用结构体和元组构建更整洁的类

“

作者：Benedikt Terhechte，原文链接，原文日期：2019-02-24
译者：WAMaker；校对：numbbbbb，BigNerdCoding；定稿：Pancf
翻译原文地址

假设你正在开发一款社交网络应用，其中包含了一个带有关注按钮和点赞按钮的用户图片展示组件。同时，为了满足单一功能原则（single responsibility principle）和视图控制器的构成，**点赞**和**关注**的实现应该另有它处。社交网络不仅有高级账户，也有企业账户，因此 `InteractiveUserImageController` （命名从来不是我的强项） 要能满足一系列的配置选项。以下是这个类一个可能的实现（为作展示，示例代码保留了不少可改进的地方）：

```
final class InteractiveUserImageController: UIView {
    /// 是否需要展示高级布局
    var isPremium: Bool
    /// 账户类型
    var accountType: AccountType
    /// 点击视图是否高亮
    var isHighlighted: Bool
    /// 用户名
    var username: String..
    /// 用户头像
    var profileImage: UIImage
    /// 当前用户是否能点赞该用户
    var canLike: Bool
    /// 当前用户是否能关注该用户
    var canFollow: Bool
    /// 大赞按钮是否能使用
    var bigLikeButton: Bool
    /// 针对一些内容使用特殊的背景色
    var alternativeBackgroundColor: Bool

    init(...) {}
}
```

至此，我们就有了不少参数。随着应用体量的增长，会有更多的参数被加进类里。将这些参数通过职能进行划分和重构固然可行，但有时保持了单一功能后仍会有大量的参数存在。要如何才能更好的组织代码呢？

Swift 结构体结构

Swift 的 `struct` 类型在这种情况能发挥巨大的作用。依据参数的类型将它们装进**一次性**结构体：

```
final class InteractiveUserImageController: UIView {
    struct DisplayOptions {
        /// 大赞按钮是否能使用
        var bigLikeButton: Bool
        /// 针对一些内容使用特殊的背景色
        var alternativeBackgroundColor: Bool
        /// 是否需要展示高级布局
        var isPremium: Bool
    }
    struct UserOptions {
        /// 账户类型
        var accountType: AccountType
        /// 用户名
        var username: String
        /// 用户头像
        var profileImage: UIImage
    }
    struct State {
        /// 点击视图是否高亮
        var isHighlighted: Bool
        /// 当前用户是否能点赞该用户
        var canLike: Bool
        /// 当前用户是否能关注该用户
        var canFollow: Bool
    }

    var displayOptions = DisplayOptions(...)
    var userOptions = UserOptions(...)
    var state = State(...)

    init(...) {}
}
```

正如你所见，我们把这些状态放入了独立的 `struct` 类型中。不仅让类更整洁，也便于新上手的开发者找到相关联的选项。

已经是一个不错的改进了，但我们能做得更好！

我们面临的问题是查找一个参数需要额外的操作。

由于使用了一次性结构体类型，我们需要在某处定义它们（例如：`struct DisplayOptions`），也需要将它们实例化（例如：`let displayOptions = DisplayOptions(...)`）。大体上来说**没什么问题**，但在更大的类中，为确定 `displayOptions` 的类型仍旧需要一次额外的查询。然而，与 C 语言不同，像下面这样的匿名 `struct` 在 Swift 里并不存在：

```
let displayOptions = struct {
    /// 大赞按钮是否能使用
    var bigLikeButton: Bool
    /// 针对一些内容使用特殊的背景色
    var alternativeBackgroundColor: Bool
    /// 是否需要展示高级布局
    var isPremium: Bool
}
```

元组 – 匿名结构体在 Swift 中的实现

实际上，Swift 中还真有这么一个类型。它就是我们的老朋友，`tuple`。自己看吧：

```
var displayOptions: (
  bigLikeButton: Bool,
  alternativeBackgroundColor: Bool,
  isPremium: Bool
)
```

这里定义了一个新的类型 displayOptions，带有三个参数（bigLikeButton， alternativeBackgroundColor， isPremium），它能像前面的 struct 一样被访问：

```
user.displayOptions.alternativeBackgroundColor = true
```

更好的是，参数定义不需要做额外的初始化，一切都井然有序。

强制不可变性

最后， tuple 既可以是 可变的 也可以是 不可变的。正如你在第一行所看到的那样：我们定义的是 var displayOptions 而不是 var 或 let bigLikeButton。bigLikeButton 和 displayOptions 一样也是 var。这样做的好处在于强制把静态常量（例如行高，头部高度）放入一个不同的（let）组。

添加数据

当需要用一些值初始化参数时，你也能很好的利用这个特性，这是一个加分项：

```
var displayOptions = (
  bigLikeButton: true,
  alternativeBackgroundColor: false,
  isPremium: false,
  defaultUsername: "Anonymous"
)
```

与之前的代码类似，这里定义了一个元组的选项集，同时将它们正确进行了初始化。

简化

相比于使用结构体而言，使用了元组的选项集能更轻易的简化代码：

```
class UserFollowComponent {
    var displayOptions = (
        likeButton: (
            bigButton: true,
            alternativeBackgroundColor: true
            ),
        imageView: (
            highlightLineWidth: 2.0,
            defaultColor: "#33854"
        )
    )
}
```

我希望这篇文章会对你有帮助。我大量应用这些简单的模式来让代码更具结构化。即便是只对 2 - 3 个参数做这样的处理，也能从中获益。