

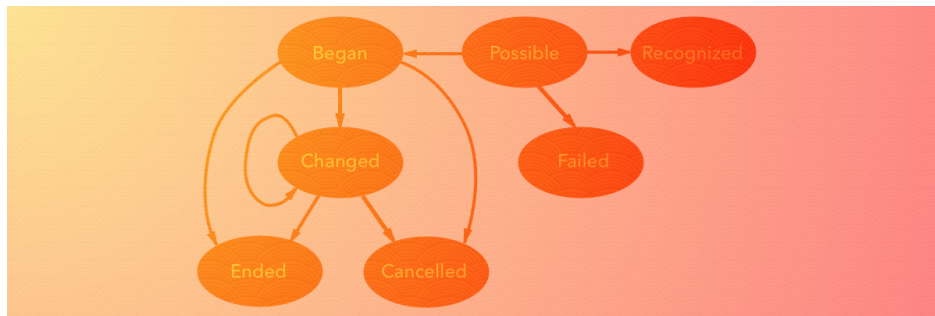
GameplayKit：非游戏类应用的状态机



作者 Purpleen (/u/7e3d4f5939cc)

2015.09.01 11:31 字数 1772

(/u/7e3d4f5939cc)



开发移动应用是一项非常复杂的工作，但作为开发者，我们就是来解决这个复杂的。状态机（state machine）是一个很好的工具，它可以帮助我们简化开发中的复杂问题。因此，在本篇基于Swift语言的Xcode教程中，我们将学习为iOS 9和OS X 11 El Capitan系统开发App时，如何使用状态机。

视图控制器可以很容易地定义为一个嵌入多个复杂功能的类。例如，假设一个须通过社交网络展示用户个人信息的视图控制器。我们遇到的第一个情况就是：视图控制器的视图出现时数据尚未加载到App中。在这种情况下，我们需要处理两个状态：数据可用和数据不可用。视图和相应的布局依赖于当前状态。处理这两个状态的最简单方法是定义一个boolean变量。如，若数据可用，设置boolean变量为true；若数据不可用，在设置boolean变量为.....嗯，稍等，这里不简单。因为“数据不可用”究竟是什么意思？可能是，当我们从远程服务器调用时，数据尚不可用；也可能是，我们未能从远程服务器获取到数据，导致数据不可用。基于这个“不可用”，我们需要根据数据的真实状态来定义UI面板。

因此，现在产生了三种状态，我们的应用程序需要能够从三种状态中来回切换。例如，当从服务器接收到数据时，UI面板能从显示动态指示器切换到隐藏动态指示器。但是如果应用程序未能接收到数据，我们需要一直隐藏动态指示器，并通知用户“没有数据”。

此时，初始视图控制器已经形成。现在我们不仅需要跟踪当前状态，还需要跟踪状态转换。

在iOS 9和OS X 11 El Capitan中，有一个新的API可以无缝地创建一个状态机。这个新API是GameplayKit的一部分，通常用于开发视频游戏。接下来，我们就来学习如何使用这个视频游戏API来管理应用程序状态。

创建状态

在这里会用到两个类：GKState和GKStateMachine。首先，需要为状态机创建状态。为此，要使用子类GKState。

回顾之前视图控制器的例子，我们需要一个状态。例如，定义一个接收服务器返回数据的状态，我们需要使用CGState子类：

```
import UIKit
```

```

import GameplayKit

class RetrievingDataState: GKState {

    unowned var profileViewController : ViewController?

    override func isValidNextState(stateClass: AnyClass) -> Bool {

        return (stateClass == DataAvailableState.self) || (stateClass ==
        DataNotAvailableState.self)

    }

    override func didEnterWithPreviousState(previousState: GKState?) {

        // Perform here the actions that you want to do with your UI when it enters this state
        like for example, showing and starting an activity indicator

        profileViewController?.activityIndicator.startAnimating()

    }

    override func willExitWithNextState(nextState: GKState) {

        // Perform here the actions that you want to do with your UI when it exits this state
        like for example, stopping the activity indicator.

        profileViewController?.activityIndicator.stopAnimating()

    }
}

```

状态机使用`isValidNextState`方法来定义是否切换到了新状态。返回`true`说明状态可切换。在本文例子中，可以从`RetrievingDataState`切换到`DataAvailableState`或者`DataNotAvailableState`，这主要取决于服务器调用的结果。

`didEnterWithPreviousState`和`willExitWithNextState`两个函数用来定义当进入或退出一个状态时，需要进行什么操作。本例中，当进入`RetrievingDataState`状态时，启动一个活动指示器来显示此应用程序正在处理一些功能；当退出该状态时，需要停止这个活动指示器。

后续如果我们接收到数据（如带有信息的标签），需要再次展示某些功能时，我们可以再进行这个操作。我们甚至可以基于即将接收或者退出的状态来定义所要执行的动作。因为我们将所有的逻辑存储在一个独立并明确定义的地方，这将便于我们很好地控制每个情况下要如何操作，并在一定程度上可以使视图控制器更加轻量。

注意：我已经添加了一个属性来保存指向视图控制器的引用，所以我们可以通过状态子类来执行其方法以及访问其属性值。在这种情况下，这个引用必须是`unowned`，因为这个视图控制器将拥有一个指向该状态机的强引用，并且该状态机也将有一个指向每一个状态的强引用。如果这个状态拥有一个指向视图控制器的强引用，它将引起一个循环保留。因此，为避免这个情况，这个引用必须是`unowned`的。

为了实现其他的状态，我们再创建一个`GKState`的子类：

```

41class DataAvailableState: GKState {

```

```
unowned var profileViewController : ViewController?

override func isValidNextState(stateClass: AnyClass) -> Bool {

    return stateClass == RetrievingDataState.self

}

override func didEnterWithPreviousState(previousState: GKState?) {

    if let viewController = profileViewController {

        // Perform here the actions that you want to do with your UI when it enters this state.

    }

}

override func willExitWithNextState(nextState: GKState) {

    if let viewController = profileViewController {

        // Perform here the actions that you want to do with your UI when it exits this state.

    }

}

}

class DataNotAvailableState: GKState {

    unowned var profileViewController : ViewController?

    override func isValidNextState(stateClass: AnyClass) -> Bool {

        return stateClass == RetrievingDataState.self

    }

    override func didEnterWithPreviousState(previousState: GKState?) {

        if let viewController = profileViewController {

            // Perform here the actions that you want to do with your UI when it enters this state.

        }

    }

    override func willExitWithNextState(nextState: GKState) {

        if let viewController = profileViewController {

            // Perform here the actions that you want to do with your UI when it exits this state.

        }

    }

}
```

```
}  
  
}  
  
}
```

创建状态机

一旦我们拥有一些状态，我们就能创建状态机。

首先为上文提到的每个状态分别创建一个实例，然后为每个实例分配一个指向视图控制器的引用：

```
let retrievingDataState = RetrievingDataState()  
  
retrievingDataState.profileViewController = self  
  
let dataAvailableState = DataAvailableState()  
  
dataAvailableState.profileViewController = self  
  
let dataNotAvailableState = DataNotAvailableState()  
  
dataNotAvailableState.profileViewController = self
```

最后，为状态机创建数组来传递状态：

```
let stateMachine = GKStateMachine(states: [retrievingDataState, dataAvailableState,  
dataNotAvailableState])
```

启动状态机进入其中一个状态。本例中，第一个状态是RetrievingDataState。

```
stateMachine.enterState(RetrievingDataState)
```

状态机进入这个状态，然后执行didEnterWithPreviousState:方法。此时，动态指示器将出现在这个屏幕上。当执行这个方法时，previousState的值为nil，状态机第一个进入的状态是因为RetrievingDataState。

当你接收到这个服务器的响应时，根据结果，你将改变状态：

```
// When you receive updated data from the server  
  
stateMachine.enterState(DataAvailableState)
```

退出RetrievingDataState时，程序将执行willExitWithNextState:方法，同时动态指示器将停止。之后，将执行DataAvailableState状态的didEnterWithPreviousState:方法，以此类推。

总结


在应用程序开发过程中，我们通常要处理多个状态。同时处理这么多的状态时，事情就会很复杂：网络连接是否可用？服务器是否可用？数据是否已更新？数据是否无效？应用程序是否添加背景？打开推送通知是否启动应用程序.....等等。

尝试同时处理所有这些状态是一项非常复杂的工作。但是 GKState和CGStateMachine可以帮助我们非常清晰地定义每个状态，并且处理起来非常简单。

享受编程吧！！

Vicente Vicens

本文首发于CocoaChina (<http://www.cocoachina.com/game/20150831/13026.html>), 译者Purpleen, 编译自《GameplayKit: State Machine for non-game Apps (<https://www.invasivecode.com/weblog/gameplaykit-state-machine/>)》, 转载请注明出处。

 日记本 (/nb/101717)

[举报文章](#) © 著作权归作者所有



(<http://cwb.assets.jianshu.io/notes/images/167452/w>