一个 `subject`，并不一定可以收到它接下来的所有事件。

What’s next?

看过了这个例子，你可能会觉得，尽管 demand 是一种控制订阅事件数量的机制，但理解它的前前后后要付出的努力，可能真的要比它实际的作用大的多得多。特别是，当事件的发送，和订阅不在同一个线程的时候，这个功能简直就是个 Bug 一样的存在了。下一节，我们就来看下这种情况以及对应的应对方法。

Combine 中的异步事件调度

影片地址

Combine 中的异步事件调度[19]

一个形式上异步的测试用例

接着上一节的话题，为了让事件的生成和订阅发生在不同的线程里，我们先在测试用例中使用 `receive(on:)`，把订阅代码“调度”走：

```
func testReceiveOnImmediate() {
  let e = expectation(description: "")
  let subject = PassthroughSubject<Int, Never>()
  var received = [Subscribers.Event<Int, Never>]()

  let sink = subject.receive(on: ImmediateScheduler.shared)
    .sink(receiveCompletion: {
      received.append(.complete($0))
      e.fulfill()
    }, receiveValue: {
      received.append(.value($0))
    })

  subject.send(1)
  subject.send(completion: .finished)

  wait(for: [e], timeout: 5.0)

  XCTAssertEqual(received, [1].asEvents(completion: .finished))

  sink.cancel()
}
```

现在，事件的发送在测试线程里（也就是主线程），事件的接收在 `ImmediateScheduler` 指定的线程里（在这个例子里，其实仍旧是主线程，我们只是从一个最简单的调度开始，稍后

[19] https://boxueio.com/series/build-boxue-app-in-mvvm/episode/630

就会看到真正调度到其它线程的情况）。为了观察事件的订阅，我们使用了 `expectation`，并在订阅到完成事件时，把它设置成 `fulfill` 状态。最后，使用 `wait` 等待 `expectation` 被满足。这样一来，`received` 中应该可以收集到来自主线程的事件 1 和 `.finsihed`。

当然，这个测试用例更多地是在从想法上表达如何测试事件发送和订阅在不同线程中的情况。实际上，`testReceiveOnImmediate` 中没有任何异步执行的环节，整个过程的执行，仍旧是同步的。

一个会丢事件的订阅场景

接下来，我们把订阅代码真正调度到一个后台线程，来继续观察 `received` 的值：

```
1  func testReceiveOnFailure() {
2   let queue = DispatchQueue(label: "test")
3   let e = expectation(description: "")
4   let subject = PassthroughSubject<Int, Never>()
5   var received = [Subscribers.Event<Int, Never>]()
6   print(Thread.current)
7   let sink = subject.receive(on: queue)
8    .sink(receiveCompletion: {
9     print(Thread.current)
10    received.append(.complete($0))
11    e.fulfill()
12   }, receiveValue: {
13    received.append(.value($0))
14   })
15
16  subject.send(1)
17  subject.send(completion: .finished)
18
19  wait(for: [e], timeout: 5.0)
20  // The following assert will FAIL!
21  XCTAssertEqual(received, [1].asEvents(completion: .finished))
22
23  sink.cancel()
24 }
```

这里，唯一的变化，就是我们把创建订阅者的代码调度到了 `queue`。但让人诧异的是，上面这个测试用例不仅无法通过测试，如果你尝试多执行几次，就连 `received` 中的值都可能发生变化。它有时可以收集到 `.finished` 事件，有时又是一个空数组。

如果我们把发送 `.finished` 事件的代码也调度到 `queue`：

```
queue.async { subject.send(completion: .finished) }
```

`received` 中就可以稳定地收到 `.finsihed` 事件，但无论如何，它始终都无法接收到 `subject` 发送的事件 1。这就再一次印证了我们之前说过的一个现象：订阅了一个 `Publisher` 之后，不一定可以收到它的所有事件。事件有可能会被漏掉，在绝大多数情况下，这可绝对不是我们期望的结果。

为什么会如此呢?

一个便于观察 Publisher 的方法

根本的原因，就是当事件采用异步接收的模式时，`demand` 也会通过调度一个专门的 `request(_ demand:)` 方法来传递。而执行 `subject.send(1)` 的时候，传递 `demand` 的方法还没被调度执行，因此事件 1 就丢了。为了更好地观察这个现象，我们先创建一个辅助方法 `debug`:

```
public extension Publisher {
  func debug(
    prefix: String = "",
    function: String = #function,
    line: Int = #line) -> Publishers.HandleEvents<Self> {
    let pattern =
      "\(prefix + (prefix.isEmpty ? "" : " "))\(function), line \(line)"

    return handleEvents(receiveSubscription: {
      Swift.print("\(pattern)subscription \($0)")
    }, receiveOutput: {
      Swift.print("\(pattern)output \($0)")
    }, receiveCompletion: {
      Swift.print("\(pattern)completion \($0)")
    }, receiveCancel: {
      Swift.print("\(pattern)cancelled")
    }, receiveRequest: {
      Swift.print("\(pattern)request \($0)")
    })
  }
}
```

它的作用，就是在 `Publisher` 发生各种事件的时候，在控制台打印一段文字。其中：

- `prefix` 是打印信息的前缀，如果不为空，就在 `prefix` 后面追加一个空格便于识别；

- `function` 和 `line`，它们使用了 Swift 内置的宏 `#function` 和 `#line`，分别表示了调用 `debug` 的函数名称和所在行数；

最后，`debug` 返回了一个 `HandleEvents` 对象，它允许我们为 `Publisher` 可能发生的各种事件，注册一个 closure。这个过程是通过 `handleEvents` 方法完成的，它的 5 个参数，分别表示 `Publisher` 收到来自上游的 `Subscription` 对象、发生普通事件、发生完成事件、取消订阅以及收到更新 `demand` 请求，这 5 种不同的事件。当这些事件发生的时候，我们在控制台打印了对应的消息。

重新观察之前的丢消息场景

有了 `debug` 之后，我们改造下之前的测试用例，给 `subject` 挂上 `debug` 方法，重新观察下丢消息的场景：

```
1  func testReceiveWithDebug() {
2   let subject = PassthroughSubject<Int, Never>()
3   var received = [Subscribers.Event<Int, Never>]()
4
5   print("Start...")
6   let cancellable = subject
7    .debug()
8    .receive(on: DispatchQueue.main)
9    .sink(receiveValue: { received.append(.value($0)) })
10
11  print("Phase 1...")
12  subject.send(1)
13  XCTAssertEqual(received, [].asEvents(completion: nil))
14
15  print("Phase 2...")
16  RunLoop.current.run(until: Date(timeIntervalSinceNow: 0.001))
17  XCTAssertEqual(received, [].asEvents(completion: nil))
18
19  print("Phase 3...")
20  subject.send(2)
21  XCTAssertEqual(received, [].asEvents(completion: nil))
22
23  print("Phase 4...")
24  RunLoop.current.run(until: Date(timeIntervalSinceNow: 0.001))
25  XCTAssertEqual(received, [2].asEvents(completion: nil))
26
27  cancellable.cancel()
28 }
```

这个观察的过程，分成 4 个阶段：

- 首先，subject 生成事件 1 的时候，和之前一样，这个事件会丢掉，received 中仍旧是空的；
- 其次，我们启动了主线程中的 RunLoop，这时，被调度的订阅代码才会得以执行，就像之前说过的，一个专门的 request(_ demand:) 会被执行，向 subject 发送 .unlimited 订阅请求。但此时，received 仍旧是空的；
- 第三，subject 继续生成了事件 2，此时，receiveValue 指定的 closure 会被放入 RunLoop 等待调度；
- 第四，只要重新让 RunLoop 执行一下，就可以订阅到第三步的事件了；
- 最后，cancel 方法会取消 subject，debug 应该也可以捕获到这个事件；

有了这番分析之后，我们只要执行这个测试用例，就会在控制台看到类似这样的结果：

```
Phase 1...
Phase 2...
testReceiveWithDebug(), line 502: request unlimited
Phase 3...
testReceiveWithDebug(), line 502: output 2
Phase 4...
testReceiveWithDebug(), line 502: cancelled
```

对着刚才的分析，就会发现，它们是完全一样的。说到这，也就彻底解释通了为什么在这种场景下会丢失事件的原因了。在之前泊学 App 的代码里，也有一段类似的实现：

```
HomeDataAPI(remoteUserSession: userSessionRepository.remoteUserSession())
 .get()
 .receive(on: RunLoop.main)
 .sink(
  receiveCompletion: {
   /// Update UI...
  },
  receiveValue: {
   /// Update UI...
  })
 .store(in: &disposables)
```

之前，由于获取到首页数据之后，要执行一些更新 UI 的工作，因此我把订阅的代码通过 receive(on:) 调度到了主线程，结果发现无论如何也订阅不到接收到的数据。调试这个问题花费了很多天的时间。现在，有了这些对于 Combine 订阅模型的研究，就不难理解究竟是为什么了。

妥善使用 receive(on:) 的方法

通过这些例子，我们不难相信，在当前这个版本的 Combine 里，`receive(on:)` 绝对是一个危险的东西。如果你想调度特定的代码执行，更好的方法应该是在订阅的部分里，明确调度你的代码，而不是依赖 `receive(on:)`。如果你一定要如此，那么你几乎总是应该把它搭配上 `buffer` 操作符一起使用：

```
func testBufferedReceiveOn() {
  let subject = PassthroughSubject<Int, Never>()
  let e = expectation(description: "")
  var received = [Subscribers.Event<Int, Never>]()

  let cancellable = subject
    .buffer(size: Int.max, prefetch: .byRequest, whenFull: .dropNewest)
    .receive(on: DispatchQueue(label: "test"))
    .sink(receiveCompletion: {
      received.append(.complete($0))
      e.fulfill()
    }, receiveValue: {
      received.append(.value($0))
    })

  subject.send(1)
  subject.send(completion: .finished)

  wait(for: [e], timeout: 5.0)
  XCTAssertEqual(received, [1].asEvents(completion: .finished))

  cancellable.cancel()
}
```

这样一来，即便订阅的代码会晚于事件生成的代码，因为有了缓存，我们还是可以订阅到所有的历史事件。但为了彻底理解这个过程，我们整整用了 11 小节的内容。估计你也和我一样会觉得从实用的角度来说不那么划算。总之，还是尽可能避免这种异步订阅的代码，自己明确去安排那些要在不同线程执行的代码更好。至少对不同的开发者来说，它们更好理解和维护。

What's next?

以上，就是由于泊学 App 开发过程中的一个 bug，而引发的我们对 Combine 原理研究的全部内容了。下一节开始，我们将回到泊学 App 的开发，开始实现内容浏览的首页。

At Last

首先，感谢并恭喜你一路走来看到了这里（当然，你可能也只是很好奇这本册子的最后究竟有什么东西 .）。无论如何，希望它可以在你学习 Combine 的道路上提供一些帮助。并且，如果里面有一些理解不到位，甚至是错误的地方，也欢迎通过下面的方式联系我，我会及时反馈并更新。

- 新浪微博：https://www.weibo.com/boxueio
- 电子邮件：11@boxue.io