MVVM设计模式的概念和MVC的区别

MVVM (Model-View-ViewModel) 和 MVC (Model-View-Controller) 都是用于软件开发的架构设计模式，它们的目标都是分离应用程序中的不同关注点，以提高代码的可维护性和可扩展性。

MVVM 设计模式

MVVM基本介绍

MVVM 是一种由微软开发并广泛应用于前端框架（例如 Vue.js 和 Angular）的设计模式。MVVM 模式的三个主要组件分别是：

- **Model（模型）**：负责数据逻辑，包含应用程序的业务逻辑和数据处理部分。它代表了应用程序的数据结构和状态。
- **View（视图）**：表示用户界面（UI），即用户看到的内容。视图只负责展示数据和捕获用户输入，通常与用户交互直接相关。
- **ViewModel（视图模型）**：作为 View 和 Model 之间的桥梁。它既包含视图的逻辑，又负责将模型的数据转换为视图可以直接使用的格式。ViewModel 通过数据绑定的方式自动同步 View 和 Model。

MVVM 的特点

- **数据绑定**：视图通过数据绑定自动更新，而无需手动操作 DOM。这意味着当 Model 的数据发生变化时，View 也会自动更新，反之亦然。
- **双向数据绑定**：ViewModel 提供了双向绑定的能力，即 View 的修改会自动更新 Model，Model 的修改会自动反映在 View 中。

MVC 设计模式

MVC基本介绍

MVC 是一种传统的设计模式，广泛应用于后端开发以及 Web 应用程序（例如 Django 和 Rails）。MVC 的三个核心组件分别是：

- **Model（模型）**：负责应用程序的数据和业务逻辑，处理数据的存储和管理。
- **View（视图）**：负责显示数据和用户界面的部分。它获取来自 Model 的数据，并将其呈现给用户。
- **Controller（控制器）**：处理用户输入和操作逻辑。它接收来自 View 的用户输入，并根据需要更新 Model 的数据或改变 View 的显示。

MVC 的特点

- **单向数据流**：Controller 负责管理用户输入并将其传递给 Model，Model 更新后再通知 View 刷新界面。
- **较少自动化**：相比 MVVM 的双向数据绑定，MVC 通常需要手动调用来更新 View。

MVVM 与 MVC 的区别

- **数据绑定**：MVVM 支持双向数据绑定，View 和 ViewModel 之间通过自动化的方式进行数据同步；而 MVC 通常采用单向数据流，用户操作由 Controller 处理后再影响 Model 和 View。

- **复杂度**：MVC 的逻辑较为直接，适合简单的应用开发；MVVM 增加了 ViewModel 层，适合前端框架和大型应用中的复杂交互。
- **职责划分**：在 MVC 中，Controller 是输入和逻辑的处理中心；而在 MVVM 中，ViewModel 承担了更多逻辑工作，View 主要负责数据的展示。
- **应用场景**：MVC 主要用于服务器端开发，而 MVVM 则更适合现代前端开发，特别是在需要动态数据展示的应用中，如 SPA（单页应用）。

总结

- **MVC**：强调视图和模型的分离，通过控制器来协调两者的交互，适合服务器端开发。
- **MVVM**：强调视图和数据的自动同步，依赖双向数据绑定机制，适合现代前端框架的开发。

M/V/VM的对应代码及其逻辑关系

项目 MVVM（Model-View-ViewModel）设计模式架构

Memorize 项目使用了 MVVM（Model-View-ViewModel）设计模式。以下是 M、V、VM 在项目中的具体代码对应，以及它们之间的逻辑关系：

Model（模型）

Model 负责处理数据和业务逻辑，在项目中对应的是 `MemoryGame<CardContent>` 结构体（位于 `MemoryGame.swift` 文件）。该结构体负责存储游戏卡片，并处理游戏的核心逻辑，如生成卡片内容、卡片选择、洗牌等操作。它专注于数据和逻辑处理，不直接与界面交互。

`MemoryGame` 结构体管理游戏卡片的生成和洗牌，`Card` 结构体则表示每张卡片的状态（例如是否翻开、是否匹配等）。

View（视图）

View 负责应用程序的用户界面，显示数据并与用户交互。在项目中，`EmojiMemoryGameView` 和 `CardView` 组件（位于 `EmojiMemoryGameView.swift` 文件）充当 View 的角色，它们展示卡片的外观，并响应用户的操作，如点击卡片或按下“Shuffle”按钮。

`EmojiMemoryGameView` 负责显示游戏的整体界面，通过 `viewModel` 获取卡片信息并触发操作（例如洗牌）。`CardView` 则是单张卡片的视图，负责展示卡片的外观（如显示 Emoji 或空白卡片）。

ViewModel（视图模型）

ViewModel 负责协调视图与模型之间的交互，管理应用程序的状态和逻辑。它将数据从 Model 提供给 View，同时处理用户输入。在项目中，`EmojiMemoryGame` 类（位于 `EmojiMemoryGame.swift` 文件）担任 ViewModel 的角色，连接了 `MemoryGame` 模型与 `EmojiMemoryGameView` 视图。

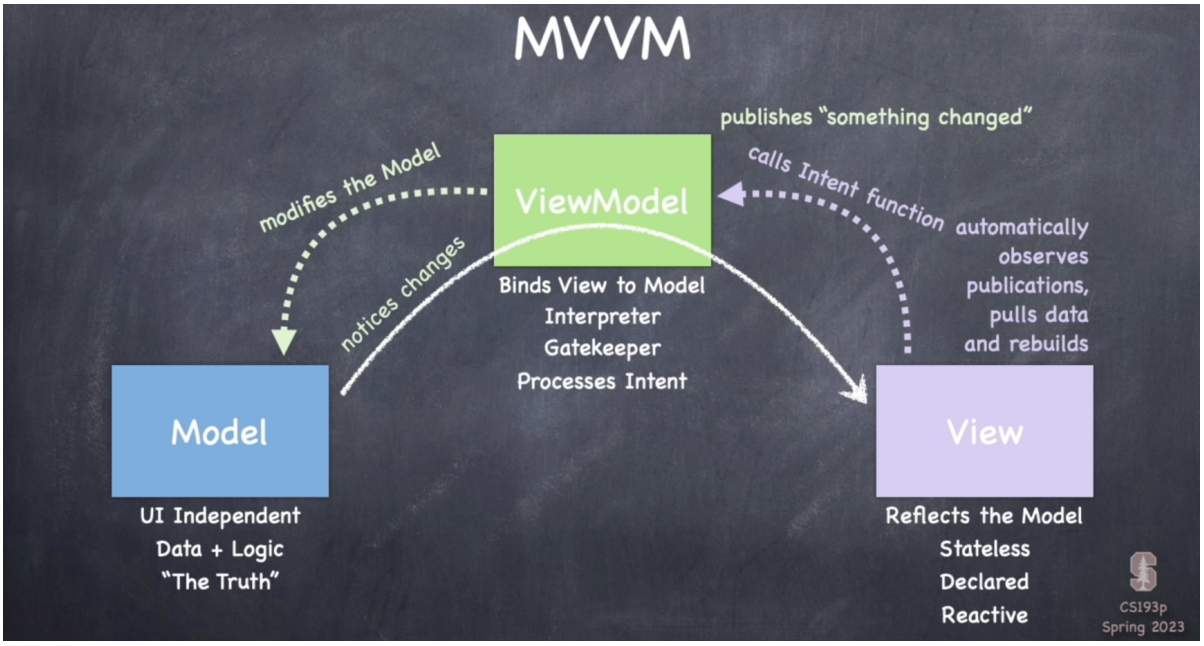
`EmojiMemoryGame` 中的 model 是 `MemoryGame` 模型的实例，负责初始化游戏逻辑。通过 `@Published` 属性包装器，确保数据更新时视图能够自动刷新。`cards` 属性为 View 提供了需要显示的卡片信息，`shuffle` 和 `choose` 方法则处理用户操作并与 Model 交互。

M、V、VM 之间的逻辑关系

1. **Model（MemoryGame）** 负责处理数据和业务逻辑，专注于游戏的核心逻辑和状态管理。它存储游戏数据（如卡片内容和状态），并定义了处理游戏行为的操作，如卡片生成、选择和洗牌。Model 不与界面直接交互，只负责处理数据层面的工作。
2. **ViewModel（EmojiMemoryGame）** 充当 Model 和 View 之间的桥梁。它将 Model 的数据（如卡片信息）通过 @Published 属性包装器传递给 View，确保当数据发生变化时，视图能够自动更新。ViewModel 还处理用户输入操作（如点击卡片、洗牌），并与 Model 交互，修改游戏状态。ViewModel 负责管理应用程序的状态，并确保逻辑与 UI 的解耦。
3. **View（EmojiMemoryGameView 和 CardView）** 负责显示用户界面，并响应用户交互。它从 ViewModel 获取要显示的数据，如卡片信息，并将用户的操作（如点击卡片或点击“Shuffle”按钮）传递给 ViewModel。View 只关心如何展示数据和处理交互，不直接参与逻辑处理。

即 **Model** 管理数据和逻辑；**ViewModel** 将 **Model** 的数据传递给 **View**，并响应用户的操作；**View** 显示界面并与用户交互，将用户操作反馈给 **ViewModel**，从而间接影响 **Model**。这种架构保证了视图与业务逻辑的分离，使得代码更加清晰、易维护。

图片的名词解释



Model（模型）

- UI Independent：表示 Model 是独立于用户界面的，它与 UI 没有直接的交互。它负责业务逻辑和数据处理，任何与用户界面无关的功能都在 Model 中实现。
- Data + Logic：Model 负责存储和管理应用程序的核心数据以及业务逻辑。它与数据库、后端 API 交互，处理数据的获取、更新和存储。
- "The Truth"（真相）：Model 代表了应用程序的真实数据来源，所有的数据和状态都应该从 Model 获取并管理，它是应用程序中“权威的数据源”。

ViewModel（视图模型）

- Binds View to Model（将视图与模型绑定）：ViewModel 的主要任务是将 View 和 Model 绑定在一起。它从 Model 中获取数据，并将其提供给 View 进行展示，同时处理来自 View 的用户输入并反馈给 Model。

- Interpreter（解释器）：ViewModel 充当解释器的角色，它接收来自 View 的用户操作，解释并将这些操作转换为对 Model 的调用。它还将 Model 的变化解释为可以更新 View 的数据。
- Gatekeeper（守门人）：ViewModel 控制 Model 和 View 之间的交互，它确保 View 只能访问到 Model 的合适数据，并且用户输入被正确地处理。ViewModel 保护了 Model 不直接与 View 交互。
- Processes Intent（处理意图）：ViewModel 接收用户在 View 上的操作意图，比如点击按钮或输入数据，处理这些意图并相应地与 Model 交互。它决定这些用户意图如何影响数据和业务逻辑。

View（视图）

- Reflects the Model（反映模型）：View 的主要任务是展示 Model 提供的数据，界面上的所有显示内容都是 Model 状态的反映。View 通过 ViewModel 获取数据，进行渲染和展示。
- Stateless（无状态）：View 本身不管理状态，它仅仅负责展示从 ViewModel 获取的数据。所有的状态都在 Model 和 ViewModel 中管理，这让 View 更加简单和轻量化。
- Declared（声明式的）：View 是声明式的，这意味着开发者通过声明某种 UI 结构来定义界面的样式和内容，而不用关心具体的 UI 更新细节。这种声明式方式让开发者专注于 ViewModel 和 Model，而不必直接操纵 UI 组件。
- Reactive（响应式的）：View 会自动响应 ViewModel 中数据的变化。当 Model 数据更新后，ViewModel 会将变化通知 View，View 会自动更新 UI。开发者不需要手动操作 DOM 来更新界面。

其他

- Modifies the Model（修改模型）：这是 ViewModel 的职责之一，ViewModel 根据用户输入或者其他事件，修改 Model 中的数据。通过这种方式，Model 数据发生变化，进而影响 View 的显示。
- Notices changes（注意到变化）：当 Model 中的数据发生变化时，ViewModel 会监控并“注意到”这些变化。它会捕获到这些变化并决定是否需要更新 View。
- Publishes "something changed"（发布“某些东西发生了变化”）：当 ViewModel 修改 Model 后，它会发布一个信号，告诉系统 Model 的数据发生了变化。View 通过数据绑定机制监听这些变化并更新 UI。
- Calls Intent function（调用意图函数）：View 接受用户输入（例如点击按钮、输入文字）后，会调用意图函数。这些函数由 ViewModel 实现，负责解释用户的操作意图并采取相应的动作（比如更新 Model 或者处理业务逻辑）。
- Automatically observes publications, pulls data and rebuilds（自动观察发布，获取数据并重建）：View 是响应式的，它会自动观察 ViewModel 发布的变化通知。当 Model 中的数据发生变化时，View 通过 ViewModel 获取新的数据，并自动重新渲染 UI 以展示最新的数据。
- Processes Intent（处理意图）：这与 ViewModel 的责任有关。用户在 View 中进行的操作（点击、输入等）被作为“意图”传递到 ViewModel 中，ViewModel 处理这些意图，并决定如何修改 Model 或执行其他操作。

项目应用示例