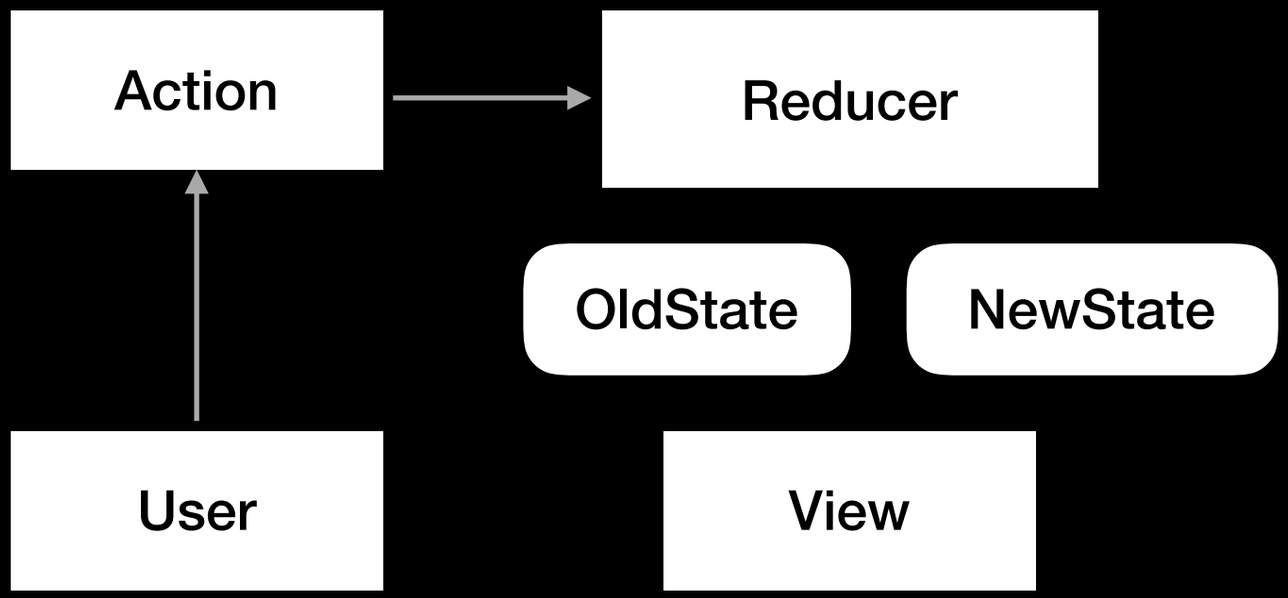
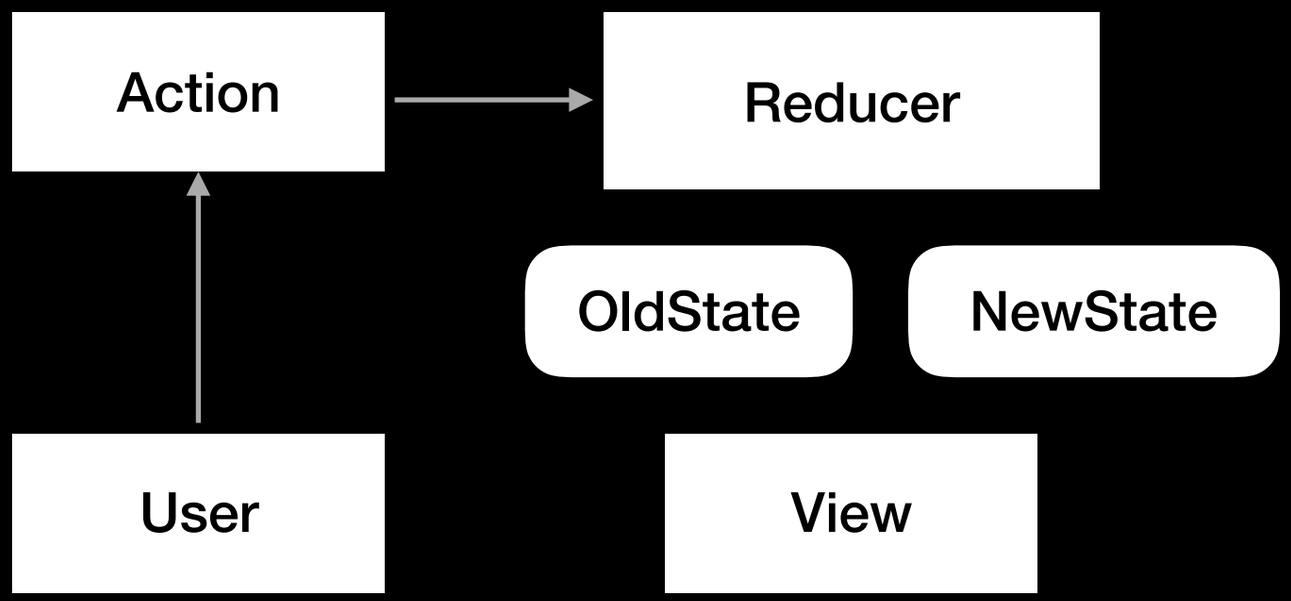
怎么搭一个HelloSlide?

- 名词总结：

名字	含义
Outline	类名 用户输入的文本对应的数据结构
Slide	类名 用户的输入经过排版器处理后的数据结构 界面显示的直接数据来源
parser	类名 包括Outline、Slide的实例和操作他们的函数
view	界面 仅指显示
state	界面数据的抽象称呼 仅指和界面刷新息息相关的数据
action	用户操作的抽象称呼
reducer	状态机函数 根据旧state和action输出新的state

- 你是个User，我们的用户大大，你进入了键盘编辑的界面
- 你在这个界面里输入了一句话“HelloSlide”，一个Action就发出来了
- Reducer收到了Action，这个了不起的状态机就开始工作了！它检查了OldState，并根据当前的Action的类型，更新得NewState
- NewState来了，咱的View就因此而刷新了
- （PS：考虑到User并不能实际滴去发Action，他只能和界面去交互，所以真实滴Action是通过View发出去的啦）





- 把用户所有可能修改的操作抽象成Action
- 把和用户的显示第一手相关的数据抽象成State
- 用户给Reducer发送Action，Reducer因此修改State，才能导致View的改变
- 这就是一个可可爱爱的Redux框架啦

- 但是这和咱们SwiftUI的MV模式有什么关系呢?

- 答：首先，MV模式是死的
- Model驱动View，实际是用户先修改Model的值，监听View的Model就变啦！
  就像一个空荡荡的口号，数据和界面，让数据驱动界面固然是非常美好的，可是这nm也太不现实了吧！我们的原数据（Outline）总不可能白花花地显示在界面上，它需要经过parser处理，也就是一堆函数的磨炼！
- 可是函数怎么调用，谁去调用，MV这个抽象的口号根本没有给出实际的解释

- 那Redux模式是活的吗?

- 下面是一个Reducer的实现～上面的两个case就是state的抽象（上框），而下面的就是根据Action和现state去调用函数，并返回未来的状态（下框）

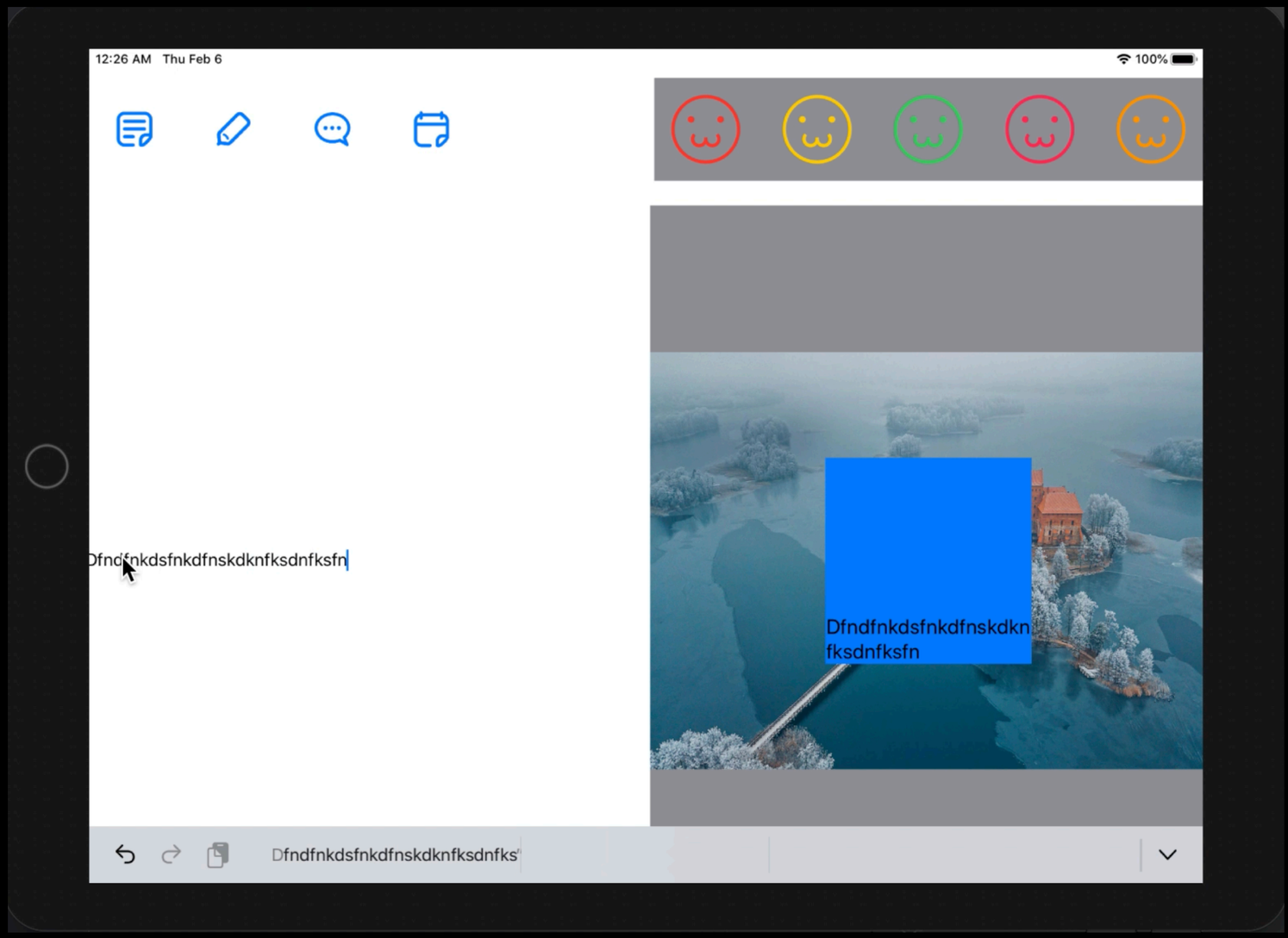
```
enum KeyboardState {
    //行号和处于行的状态
    case startline(linenumber : Int)
    case inline(linenumber : Int)

// apply是一个状态机函数， 它根据旧状态，用户的行动，确定新的状态
func apply(inparser : Parser, thisaction : KeyboardAction)->KeyboardState{
    switch self {
    case .startline( _):
        switch thisaction {
        case .editinline(let linenumber,let newstring):
            // 这里随便写了parser的调用的函数
            inparser.chageOutline(lineid: linenumber, pageid: 0, newstring: newstring)
            inparser.drawSingleSlide()
            return .inline(linenumber: linenumber)
        default:
            inparser.drawSingleSlide()
            return .startline(linenumber: 0)
        }
    default:
        switch thisaction {
        case .deleteline(linenumber: 0):
            inparser.drawSingleSlide()
            return .startline(linenumber: 0)
        default:
```

- 所以函数由Reducer调用，而调用哪一个函数由当前的状态决定

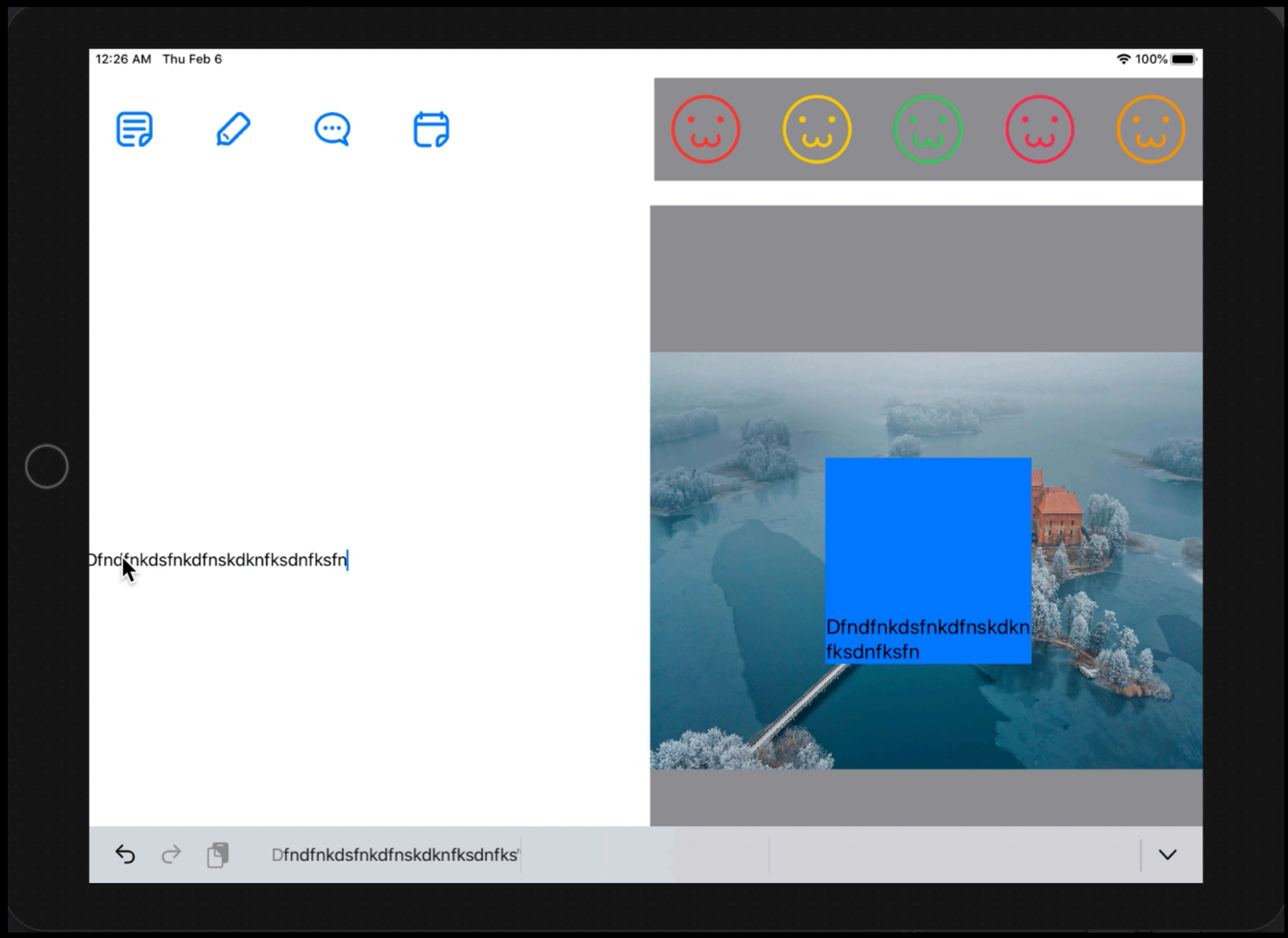
- 那么Redux要怎样在软件中实践呢?
- 理清楚view，action，state，reducer分别是什么，干哪些任务就可以了
- Let’ start

view



- 总结来说就是一个编辑一个显示
- 编辑有几种模式（键盘，pencil，脑图，nlp）
- 显示可能绘被改个模版改个颜色

view



- 最重要的关系是左边编辑好了右边显示要实时修改
- 其他关系是 1. 戳左上方按钮切换模式和编辑界面 2. 戳右上方按钮修改预览

state

- 看样子，我们的view的直接数据源就是outline和slide，实时预览对slide数据加个监听就好了

- 直接把slide作为我们界面的state并不是不可以，但是只有它不够，或者说它不是我们state中的重点

1. 只有slide作为界面刷新的数据不够：我们的界面还需要我们处在哪种编辑模式，在这种编辑模式下我们编辑到哪里的数据

2. 对slide做observe不是很方便，因为它是个很多层嵌套成的组件，有时候我们只是修改了组件里的一个子层（层层的observe我试了一下没成功）

state

```
struct AppState {
    enum viewstate{
            case keyboard // 处在键盘输入界面
            case nlp // 处在NLP输入界面
            case brainmap // 处在脑图输入界面
            case pencil // 处在Apple Pencil输入界面
            case file // 处于文档打开界面
    }
    // 是否显示键盘编辑
    var keyboardout = true
    // 是否显示nlp编辑
    var nlpout = false
    var brainmapout = false
    var pencilout = false
```

- 这是一个真实的State，记录着界面显示需要的真实数据；一个抽象的状态表示enum和一堆是否显示的数据

state

```
enum KeyboardState {

    // 行号和处于行的状态
    case startline(linenumber : Int, pageid : Int)
    case inline(linenumber : Int, pageid : Int)
```

- 这是另一个真实的State，键盘编辑界面；考虑光标可能在一行中编辑或者刚删除或者添加某个节点在行的开头；

state

总的来说，用enum做了一部分抽象（用于方便之后状态机的跳转）；同时用到了一部分界面显示需要的直接数据

state

那么再问一个问题，parser，尤其是parser中的slide存在哪里?

答：Parser是一个单例，在程序打开文件的时候初始化一次，在后续的使用中总是被不断引用；用SwiftUI中的环境变量机制，类似对于全局类的检测，使得我们在任何界面的时候都能得到它

action

- 同时，由view可以很容易引出action定义，即用户有哪些操作
- 第一大类：切换模式：脑图模式，键盘输入模式，语义识别模式，cv模式的切换（列出可能性）
- 第二大类：修改模版或者修改颜色
- 第三大类：修改大纲：在以上四个模式的任意一个模式中，我们可能存在有修改大纲的操作，以键盘输入模式为例：当我们在键盘输入时，会有点击输入，点击删除，新增某个节点，删除某个节点

action

可以看到三面可是有三种操作其实并非平行关系

1. 一、三类而是总分关系，进入某种模式后才能做第三类修改

2. 第二类是平行关系，反正和一、三都没啥联系，只要戳这个都会改预览

reducer

终于到我们可爱的reducer了！前面说了，reducer是用来处理用户的action，既然action有总分关系，reducer也应该职责更分明：

- 一个爸爸reducer，四个儿子reducer
- 这4个儿子是每次只有一个reducer在work

reducer

和action的关系：

- 第一类的action给爸爸处理，第三类的给各自的儿子处理
- 第二类，可怜的第二类，为了不重复写代码，把它交给爸爸了

reducer

其中：每个action除了负责刷新state（就是我们界面的数据源），还要负责调用parser中的相应的函数，如：

```
case .changecolor(let colorid):
    inparser.changeTemplateid(input: colorid)
    return AppState(keyboardout: self.keyboardout, nlpout: self.nlpout, brainmapout: self.brainmapout, pencilout:
        self.pencilout)
```

- 可见这货收到changecolor的action就调用了parser中修改template的函数数据自然刷新了

reducer

```
case .editinline(let linenumber,let newstring):
    // 这里随便写了parser的调用的函数
    inparser.chageOutline(lineid: linenumber, pageid: 0, newstring: newstring)
    inparser.drawSingleSlide()
    return .inline(linenumber: linenumber)
```

- 再比如，这货收到editline的action就调用了parser中changeoutline（修改大纲的函数），和drawSingleSlide（重绘页面函数）

最后：关于接口

- Parser是个神奇的单例，提供了很多函数接口；
- 我们不能直接修改parser里面outline和slide的值，只能通过函数修改；
- 除了reducer外其他地方都不能调用parser；也就是说view是通过给view发action来间接的操作其中的数据滴～