# iOS 上基于 RxSwift 的动态表单填写

在前端表单填写无处不在，但在 iOS 上，这件事并没有那么轻松，主要是因为我们需要 `UITableView` 和 `UICollectionView` 展示一项一项的数据。

但 `UITableView` 的 `UICollectionView` 的 API 不像写一个 HTML ，需要什么就直接写什么，你可以从 HTML 上直接明白这大概是怎么样的一个表单。但在 iOS 上，我们会写一大堆的返回 Cell 的逻辑。在更新 UI 的时候，也不像 HTML 那么轻松，直接更新对应的 DOM 即可。但由于 Cell 的重用，我们不能很好地找到对应的 “DOM” 。而完整的表单还涉及到表单验证问题，这进一步增加了在 iOS 表单提交的难度。

本文我们将关注改进当前表单填写的体验和代码上的优化。此外我们还会关注表单的组合关系：

- 改进填写体验
- 复杂表单填写
- 表单验证
- 表单组合

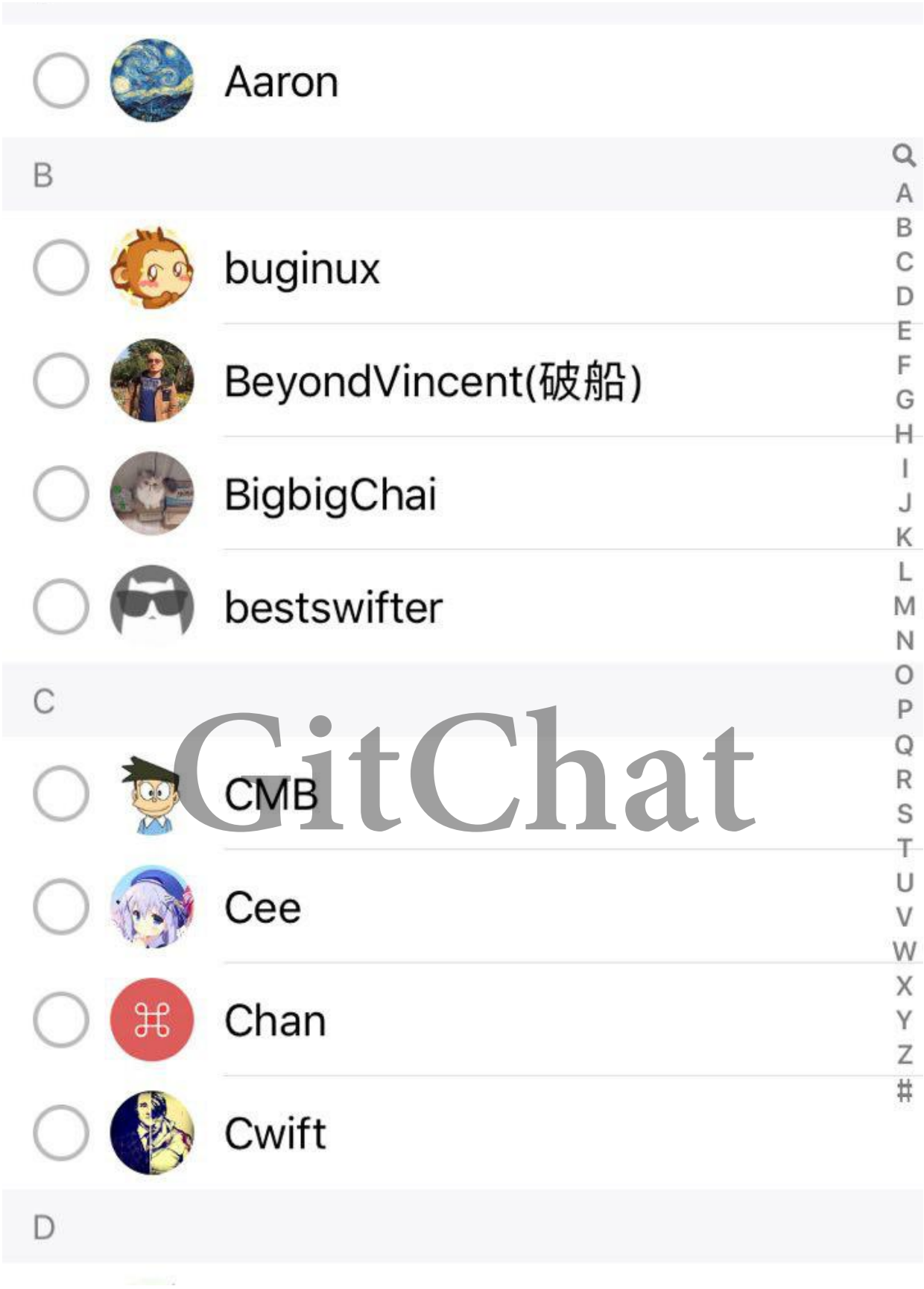


> 示例代码地址：https://github.com/DianQK/gitchat-form

## 改进填写体验

这是一个非常简单的表单：

选择多个用户，并提交。






体验一下 TIM （腾讯出品的办公版 QQ）这个选择用户的交互，你可以感受到点击到 Cell 的反馈不够流畅。这里值得思考一下：

> 我们是否需要刷新视图？
> 是。
> 我们是否可以通过 reload 刷新 Cell ？

> 是。
> 我们是否必须通过 reload 刷新 Cell ？
> 不是，我们可以直接拿到这个 Cell ，直接更新 Cell 上的 UI 。

那我们想办法完成这个效果。我们要做两件事情：

- 如果当前 Cell 在屏幕中，更新 Cell 为我们的预期样式。
- 更新对应的 Model.

为此我们只需要为 Model 添加绑定功能，Model 值的变化将直接更新 Cell 上的内容。

我们可以为每个对应于 Cell 的 Model 创建添加一个 `isSelected` 属性，代码如下所示：

```
struct Item {
    let name: String
    let isSelected: Variable<Bool>
}
```

接下来我们则只需要将 `isSelected` 状态绑定到 Cell 的视图变化中：



```
    Driver.just((1...9).map { Item(name: "\($0)", isSelected:
Variable<Bool>(false)) })
        .drive(tableView.rx.items(cellIdentifier: "Cell",
cellType: ReactiveTableViewCell.self)) { row, item, cell in
            cell.textLabel?.text = item.name
            item.isSelected.asDriver()
                .map { isSelected -> UITableViewCellAccessoryType
in
                    if isSelected {
                        return
UITableViewCellAccessoryType.checkmark
                    } else {
                        return UITableViewCellAccessoryType.none
                    }
                }
                .drive(cell.rx.accessoryType)
                .disposed(by: cell.reuseDisposeBag)
        }
        .disposed(by: disposeBag)
```

我们将每个 Model 的属性 `isSelected` 通过 `map` 方法绑定到 Cell 的 `accessoryType` 属性上。

通过 `reuseDisposeBag` ，我们无需担心 Cell 重用问题：

```
open class ReactiveTableViewCell: UITableViewCell {

    public private(set) var reuseDisposeBag = DisposeBag()

    open override func prepareForReuse() {
        super.prepareForReuse()
        reuseDisposeBag = DisposeBag()
    }

}
```

我们已经在准备重用时候，释放了之前的 `DisposeBag` 。

此时为了更新数据，我们只需要直接更新对应的 `isSelected` 即可，无需考虑手动更新 Cell 的状态。

比如我们可以在点击的时候切换对应的 `Item` 的选择状态：

```
tableView.rx.modelSelected(Item.self).asDriver()
    .map { $0.isSelected }
    .drive(onNext: { isSelected in
        isSelected.value = !isSelected.value
    })
    .disposed(by: disposeBag)
```



这样一来更新指定 Cell 上的状态我们只需要创建相应的 `Variable` 。

## 复杂表单填写改进

表单动态的地方主要体现在两个场景：

- 根据不同场景同一个页面展示需要填写的不同项目。
- 当前填写页展示一些选项框。

这些都是静态 `UITableView` 做不到的。

第一个场景很好理解，比方说，你不是微信超级会员，在发朋友圈的时候就没有勾选为私密的功能，这一勾选项也不在 UI 上展示，而当你充值了一定微币后，再发朋友圈时，则有了私密选项，我们就需要在该页面添加勾选为私密的 Cell.

填写表单某一项时，我们可能需要从一个列表选择一些数据、也可能在当前页面选择一些数据。常用的做法有四种：

- 可以直接修改（ `UITextField` 、 `UISwitch` ）。

- Push 一个新的页面，在这个页面选择完成后，将数据回调给上一个页面。
- 在当前页面弹出一个 Action Sheet 以选择相应的项目。
- 在 `UITableView / UICollectionView` 中插入一个选项 Cell，在这个 Cell 中选择相应的项目。

我们以这个 TIM 创建日程为例，在这个例子中将涉及多种填写内容的方式、刷新视图的方式，同时我完成了大部分的逻辑（除去选择参与人员，选择参与人员这一部分就需要读者来根据示例中处理各种逻辑的方式来完成了）。

```
let name: Variable<String>
let startTime: Variable<Date>
let endTime: Variable<Date>
let location: Variable<String>
let participants: Variable<[Member]>
let remind: Variable<Remind?>
let isBellEnabled: Variable<Bool>
let note: Variable<String>
```

这个 TIM 创建日程需要多个数据，姓名、开始时间、结束时间、地点、参与人员、提醒、响铃和备注。为此我创建了对应该表单需要的所有数据：



```
class ScheduleForm {
    let name: Variable<String>
    let startTime: Variable<Date>
    let endTime: Variable<Date>
    let location: Variable<String>
    let participants: Variable<[Member]>
    let remind: Variable<Remind?>
    let isBellEnabled: Variable<Bool>
    let note: Variable<String>
}
```

接下来我们需要做的就是将这个 `ScheduleForm` 拆分成对应到 Cell 上的填写项。

> 你需要先体验一下 TIM 的一些交互以理解下文当中处理的逻辑。

首先，我们需要确定哪些情况需要更新 `UITableView` ，哪些可以直接更新 `UITableViewCell` 。

参与人员需要更新 `UITableView` ，即 reload item。因为参与人员是一组数据，对应到视图上我们可能展示全部参与人员，Cell 的高度需要随之更新。

弹出 `UIDatePicker` 也需要更新 `UITableView` ，即 insert item。

展示提醒和备注呢填写项也需要更新 `UITableView` ，即 insert item。

备注也需要更新 `UITableView` ，多行的时候要调整 Cell 的高度。

其余比如更新日程主题、开始时间、结束时间等，我们可以选择直接更新 Cell ，代码如下所示

```
    Observable
        .combineLatest(scheduleForm.participants.asObservable(),
selectingTime.asObservable(), isNeedRemind.asObservable(),
scheduleForm.note.asObservable()) { (participants: [Member],
selectingTime: Variable<Date>?, isNeedRemind: Bool, note: String)
-> [NewScheduleSection] in
            // 处理各种 Section
            return [baseSection, participantsSection,
remindSection, noteSection]
        }
        .bind(to: tableView.rx.items(dataSource: dataSource))
        .disposed(by: disposeBag)
```

当参与人员、更新是否在选择时间、是否需要提醒、备注更新时，都将调用 `reloadData` 或者 reload item 等方法更新视图。

现在来看我们表单所有的 Item：



```
    enum NewScheduleItem: Equatable, IdentifiableType {
        case name(Variable<String>) // 日程名称
        case time(start: Variable<Date>, end: Variable<Date>) //
开始时间和结束时间
        case selectTime(Variable<Date>) // 时间选择
        case location(Variable<String>) // 位置
        case participants([Member]) // 参与人员
        case addRemind // 添加提醒和备注
        case remind(Variable<Remind?>) // 提醒时间
        case bell(isBellEnabled: Variable<Bool>) // 是否响铃
        case note(String) // 备注
    }
```

这里每一个 case 对应一个表单项目，但 Cell 可能是被重用的，比如 `name` 和 `location` 。

RxDataSources 通过 diff 两次数据比较出哪些 Cell 需要添加、删除、更新。正确实现协议 `IdentifiableType` 至关重要。

我们这里每一个 case 都只会在数据中出现一次，给每个 case 一个不同的 `identity` 即可。我们直接将 case 的名称转换成 `String` ：

```
    var identity: String {
        switch self {
        case .name:
            return "name"
        case .selectTime:
            return "selectTime"
        case .time:
            return "time"
        case .location:
            return "location"
        case .participants:
            return "participants"
        case .addRemind:
            return "addRemind"
        case .remind:
            return "remind"
        case .bell:
            return "bell"
        case .note:
            return "note"
        }
    }
```

判断相同的 item 是否需要更新需要实现 `Equatable`：



```
    static func ==(lhs: NewScheduleItem, rhs: NewScheduleItem) ->
Bool {
        switch (lhs, rhs) {
        case (.name, .name):
            return true
        case (.time, .time):
            return true
        case (let .selectTime(lTime), let .selectTime(rTime)):
            return lTime === rTime
        case (.location, .location):
            return true
        case (let .participants(lMembers), let
.participants(rMembers)):
            return lMembers == rMembers
        case (.addRemind, .addRemind):
            return true
        case (.bell, .bell):
            return true
        case (let .note(lNote), let .note(rNote)):
            return lNote == rNote
        default:
            return false
        }
    }
```

这里需要注意的是，case selectTime 我们对比的不是二者内容是否相同，而是是否为同一个对象。

我们先来完成一个相对简单的逻辑，点击**填写备注、提醒**，展示提醒和备注的填写项：

```
        tableView.rx.modelSelected(NewScheduleItem.self)
                    .flatMap { (newScheduleItem) -> Observable<Bool>
    in
                        switch newScheduleItem {
                        case .addRemind:
                            return Observable.just(true)
                        default:
                            return Observable.empty()
                        }
                    }
                    .bind(to: isNeedRemind)
                    .disposed(by: disposeBag)
```

当点击到 addRemind 时，将 true 的值设置给 isNeedRemind 。前面我们已经完成好 isNeedRemind 更新 UITableView 的逻辑，这里将 isNeedRemind 的 value 设置为 true 就完成了展开填写提醒和备注的逻辑。

## 组合表单



组合表单是指一个表单需要另一个表单的内容，我们需要将两个表单关键起来。

比如这个选择提醒的列表页，这其实也可以理解成一个表单，只是填写的内容非常少，只有一个选择提醒提前时间填写项。

这个选择提醒提前时间不仅仅是一个小的表单，还是一个为创建日程服务的表单。

这个小表单对应的逻辑均在 ScheduleRemindViewController 中，为了友好地使用 ScheduleRemindViewController ，我们需要更新两个参数：

```
var currentStartDate: Date = Date() // 当前开始时间
var remind: Variable<Remind?>! // 选择的提醒，nil 表示选择无
```

这个场景我们用 Segue 完成（等同于直接在观察者中实现 Push 一个新页面的逻辑），在传递的参数中设置开始时间，**并传递选择提醒这个实例**：

```
        override func prepare(for segue: UIStoryboardSegue, sender:
    Any?) {
        guard let identifier = segue.identifier else {
                super.prepare(for: segue, sender: sender)
                return
```

```
            }
            switch (segue.destination, identifier, sender) {
            case let (viewController as ScheduleRemindViewController,
    "changeRemind", (sender) as (selectedRemind: Variable<Remind?>,
    currentStartTime: Date)):
                viewController.currentStartDate =
    sender.currentStartTime
                viewController.remind = sender.selectedRemind
            default:
                break
            }
        }
```

接下来我们只需要在 ScheduleRemindViewController 中处理相关逻辑即可。

填写备注我们也可以选择一样的方式处理，此外我们还有一种处理方式，将 ScheduleNoteViewController 封装成 Observable，更新的文本内容通过 Observable 传递：

```
        tableView.rx.modelSelected(NewScheduleItem.self)
            .flatMap { (newScheduleItem) -> Observable<String> in
                switch newScheduleItem {
                case let .note(note):
                    return Observable.just(note)
                default:
                    return Observable.empty()
                }
            }
            .flatMap { [weak self] (defaultText) ->
    Observable<String> in
                return
    ScheduleNoteViewController.rx.createScheduleNote(defaultText:
    defaultText, previousViewController: self)
                    .flatMap { $0.rx.done }
                    .take(1)
            }
            .bind(to: scheduleForm.note)
            .disposed(by: disposeBag)
```

代码 ScheduleNoteViewController.rx.createScheduleNote(defaultText: defaultText, previousViewController: self).flatMap { $0.rx.done }.take(1) 成功地将修改备注逻辑放到 Observable 中进行传递，这比使用 Segue 要更清晰易懂一些。

我们可以直接从上几行代码理解一个完成功能的逻辑：

点击 Cell，如果点击的是点击修改备注，则进入修改备注页面，当修改完成时，获取修改的内容（只获取一次），将内容设置到 scheduleForm.note 中。

这样一来在两个 ViewController 中的传值问题就通过了共享 Variable 或装换成 Observable 完成了。

## 表单验证

为了保证用户正确地填写了表单内容，我们需要限制用户的输入，比如禁止点击某些按钮、点击后弹出错误提示。

在设置提醒时，我们不应该设置的提醒时间比当前时间还要早。

```
    .flatMap { [weak self] (selectedRemind, currentRemind) ->
Observable<Remind?> in
        guard let `self` = self else { return Observable.empty()
}
        return Observable<Remind?>.create({ (observer) ->
Disposable in

            if selectedRemind == currentRemind {
                observer.onCompleted()
            } else if let selectedRemind = selectedRemind,
selectedRemind.changedTime(for: self.currentStartDate) < Date() {
                observer.onError(CustomMessageError.message("请设
置提醒事件晚于当前时间"))
            } else {
                observer.onNext(selectedRemind)
                observer.onCompleted()
            }
            return Disposables.create()
        })
            .catchErrorShowMessageWithCompleted()
    }
```



上述代码在选择的时间提前于当前时间，弹出 请设置提醒事件晚于当前时间 的错误提示信息。

你可能会好奇为什么我这里是抛出一个错误，而不是直接弹出一个错误提示，并返回一个 Observable.empty() 。因为抛出一个错误可以更好地停止当前的事件流，此外我们也可以根据错误做更多事情，比如替换为更合适的提醒的提前时间等。

开始时间和结束时间选择稍微麻烦一些，在切换选择开始和结束时，共用的是同一个时间选择 Cell 。这里我们通过 reload item 更新了 DatePicker 对应的 Variable<Date> 。

## 总结

在 iOS 上处理表单比较复杂，我们需要处理好重用和更新视图的事情。在拥有了响应式的支持上，这让我们完成一个表单轻松了很多。本例中将每一个视图需要展示的内容均用 `Observable` 进行绑定，让我们无需再困扰视图更新的问题，专注数据的更新以及数据之间的关联的处理。

不要忘记结合本文中的内容尝试完成参与人员的填写。