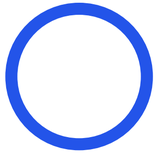
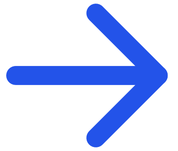


Let’s VisionOS 2024

新框架、新思维

解析 Observation 和 SwiftData 框架

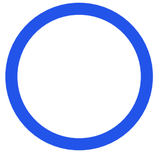


徐杨 (东坡肘子 / Fatbobman)

徐杨 东坡肘子

X: @fatbobman

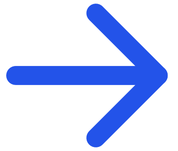
Blog: fatbobman.com



Let’s VisionOS 2024

新框架、新思维

解析 Observation 和 SwiftData 框架



徐杨 (东坡肘子 / Fatbobman)

Observation

Observation

减少由于旧有观察机制导致的过多无效的视图更新

提高 SwiftUI 应用的性能 

SwiftUI
当前的观察机制

SwiftUI
当前的观察机制

SwiftUI 独立实现的观察机制

对于视图内部的简单状态, 我们通常使用 @State 来声明，这种状态大多是值类型的。 SwiftUI 框架自身就实现了对值类型状态的观察机制，这种观察能力是独立实现的。

SwiftUI 当前的观察机制

SwiftUI 独立实现的观察机制

对于视图内部的简单状态, 我们通常使用 @State 来声明，这种状态大多是值类型的。 SwiftUI 框架自身就实现了对 @State 的观察机制，这种观察能力是独立实现的。

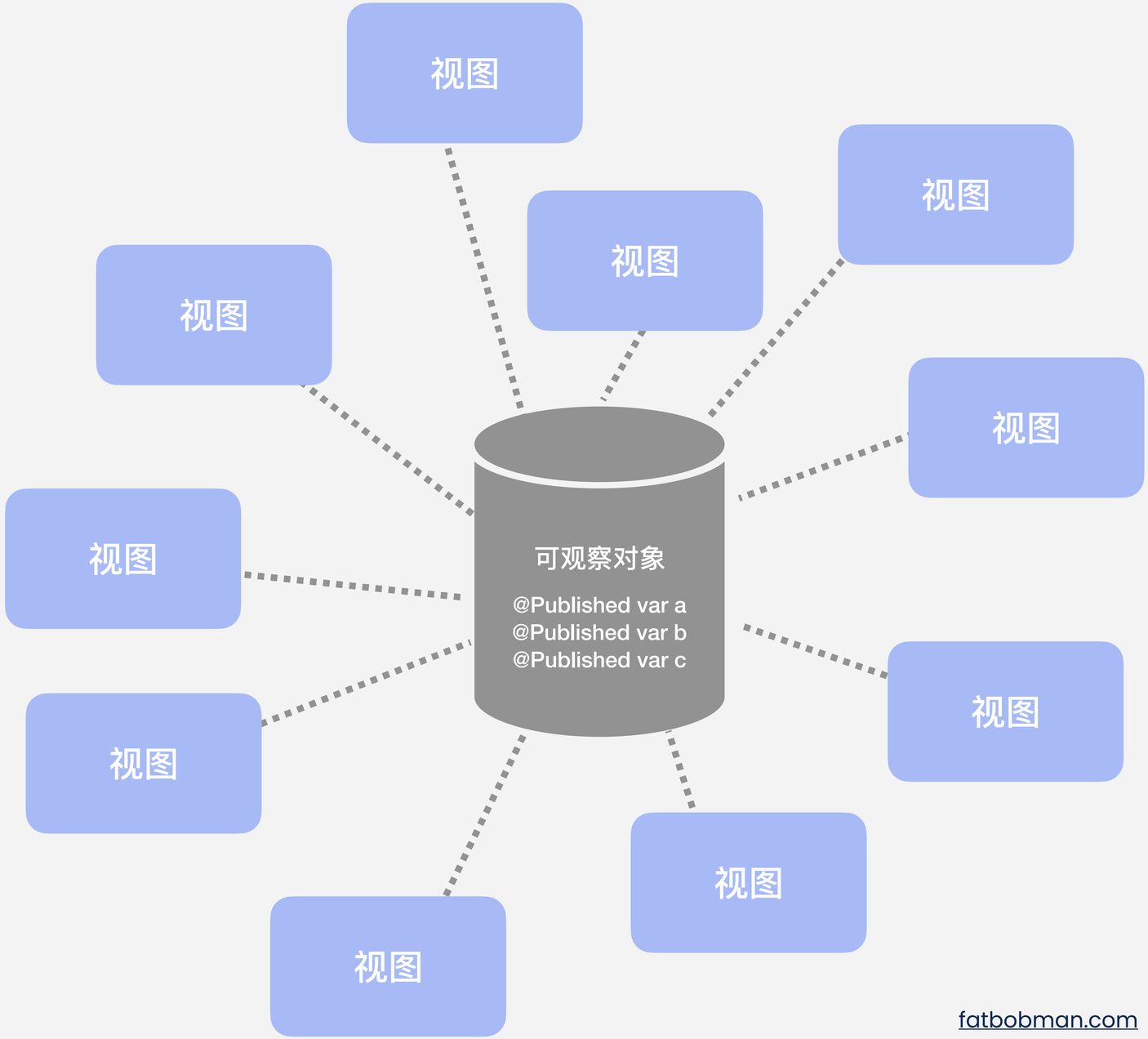
与 Combine 合作实现的观察机制

而对于影响范围较大甚至是全局的状态, 我们会将其提取到统一的容器中（ 可观察对象中 ）, 添加管理逻辑。对于引用类型的观察, SwiftUI 无法独立进行, 而是与同期推出的 Combine 框架合作完成。

导致大量视图无效更新的原因

```
class MyObservableObject: ObservableObject {
    @Published var name: String
    @Published var age: Int
    // 更多属性 ...
}

extension MyObservableObject {
  // 内部发布者实例，视图将订阅这个发布者
  var objectWillChange: Self.ObjectWillChangePublisher
}
```

导致大量视图无效更新的原因

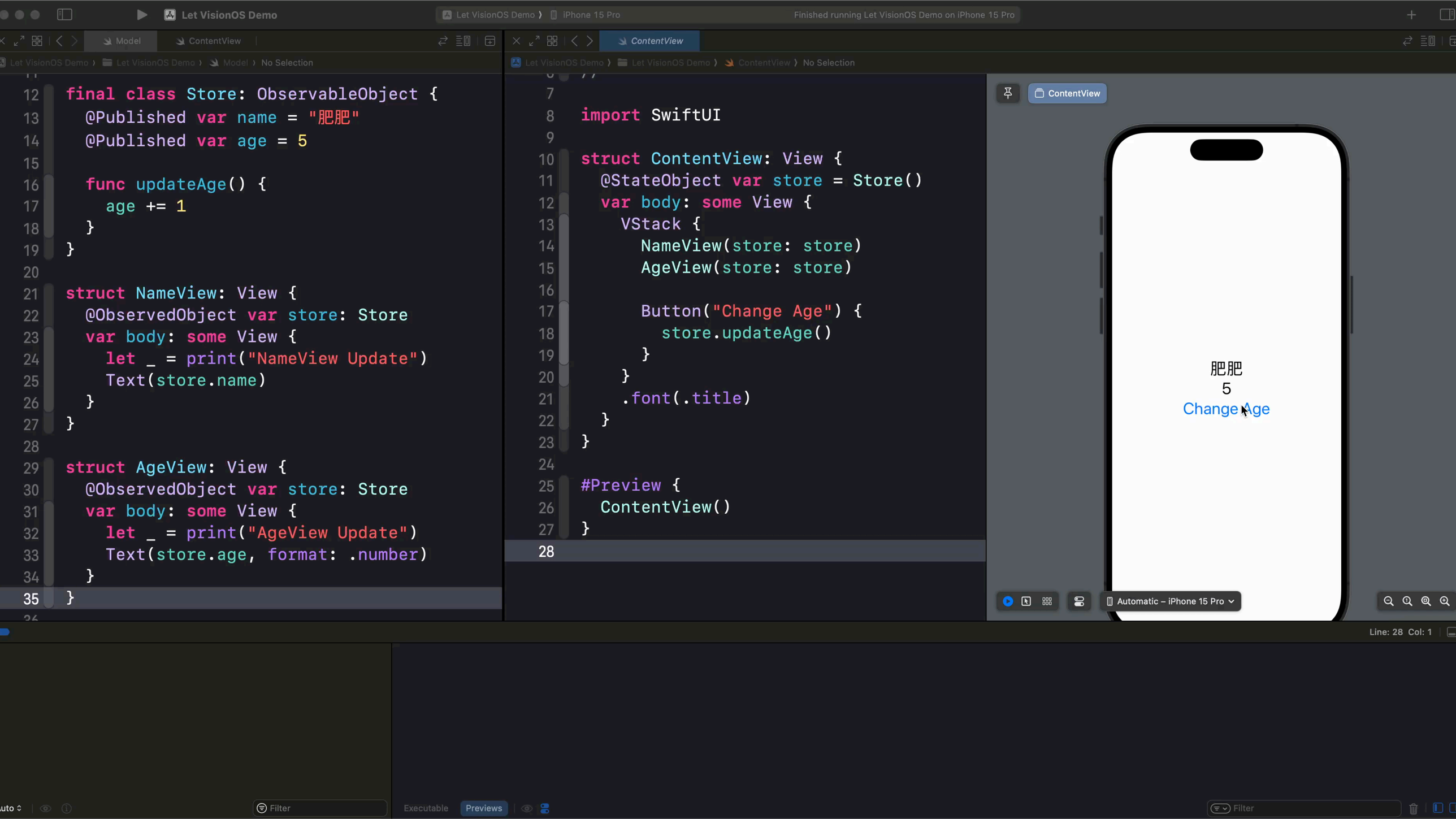


导致大量视图无效更新的原因

```
final class Store: ObservableObject {
  @Published var name = "肥肥"
  @Published var age = 5

  func updateAge() {
    age += 1
  }
}
```

导致大量视图无效更新的原因

```
struct NameView: View {
  @ObservedObject var store: Store
  var body: some View {
    let _ = print("NameView Update")
    Text(store.name)
  }
}

struct AgeView: View {
  @ObservedObject var store: Store
  var body: some View {
    let _ = print("AgeView Update")
    Text(store.age, format: .number)
  }
}
```


Let VisionOS Demo
Let VisionOS Demo › iPhone 15 Pro
Finished running Let VisionOS Demo on iPhone 15 Pro
Model
ContentView
Let VisionOS Demo › Let VisionOS Demo › Model › No Selection
final class Store: ObservableObject {
@Published var name = "肥肥"
@Published var age = 5
func updateAge() {
age += 1
}
}
struct NameView: View {
@ObservedObject var store: Store
var body: some View {
let _ = print("NameView Update")
Text(store.name)
}
}
struct AgeView: View {
@ObservedObject var store: Store
var body: some View {
let _ = print("AgeView Update")
Text(store.age, format: .number)
}
}
ContentView
Let VisionOS Demo › Let VisionOS Demo › ContentView › No Selection
import SwiftUI
struct ContentView: View {
@StateObject var store = Store()
var body: some View {
VStack {
NameView(store: store)
AgeView(store: store)
Button("Change Age") {
store.updateAge()
}
}
.font(.title)
}
}
#Preview {
ContentView()
}
ContentView
肥肥
5
Change Age
Automatic – iPhone 15 Pro
Line: 28 Col: 1
Filter
Executable
Previews
Filter

改善无效更新的手段

改善无效更新的手段

按需引入状态

避免将整个状态容器引入视图，而是仅将视图实际需要的状态作为参数传递。这种做法可以减少一些不必要的视图更新，但同时也增加工作量，并且它只适合纯展示的场景，也就是不能在视图中调用状态容器中的方法。

```
final class Store: ObservableObject {
  @Published var name = "肥肥"
  @Published var age = 5
}

struct NameView: View {
  let name: String
  var body: some View {
    let _ = print("NameView Update")
    Text(name)
  }
}
```

Let VisionOS Demo

Let VisionOS Demo ⟩ iPhone 15 Pro

Finished running Let VisionOS Demo on iPhone 15 Pro

Model | ContentView

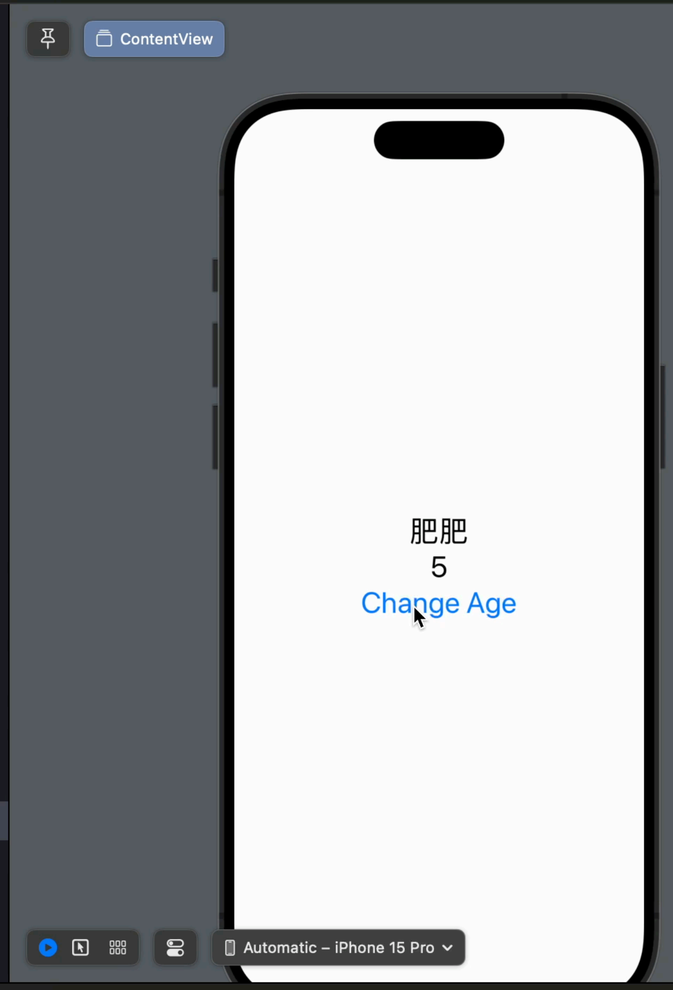
Let VisionOS Demo ⟩ Let VisionOS Demo ⟩ Model ⟩ No Selection

```swift
final class Store: ObservableObject {
  @Published var name = "肥肥"
  @Published var age = 5

  func updateAge() {
    age += 1
  }
}

struct NameView: View {
  let name: String
  var body: some View {
    let _ = print("NameView Update")
    Text(name)
  }
}

struct AgeView: View {
  @ObservedObject var store: Store
  var body: some View {
    let _ = print("AgeView Update")
    Text(store.age, format: .number)
  }
}
```

ContentView

Let VisionOS Demo ⟩ Let VisionOS Demo ⟩ ContentView ⟩ No Selection

```swift
import SwiftUI

struct ContentView: View {
  @StateObject var store = Store()
  var body: some View {
    VStack {
      NameView(name: store.name)
      AgeView(store: store)

      Button("Change Age") {
        store.updateAge()
      }
    }
    .font(.title)
  }
}

#Preview {
  ContentView()
}
```



Line: 28 Col: 1

Filter

Executable | Previews

Filter

改善无效更新的手段

利用等值协议

让视图符合 Equatable 协议，自定义比较逻辑以避免无关的属性更新导致的视图重绘。但这个方法对于基于类的容器是无效的。

```
@State var student = Student(name: "fat", age: 88)

struct StudentNameView: View, Equatable {
    let student: Student
    var body: some View {
        let _ = Self._printChanges()
        Text(student.name)
    }

    static func == (lhs: Self, rhs: Self) -> Bool {
        lhs.student.name == rhs.student.name
    }
}
```

改善无效更新的手段

拆分容器状态

将大的状态容器拆分为多个更小的状态容器。这样做可以减少更新的范围。它的缺点是牺牲了状态管理的便捷性。

```
class Store: ObservableObject {
  @Published var a: String
  @Published var b: String
}

class SubStore1: ObservableObject {
  @Published var a: String
}

class SubStore2: ObservableObject {
  @Published var b: string
}
```

改善无效更新的手段

状态的逐级比对

一些第三方框架会在状态变化时进行逐级比对，排除未变化的部分。但比对本身就是另一种对性能的消耗。当状态的层级较深时，比较的消耗也就越大，性能提升就不那么明显了。另外，这种方式也会加大学习成本。

```
public init<State>(
  _ store: Store<State, ViewAction>,
  observe toViewState: @escaping (_ state: State) -> ViewState,
  removeDuplicates isDuplicate: @escaping (_ lhs: ViewState, _ rhs: ViewState) -> B

) {
  self._send = { store.send($0, originatingFrom: nil) }
  self._state = CurrentValueRelay(toViewState(store.state.value))
  self._isInvalidated = store._isInvalidated
  self.viewCancellable = store.state
    .map(toViewState)
    .removeDuplicates(by: isDuplicate)
    .sink { [weak objectWillChange = self.objectWillChange, weak _state = self._sta

      guard let objectWillChange = objectWillChange, let _state = _state else { ret

      objectWillChange.send()
      _state.value = $0
    }
}
```

Observation 的观察方式

@Observable

withObservationTracking

```
@ObservationIgnored
private let _$observationRegistrar = Observation.ObservationRegistrar()

internal nonisolated func access<Member>(
    keyPath: KeyPath<Store , Member>
) {
  _$observationRegistrar.access(self, keyPath: keyPath)
}

internal nonisolated func withMutation<Member, MutationResult>(
  keyPath: KeyPath<Store , Member>,
  _ mutation: () throws -> MutationResult
) rethrows -> MutationResult {
  try _$observationRegistrar.withMutation(of: self, keyPath: keyPath, mutation)
}
```

```
    var name:String = ""
    {
        @storageRestrictions(initializes: _name)
        init(initialValue) {
          _name = initialValue
        }
        get {
          access(keyPath: \.name)
          return _name
        }
        set {
          withMutation(keyPath: \.name) {
            _name = newValue
          }
        }
    }
```

```
func withObservationTracking<T>(
  _ apply: () -> T,
  onChange: @autoclosure () -> () -> Void
) -> T

let sum = withObservationTracking {
    store.a + store.b
} onChange: {
    print("Store Changed a:\(store.a) b:\(store.b) c:\(store.c)")
}
```

Observation 的特征

局部观察性

仅对 apply 闭包中实际读取的属性进行观察

Observation 的特征

局部观察性

仅对 apply 闭包中实际读取的属性进行观察

变化前通知

onChange 闭包在属性值变化之前（与 willSet 调用时机类似）被调用

Observation 的特征

局部观察性

仅对 apply 闭包中实际读取的属性进行观察

变化前通知

onChange 闭包在属性值变化之前（与 willSet 调用时机类似）被调用

一次性观察

观察具备一次性特征，意味着一旦 onChange 闭包被触发，观察操作即告终止。

Observation 的特征

局部观察性

仅对 apply 闭包中实际读取的属性进行观察

变化前通知

onChange 闭包在属性值变化之前（与 willSet 调用时机类似）被调用

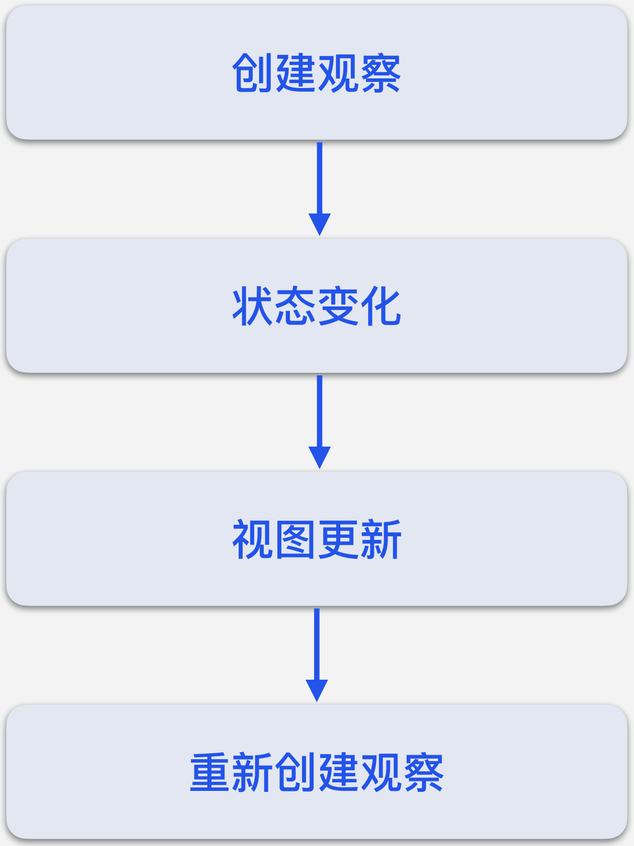
一次性观察

观察具备一次性特征，意味着一旦 onChange 闭包被触发，观察操作即告终止。

多属性、多实例监控

在单次观察过程中，可以同时监控多个可观察实例及其属性。任何被监控的属性发生变化都会触发并结束这次观察。

Observation 的特征



从 Combine 到 Observation

如果你的应用现在正面临因无效更新导致的性能问题, Observation 为我们提供了一个直接而简单的解决方案。那就是把现有的基于 Combine 的代码转换成使用 Observation 的形式, 你可能会立刻看到性能的提升。

```
final class Store: ObservableObject {
  @Published var name = "肥肥"
  @Published var age = 5

  func updateAge() {
    age += 1
  }
}
```

从 Combine 到 Observation

如果你的应用现在正面临因无效更新导致的性能问题, Observation 为我们提供了一个直接而简单的解决方案。那就是把现有的基于 Combine 的代码转换成使用 Observation 的形式, 你可能会立刻看到性能的提升。

```
final class Store {
  @Published var name = "肥肥"
  @Published var age = 5

  func updateAge() {
    age += 1
  }
}
```

从 Combine 到 Observation

如果你的应用现在正面临因无效更新导致的性能问题, Observation 为我们提供了一个直接而简单的解决方案。那就是把现有的基于 Combine 的代码转换成使用 Observation 的形式, 你可能会立刻看到性能的提升。

```
final class Store {
  var name = "肥肥"
  var age = 5

  func updateAge() {
    age += 1
  }
}
```

从 Combine 到 Observation

如果你的应用现在正面临因无效更新导致的性能问题, Observation 为我们提供了一个直接而简单的解决方案。那就是把现有的基于 Combine 的代码转换成使用 Observation 的形式, 你可能会立刻看到性能的提升。

```
@Observable
final class Store {
  var name = "肥肥"
  var age = 5

  func updateAge() {
    age += 1
  }
}
```

从 Combine 到 Observation

如果你的应用现在正面临因无效更新导致的性能问题, Observation 为我们提供了一个直接而简单的解决方案。那就是把现有的基于 Combine 的代码转换成使用 Observation 的形式, 你可能会立刻看到性能的提升。

```
struct ContentView: View {
  @StateObject var store = Store()
  var body: some View {
    VStack {
      NameView(store: store)
      AgeView(store: store)
       ...
    }
  }
}
```

从 Combine 到 Observation

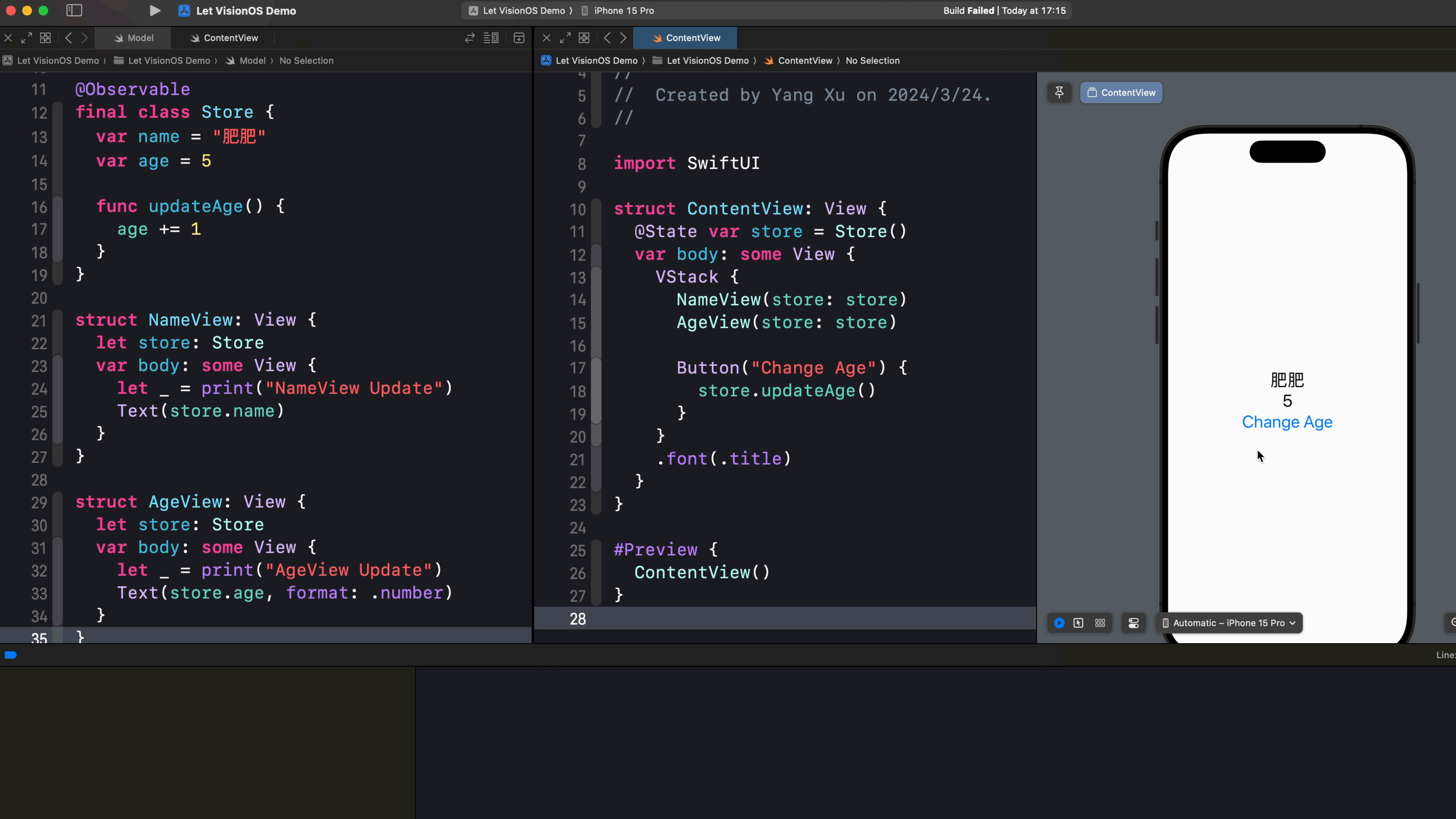
如果你的应用现在正面临因无效更新导致的性能问题, Observation 为我们提供了一个直接而简单的解决方案。那就是把现有的基于 Combine 的代码转换成使用 Observation 的形式, 你可能会立刻看到性能的提升。

```
struct ContentView: View {
  @State var store = Store()
  var body: some View {
    VStack {
      NameView(store: store)
      AgeView(store: store)
       ...
    }
  }
}
```

从 Combine 到 Observation

如果你的应用现在正面临因无效更新导致的性能问题, Observation 为我们提供了一个直接而简单的解决方案。那就是把现有的基于 Combine 的代码转换成使用 Observation 的形式, 你可能会立刻看到性能的提升。

```
struct NameView: View {
  let store: Store
  var body: some View {
    let _ = print("NameView Update")
    Text(store.name)
  }
}
```


Let VisionOS Demo
Let VisionOS Demo 〉 iPhone 15 Pro
Build Failed | Today at 17:15
Model
ContentView
Let VisionOS Demo 〉 Let VisionOS Demo 〉 Model 〉 No Selection
@Observable
final class Store {
var name = "肥肥"
var age = 5
func updateAge() {
age += 1
}
}
struct NameView: View {
let store: Store
var body: some View {
let _ = print("NameView Update")
Text(store.name)
}
}
struct AgeView: View {
let store: Store
var body: some View {
let _ = print("AgeView Update")
Text(store.age, format: .number)
}
}
ContentView
Let VisionOS Demo 〉 Let VisionOS Demo 〉 ContentView 〉 No Selection
// Created by Yang Xu on 2024/3/24.
import SwiftUI
struct ContentView: View {
@State var store = Store()
var body: some View {
VStack {
NameView(store: store)
AgeView(store: store)
Button("Change Age") {
store.updateAge()
}
}
.font(.title)
}
}
#Preview {
ContentView()
}
ContentView
肥肥
5
Change Age
Automatic – iPhone 15 Pro

Observation 带来的“新思维”

我们不应将 Observation 框架仅视为提升性能的工具。当 SwiftUI 获得如此先进的观察能力时，它实际上彻底改变了我们构思和设计应用状态的方式。这不只是修改几行代码那么简单，而是需要我们接受一种全新的思维方式，一种让应用更加灵活、可靠、可扩展且高效的思维方式。

Observation 带来的“新思维”

灵活的构筑形式

Observation 为我们提供了构建和组织应用状态的全新方式。现在，我们可以将一个可观察对象嵌套在另一个对象内，实现先前难以构建的状态关系。

```
@Observable
class A {
    var b = 1
}

@Observable
class B {
    var b = 1
    var a = A() // 可观察对象嵌套
}
```

Observation 带来的“新思维”

精准的观察构建逻辑

仅当视图实际读取可观察对象的属性（即触发它们的 getter 方法）时，才会创建观察操作。如果你仅仅是给属性赋值，或者调用可观察对象中的方法，并不会让你的视图与可观察属性创建联系。

```
struct AgeView: View {
  let store: Store
  var body: some View {
    let _ = print("AgeView Update")
    // 读取了属性，创建关联
    Text(store.age, format: .number)
  }
}

struct UpdateAge: View {
  let store: Store
  var body: some View {
    Button("Change Age") {
      // 没有调用 getter 方法，不会创建关联
      store.age += 1
    }
  }
}
```

Model | ContentView

Let VisionOS Demo › Let VisionOS Demo › Model › body

```swift
20
struct NameView: View {
  let store: Store
  var body: some View {
    let _ = print("NameView Update")
    Text(store.name)
  }
}

struct AgeView: View {
  let store: Store
  var body: some View {
    let _ = print("AgeView Update")
    Text(store.age, format: .number)
  }
}

struct UpdateAge: View {
  let store: Store
  var body: some View {
    let _ = print("UpdateAge Update")
    Button("Change Age") {
      store.age += 1
    }
  }
}
```

ContentView

Let VisionOS Demo › Let VisionOS Demo › ContentView › No Selection

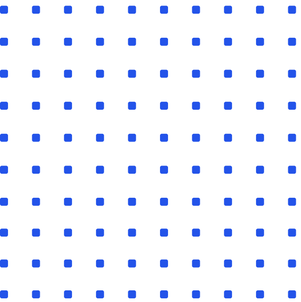
```swift
//
//  Created by Yang Xu on 2024/3/24.
//

import SwiftUI

struct ContentView: View {
  @State var store = Store()
  var body: some View {
    VStack {
      NameView(store: store)
      AgeView(store: store)
      UpdateAge(store: store)
    }
    .font(.title)
  }
}

#Preview {
  ContentView()
}
```

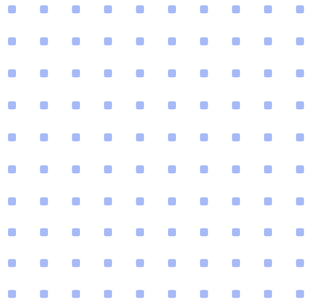
Let VisionOS Demo

Line: 20 Col: 2

能否支持低版本?

前向兼容性





新框架、新思维

王巍（Onevcat）

ObservationBP

Point-Free

Perception

fatbobman.com

SwiftData

Core Data

Core Data

资深且应用广泛

诞生于 2005 年，历史可追
溯至 NeXTSTEP 的 EOF

Core Data

资深且应用广泛

诞生于 2005 年，历史可追溯至 NeXTSTEP 的 EOF

稳定、安全

针对企业和商务用户、曾用于金融机构

Core Data

资深且应用广泛

诞生于 2005 年，历史可追溯至 NeXTSTEP 的 EOF

稳定、安全

针对企业和商务用户、曾用于金融机构

很强的扩展性

分层设计架构，新功能持久化历史跟踪和云同步都基于此

Core Data

资深且应用广泛

诞生于 2005 年，历史可追溯至 NeXTSTEP 的 EOF

稳定、安全

针对企业和商务用户、曾用于金融机构

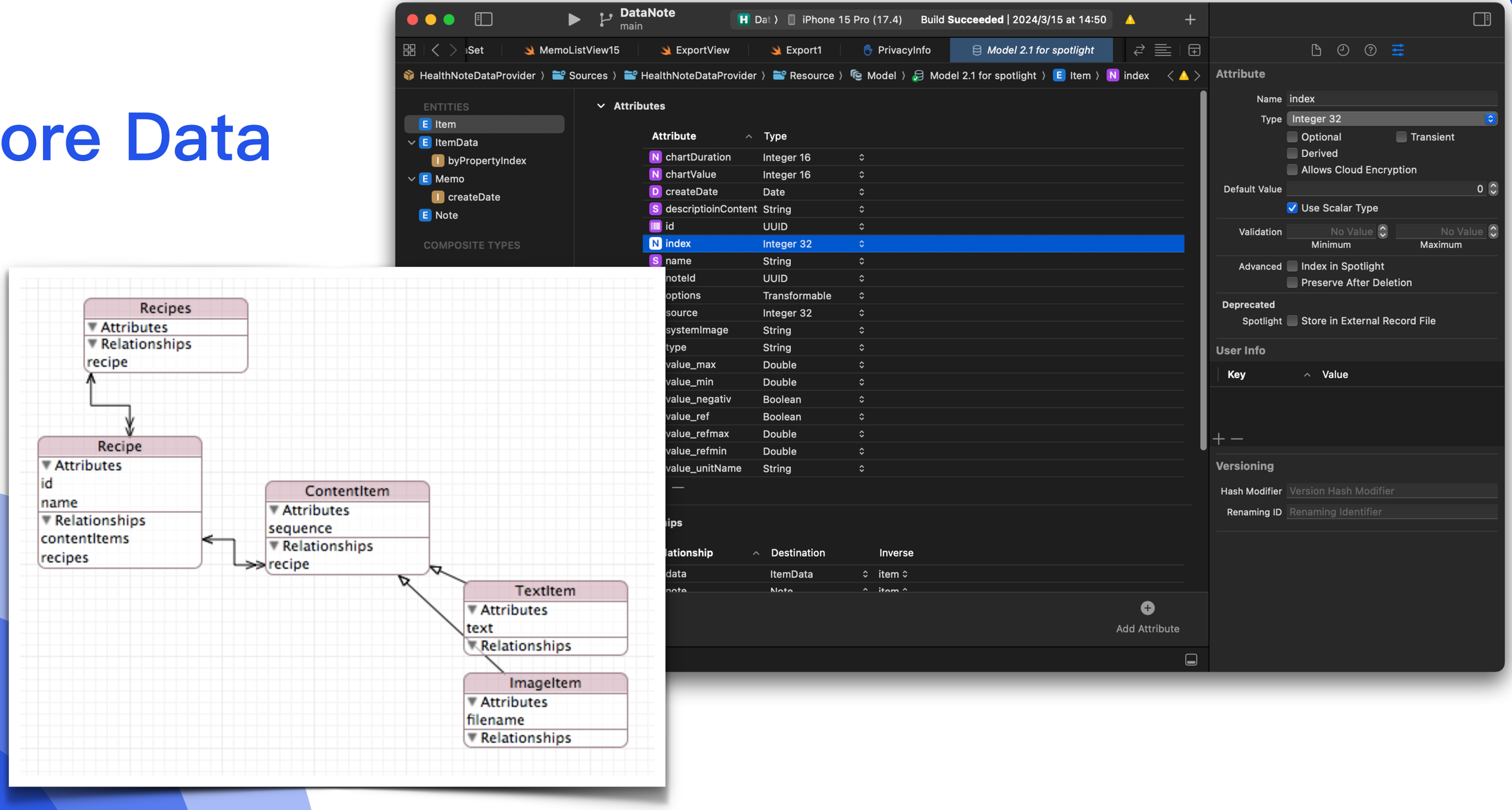
很强的扩展性

分层设计架构，新功能持久化历史跟踪和云同步都基于此

对象图管理框架

数据即对象，可视化建模适合处理复杂的对象关系，自动生成对应代码

Core Data


DataNote
main
iPhone 15 Pro (17.4)
Build Succeeded | 2024/3/15 at 14:50
MemoListView15
ExportView
Export1
PrivacyInfo
Model 2.1 for spotlight
HealthNoteDataProvider
Sources
HealthNoteDataProvider
Resource
Model
Model 2.1 for spotlight
Item
index
ENTITIES
Item
ItemData
byPropertyIndex
Memo
createDate
Note
COMPOSITE TYPES
Attributes
Attribute
Type
chartDuration Integer 16
chartValue Integer 16
createDate Date
descriptioinContent String
id UUID
index Integer 32
name String
noteId UUID
options Transformable
source Integer 32
systemImage String
type String
value_max Double
value_min Double
value_negativ Boolean
value_ref Boolean
value_refmax Double
value_refmin Double
value_unitName String
Destination
Inverse
data ItemData item
Add Attribute
Attribute
Name index
Type Integer 32
Optional
Transient
Derived
Allows Cloud Encryption
Default Value 0
Use Scalar Type
Validation No Value No Value
Minimum
Maximum
Advanced Index in Spotlight
Preserve After Deletion
Deprecated
Spotlight Store in External Record File
User Info
Key
Value
Versioning
Hash Modifier Version Hash Modifier
Renaming ID Renaming Identifier
Recipes
Attributes
Relationships
recipe
Recipe
Attributes
id
name
Relationships
contentItems
recipes
ContentItem
Attributes
sequence
Relationships
recipe
TextItem
Attributes
text
Relationships
ImageItem
Attributes
filename
Relationships

Core Data

资深且应用广泛

诞生于 2005 年，历史可追溯至 NeXTSTEP 的 EOF

稳定、安全

针对企业和商务用户、曾用于金融机构

很强的扩展性

分层设计架构，新功能持久化历史跟踪和云同步都基于此

对象图管理框架

数据即对象，可视化建模适合处理复杂的对象关系，自动生成对应代码

简化持久化操作

无需掌握数据库知识，支持多种存储格式

Core Data 在苹果生态中的优势

Core Data 在苹果生态中的优势

系统内置框架、低资源占用

应用体积更小、运行占用资源更少。在小组件、浏览器扩展等对内存限制严格的场景有独特的优势

Core Data 在苹果生态中的优势

系统内置框架、低资源占用

应用体积更小、运行占用资源更少。在小组件、浏览器扩展等对内存限制严格的场景有独特的优势

无缝的 CloudKit 集成

提供一键式整合方案，轻松实现数据的云存储、共享以及跨设备同步。对中小开发者极具吸引力

Core Data 当前面临的问题

Core Data 当前面临的问题

目标用户的转变

从面向企业转换成面向桌面和移动应用。框架显得过于复杂，现在开发者需要更简单、直接且高效的数据管理解决方案

Core Data 当前面临的问题

目标用户的转变

从面向企业转换成面向桌面和移动应用。框架显得过于复杂，现在开发者需要更简单、直接且高效的数据管理解决方案

开发工具的进步

Swift 已经成为主流，Core Data 无法利用语言的先进特性，特别是与类型安全和编译时检查相关的功能

Core Data 当前面临的问题

目标用户的转变

从面向企业转换成面向桌面和移动应用。框架显得过于复杂，现在开发者需要更简单、直接且高效的数据管理解决方案

开发工具的进步

Swift 已经成为主流，Core Data 无法利用语言的先进特性，特别是与类型安全和编译时检查相关的功能

开发框架的更新

Core Data 与 SwiftUI 之间缺乏天然的协同性，仍采用基于 Combine 的观察机制，影响应用性能

Core Data 当前面临的问题

目标用户的转变

从面向企业转换成面向桌面和移动应用。框架显得过于复杂，现在开发者需要更简单、直接且高效的数据管理解决方案

开发工具的进步

Swift 已经成为主流，Core Data 无法利用语言的先进特性，特别是与类型安全和编译时检查相关的功能

开发框架的更新

Core Data 与 SwiftUI 之间缺乏天然的协同性，仍采用基于 Combine 的观察机制，影响应用性能

SwiftData 的传承与变革

SwiftData 的传承与变革

继承了稳定性

SwiftData 基于 Core Data 构建，稳定性经过验证

SwiftData 的传承与变革

继承了稳定性

SwiftData 基于 Core Data 构建，稳定性经过验证

兼容 CoreData 数据

数据库完全兼容，只需进行代码层面调整

SwiftData 的传承与变革

继承了稳定性

SwiftData 基于 Core Data 构建，稳定性经过验证

兼容 CoreData 数据

数据库完全兼容，只需进行代码层面调整

与 Swift 紧密结合

利用 Swift 语言特性，提供安全的并发模式、强类型、编译时检查等功能

SwiftData 的传承与变革

继承了稳定性

SwiftData 基于 Core Data 构建，稳定性经过验证

兼容 CoreData 数据

数据库完全兼容，只需进行代码层面调整

与 Swift 紧密结合

利用 Swift 语言特性，提供安全的并发模式、强类型、编译时检查等功能

降低学习使用门槛

省略和隐藏了复杂、难以理解或很少使用的功能，隐藏了分层结构

SwiftData 的传承与变革

继承了稳定性

SwiftData 基于 Core Data 构建，稳定性经过验证

兼容 CoreData 数据

数据库完全兼容，只需进行代码层面调整

与 Swift 紧密结合

利用 Swift 语言特性，提供安全的并发模式、强类型、编译时检查等功能

降低学习使用门槛

省略和隐藏了复杂、难以理解或很少使用的功能，隐藏了分层结构

对 SwiftUI 更友好

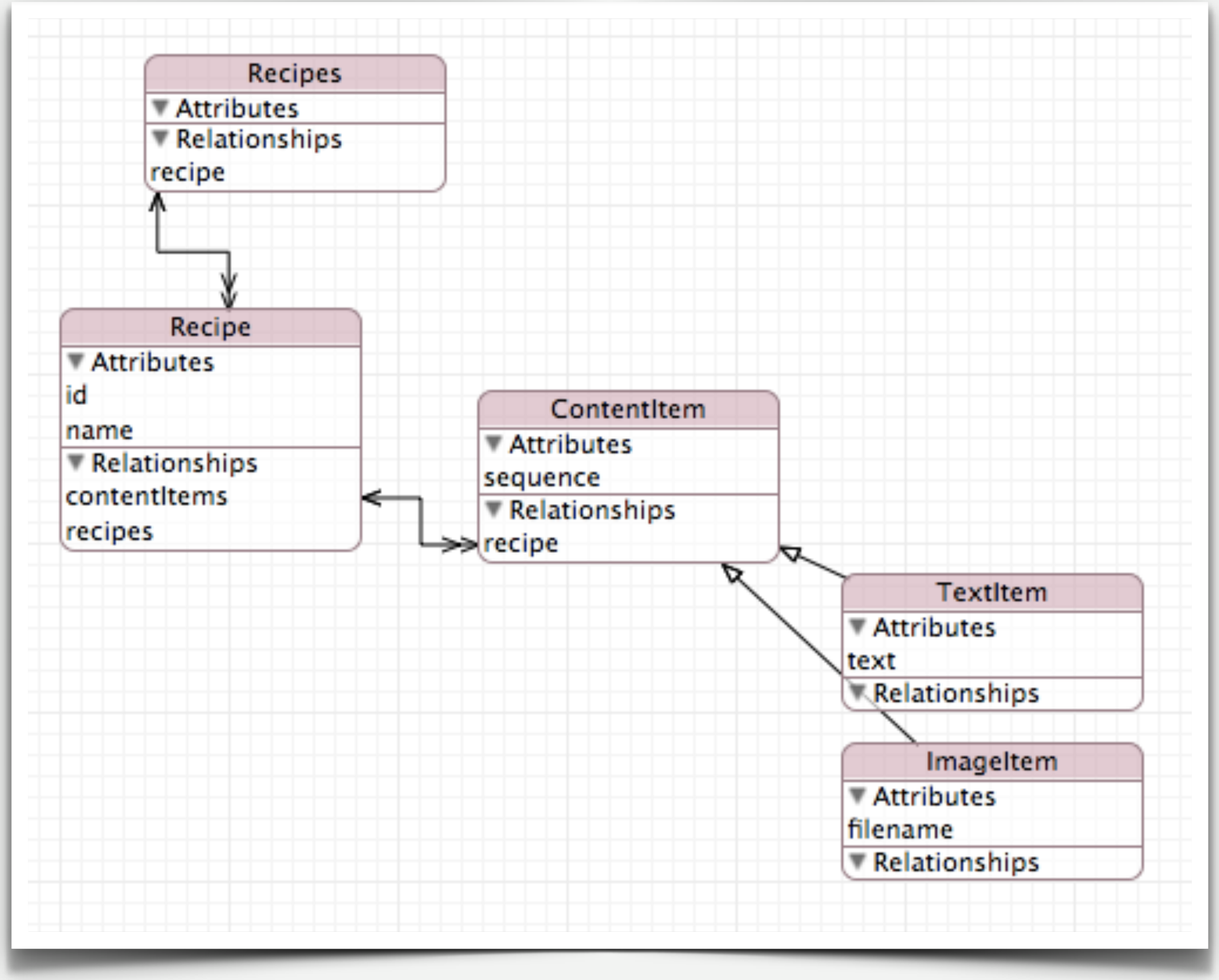
新的观察机制，在视图中展示时无需创建适配层

SwiftData 的创新功能

SwiftData 的创新功能

基于纯代码的数据建模方式

基于纯代码的数据建模方式



基于纯代码的数据建模方式

```
public final class Memo {
  public var createTimestamp: Date = .now
  public var content: String?
  public var star: Bool = false
  public var hasImages: Bool = false
  public var itemData: ItemData?

  public var assets: [ImageAsset]?
  public init(
    createTimestamp: Date,
    content: String?,
    itemData: ItemData?,
    star: Bool,
    hasImages: Bool = false
  ) {
    self.createTimestamp = createTimestamp
    self.content = content
    self.itemData = itemData
    self.star = star
    self.hasImages = hasImages
  }
}
```

基于纯代码的数据建模方式

```
@Model
public final class Memo {
  public var createTimestamp: Date = .now
  public var content: String?
  public var star: Bool = false
  public var hasImages: Bool = false
  public var itemData: ItemData?

  public var assets: [ImageAsset]?
  public init(
    createTimestamp: Date,
    content: String?,
    itemData: ItemData?,
    star: Bool,
    hasImages: Bool = false
  ) {
    self.createTimestamp = createTimestamp
    self.content = content
    self.itemData = itemData
    self.star = star
    self.hasImages = hasImages
  }
}
```

基于纯代码的数据建模方式

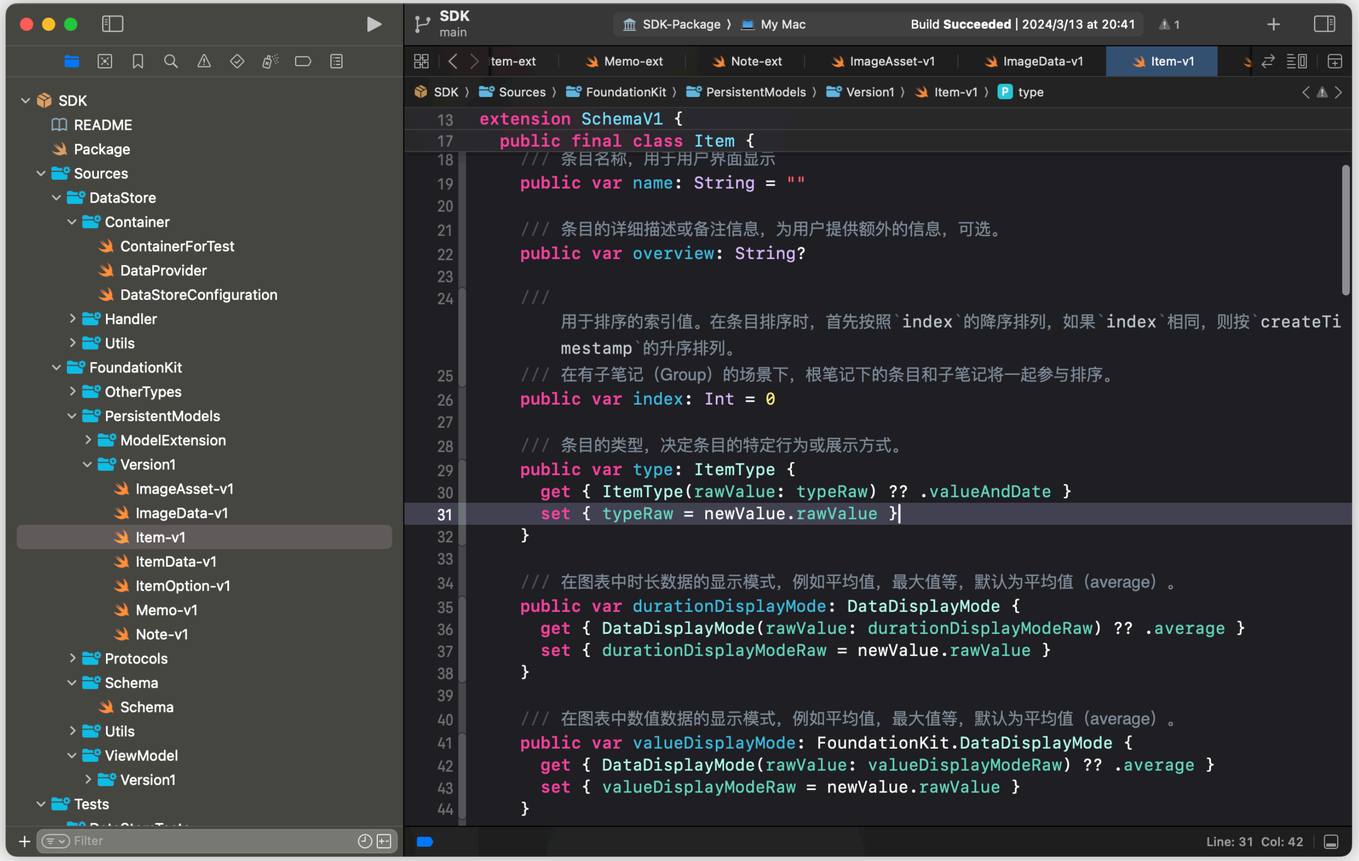
```
@Model
public final class Memo {
  public var createTimestamp: Date = .now
  public var content: String?
  public var star: Bool = false
  public var hasImages: Bool = false
  public var itemData: ItemData?
  @Relationship(deleteRule: .cascade)
  public var assets: [ImageAsset]?
  public init(
    createTimestamp: Date,
    content: String?,
    itemData: ItemData?,
    star: Bool,
    hasImages: Bool = false
  ) {
    self.createTimestamp = createTimestamp
    self.content = content
    self.itemData = itemData
    self.star = star
    self.hasImages = hasImages
  }
}
```

纯代码建模带来的“新思维”

基于纯代码的数据建模方式

封装成独立的模块

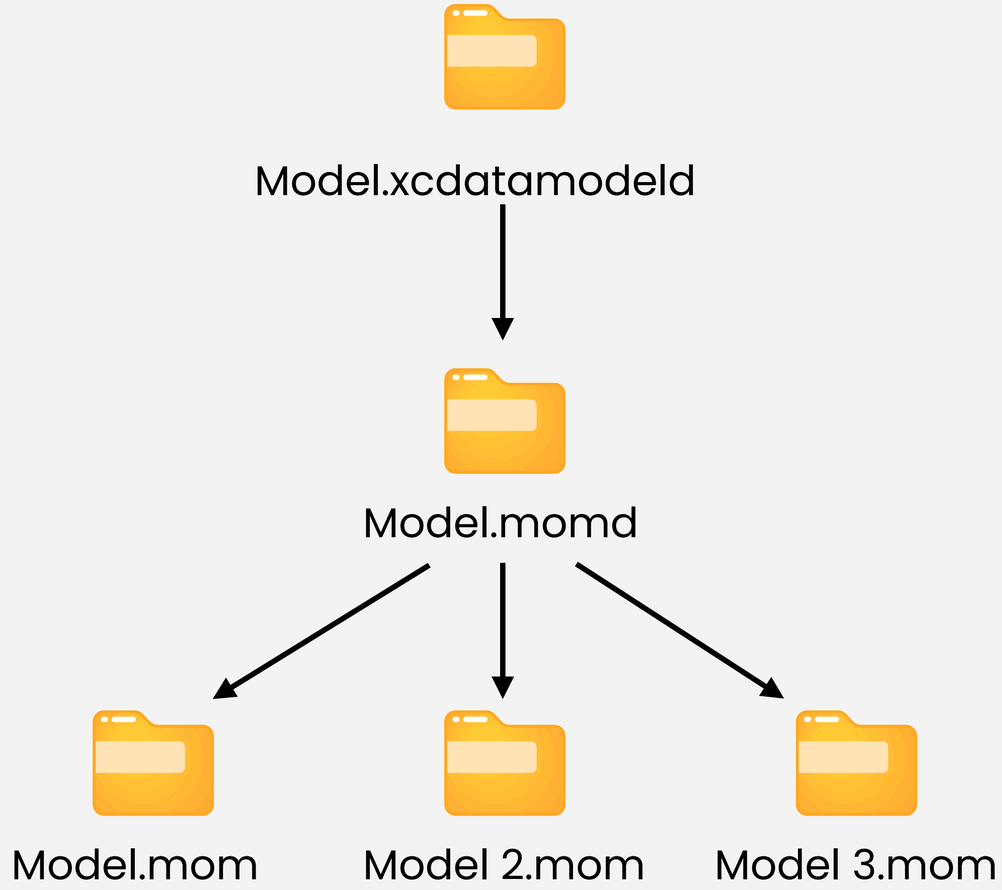
不使用模型文件，无需包含其他资源，促进开发者用模块的方式对数据操作进行更好的抽象和封装



基于纯代码的数据建模方式

封装成独立的模块

不使用模型文件，无需包含其他资源，促进开发者用模块的方式对数据操作进行更好的抽象和封装



基于纯代码的数据建模方式

更全面的单元测试覆盖

轻松为每个测试单元构建独立的数据容器，构建测试更方便，测试也更迅捷

```
class CoreDataStack {
    private static var _model: NSManagedObjectModel?
    static func model(name: String = CoreDataStackSetting.defaultModelName) -> NSManagedObjectModel {
        if _model == nil {
            do {
                _model = try loadModel(name: name, bundle: Bundle.main)
            } catch {
                let err = error.localizedDescription
                fatalError("❌数据库 momd 文件无法加载")
            }
        }

        return _model!
    }

    private static func loadModel(name: String, bundle: Bundle) throws -> NSManagedObjectModel {
        guard let modelURL = bundle.url(forResource: name, withExtension: "momd") else {
            fatalError("❌数据库 momd 文件无法加载")
        }
        guard let model = NSManagedObjectModel(contentsOf: modelURL) else {
            fatalError("❌数据库 momd 文件无法解析")
        }
        return model
    }

    public lazy var persistentContainer: NSPersistentCloudKitContainer = {
        let container = NSPersistentCloudKitContainer(
            name: modelName,
            managedObjectModel: Self.model(name: modelName)
        )
        // 其它配置代码
        ........
    }
}
```

基于纯代码的数据建模方式

更全面的单元测试覆盖

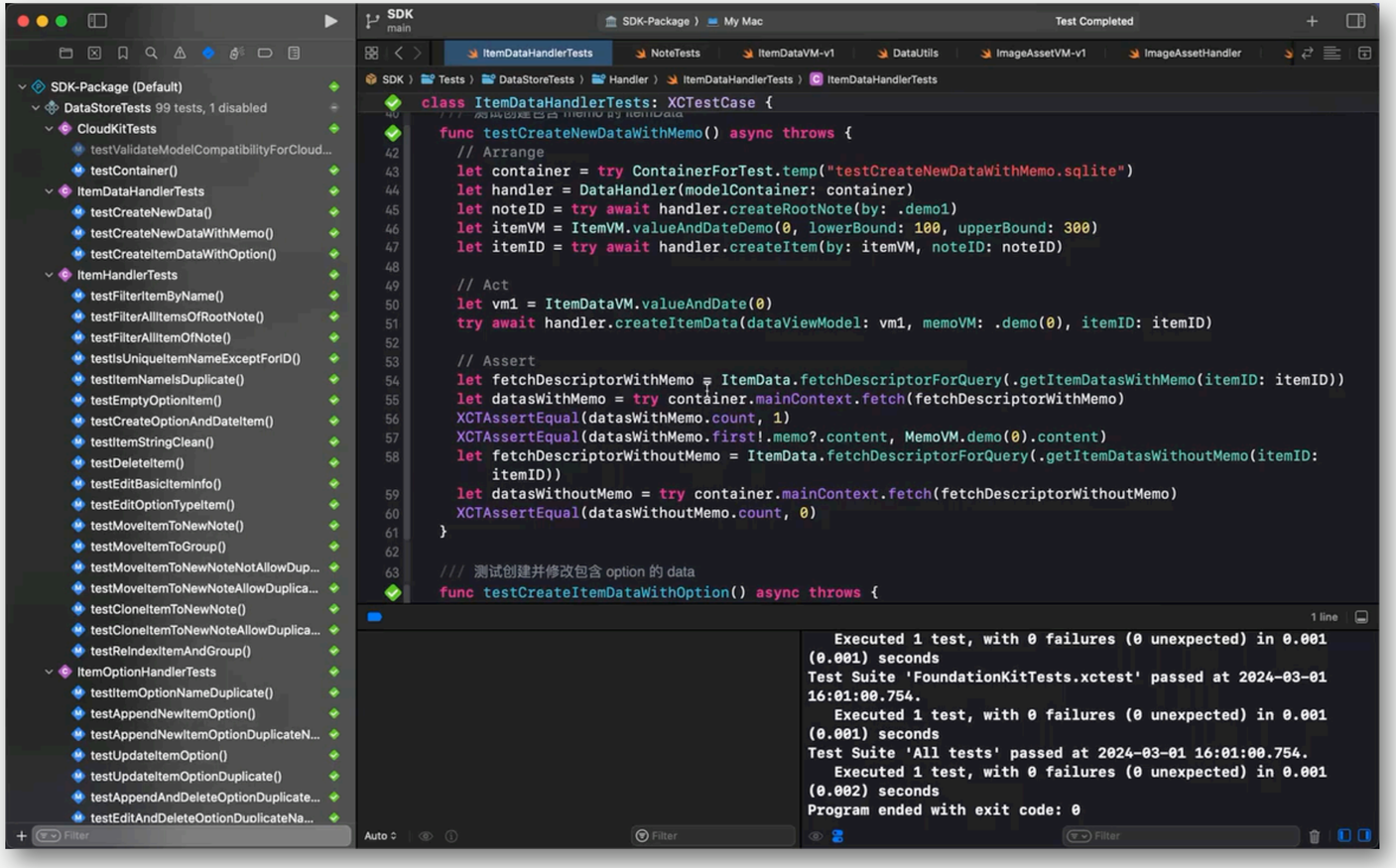
轻松为每个测试单元构建独立的数据容器，构建测试更方便，测试也更迅捷

```
enum ContainerForTest {
  static func temp(
    _ name: String,
    delete: Bool = true
  ) throws -> ModelContainer {
    let url = URL.temporaryDirectory.appending(component: name)
    if delete, FileManager.default.fileExists(atPath: url.path) {
      try FileManager.default.removeItem(at: url)
    }
    let schema = Schema(CurrentScheme.models)
    let configuration = ModelConfiguration(url: url)
    let container = try! ModelContainer(
      for: schema,
      configurations: configuration
    )
    return container
  }
}
```

基于纯代码的数据建模方式

更全面的单元测试覆盖

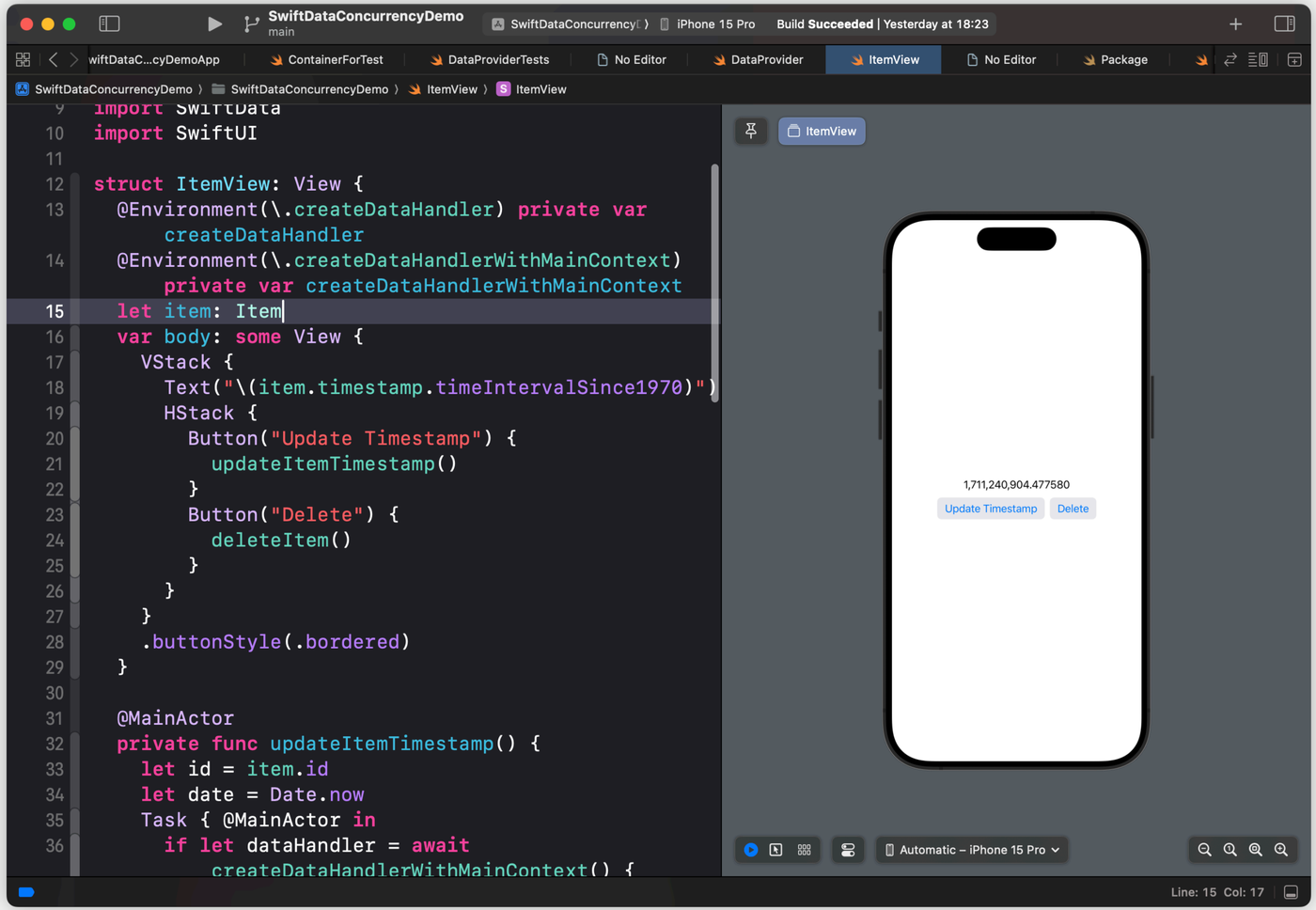
轻松为每个测试单元构建独立的数据容器，构建测试更方便，测试也更迅捷



基于纯代码的数据建模方式

更适合在视图中使用

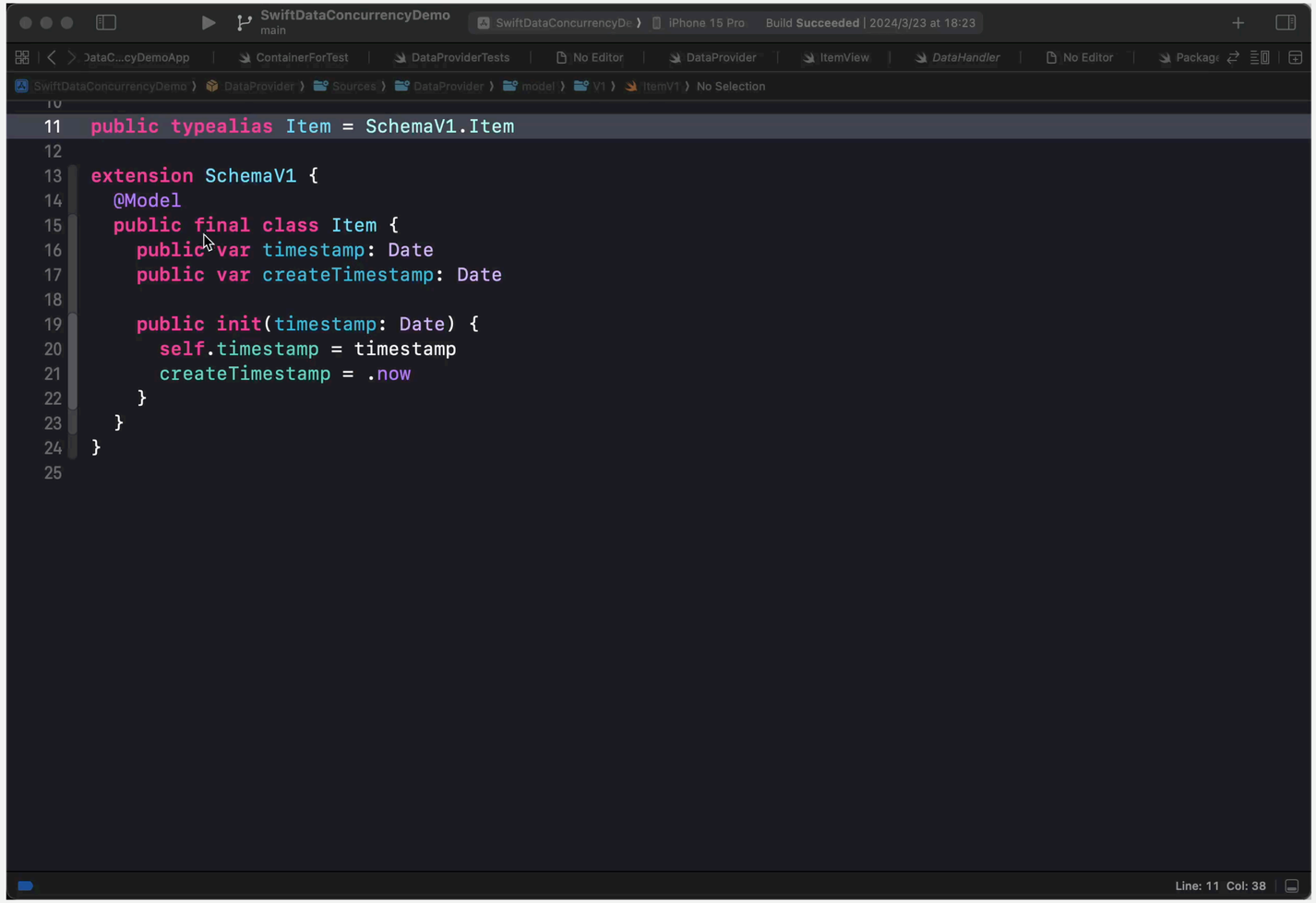
无需构建适配层；采用新的观察方式，响应更精准；对预览更友好



基于纯代码的数据建模方式

更适合在视图中使用

无需构建适配层；采用新的观察方式，响应更精准；对预览更友好



基于纯代码的数据建模方式

拓展数据管理框架的应用场景

因其易用的特性，可以在很多场景下取代 UserDefaults、NSUbiquitousKeyValueStore，甚至用作缓存机制

Cache

NSUbiquitousKeyValueStore

File Storage

UserDefaults

基于纯代码的数据建模方式

封装成独立的模块

不使用模型文件，无需包含其他资源，促进开发者用模块的方式对数据操作进行更好的抽象和封装

更全面的单元测试覆盖

轻松为每个测试单元构建独立的数据容器，构建测试更方便，测试也更迅捷

更适合在视图中使用

无需适配层，采用新的观察方式

拓展数据管理框架的应用场景

可以在很多场景下取代 UserDefaults、NSUbiquitousKeyValueStore，甚至用作缓存机制

基于 Actor 的并发操作

基于 Actor 的并发操作

SwiftData 提供了 @ModelActor 宏，鼓励开发者利用这一功能来创建一个 Actor 类型，进而在其中封装数据操作逻辑。

用 Actor 的串行操作特征将开发者从手动处理 perform 代码的沉重负担重解脱出来。

```
func delItem(id:NSManagedObjectID) {
    let bgContext = container.newBackgroundContext()
    bgContext.perform {
        let item = bgContext.object(with: id)
        bgContext.delete(item)
        try! bgContext.save()
    }
}
```

基于 Actor 的并发操作

SwiftData 提供了 @ModelActor 宏，鼓励开发者利用这一功能来创建一个 Actor 类型，进而在其中封装数据操作逻辑。

用 Actor 的串行操作特征将开发者从手动处理 perform 代码的沉重负担重解脱出来。

```
@ModelActor
actor DataHandler {
    func updateItem(identifier: PersistentIdentifier,
                    timestamp: Date) throws {
        guard let item = self[identifier, as: Item.self] else {
            throw MyError.objectNotExist
        }
        item.timestamp = timestamp
        try modelContext.save()
    }

    func newItem(timestamp: Date) throws -> Item {
        let item = Item(timestamp: timestamp)
        modelContext.insert(item)
        try modelContext.save()
        return item
    }
}
```

基于 Actor 的并发操作

SwiftData 提供了 @ModelActor 宏，鼓励开发者利用这一功能来创建一个 Actor 类型，进而在其中封装数据操作逻辑。

用 Actor 的串行操作特征将开发者从手动处理 perform 代码的沉重负担重解脱出来。

```
// @ModelActor 添加的代码
actor DataHandler {
  public nonisolated let modelExecutor: any SwiftData.ModelExecutor

  public nonisolated let modelContainer: SwiftData.ModelContainer

  public init(modelContainer: SwiftData.ModelContainer) {
      let modelContext = ModelContext(modelContainer)
      self.modelExecutor = DefaultSerialModelExecutor(
        modelContext: modelContext
      )
      self.modelContainer = modelContainer
  }
}
```

基于 Actor 的并发操作

新的注入方式

基于 Actor 的封装方式同时也改变了数据操作逻辑在视图或其他状态管理模块中的集成方式，推动了一种更安全、更可靠的数据注入模式。

```
public func dataHandlerCreator(
    preview: Bool = false
 ) -> @Sendable () async -> DataHandler {
    let container = preview ? previewContainer : sharedModelContainer
    return { DataHandler(modelContainer: container) }
 }


public struct DataHandlerKey: EnvironmentKey {
  public static let defaultValue: @Sendable () async -> DataHandler? = { nil }
}

extension EnvironmentValues {
  public var createDataHandler: @Sendable () async -> DataHandler? {
    get { self[DataHandlerKey.self] }
    set { self[DataHandlerKey.self] = newValue }
  }
}
```

模块化

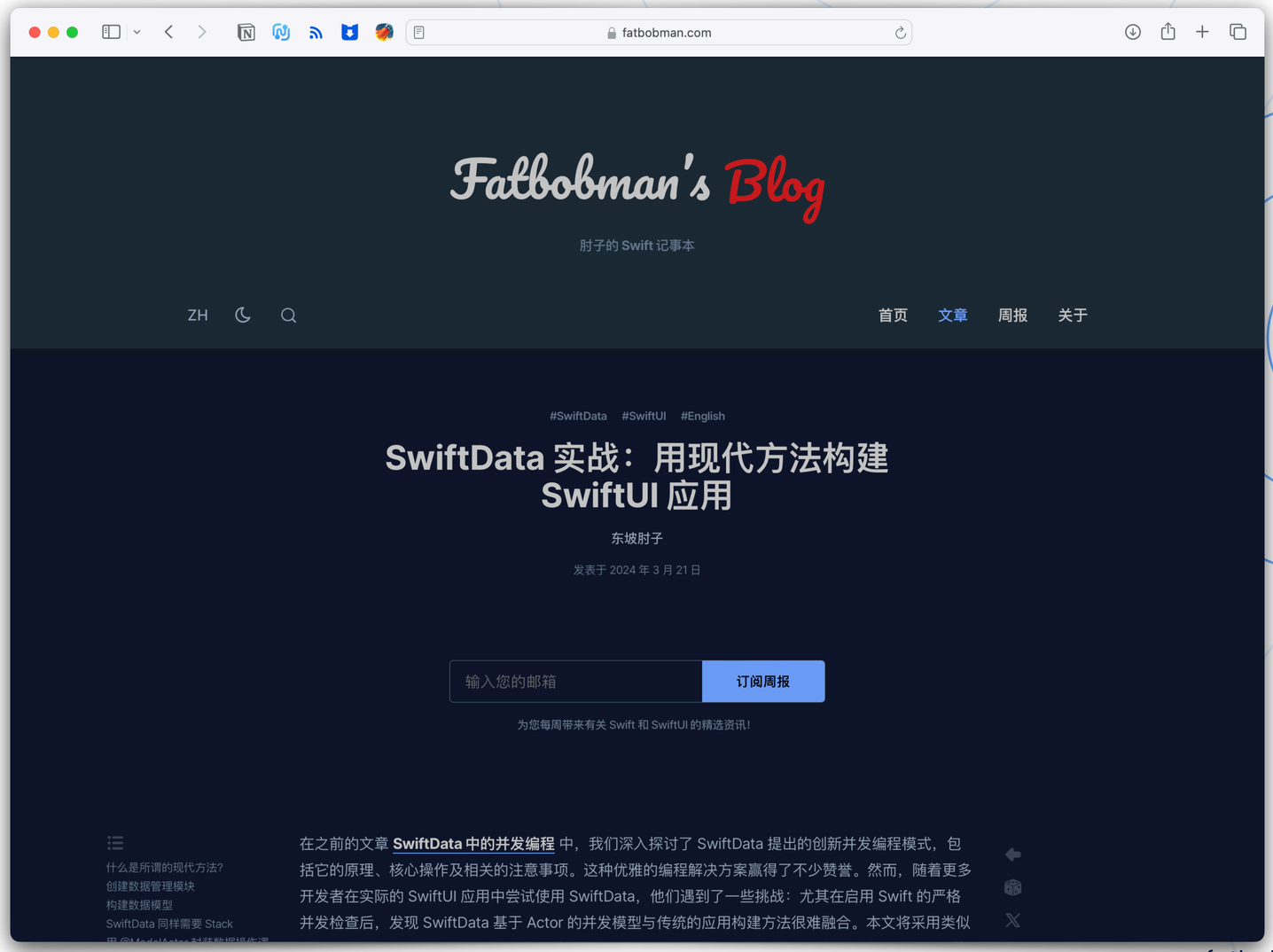
全面可测试

线程安全

与架构无关

预览处理

展示与操作分离



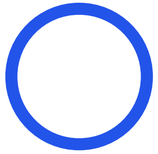
- 它们诞生于对旧有框架的现代化需求
- 紧密结合了 Swift 语言的先进特性
- 具备了塑造现代应用开发逻辑的潜力

徐杨 东坡肘子

Discord: 1.5K members

X: @fatbobman

Blog: fatbobman.com



Thanks EveryOne

Thanks Let’s VisionOS 2024