

Todo III - 自定义Observable统一用户交互处理

RxSwift - step by step

← 返回视频列表

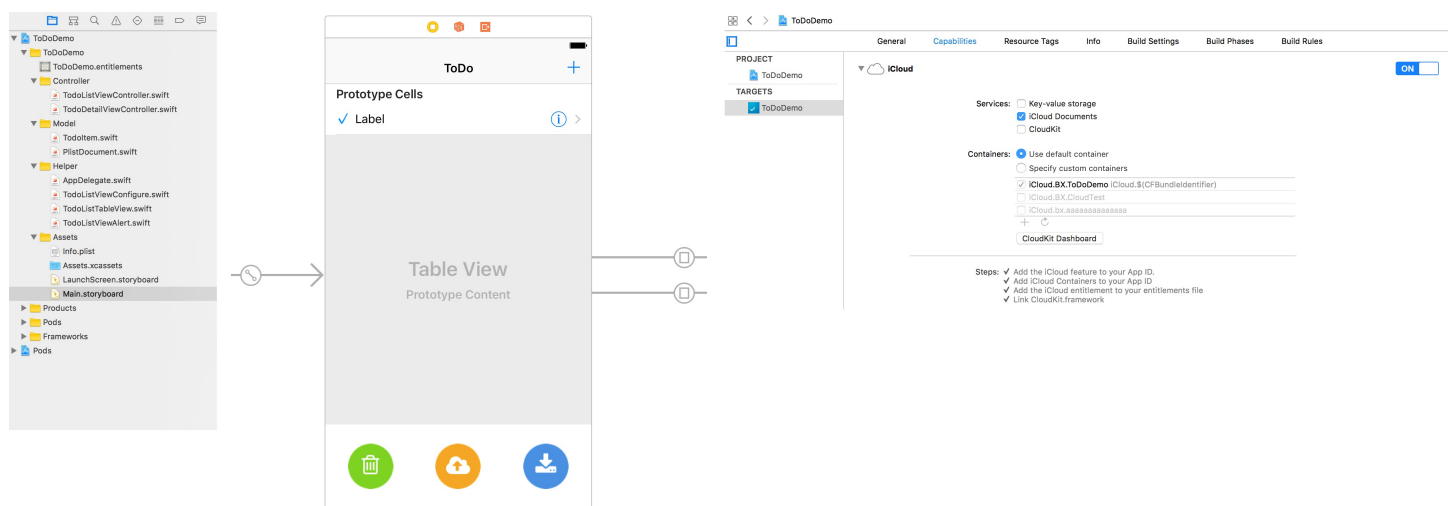
预计阅读时间: 36分钟

< PREVIOUS

NEXT >

处理用户交互的代码，有些是同步的，有些则是异步的。而把这些代码分别封装在Observable里，可以让我们用更一致的方式，处理交互的各种情况。

首先，[在这里下载](#)这一节的起始代码，相比之前完成的版本，我们添加了以下内容：



- 在 **TARGETS > Capabilities** 中，打开了iCloud功能，为了上传文件，要选中其中的iCloud Documents选项，接下来Xcode就会自动进行设置。在所有的Steps都设置完成后，Xcode会在项目中添加一个 *ToDoDemo.entitlements* 的文件；
- 为了通过iCloud读写文件，在Model目录中添加了一个 *PlistDocument.swift*，它实现了一个 *UIDocument* 的派生类 *PlistDocument*，定义了保存和读取文件时的必要方法；
- 在Helper目录添加了 *Flash.swift*，其中，只定义了一个方法 *Flash*，用于在操作完成后，给用户一个提示；
- 在 *Helper/ToDoListViewConfigure.swift* 中，添加了两个方法 *ubiquityURL(filename:)* 和 *syncTodoToCloud()* 用于把保存在本地的Todo同步到iCloud；
- 最后，在 *ToDoListViewController.swift* 中，设置了新添加按钮的 *@IBAction*；

对保存Todo的改造

了解了这些主要的改动后，我们打开 *TodoListViewConfigure.swift*，先来改造在本地保存Todo列表的功能。最初的实现是这样的

```
func saveTodoItems() {
    let data = NSMutableData()
    let archiver = NSKeyedArchiver(forWritingWith: data)

    archiver.encode(todoItems.value, forKey: "TodoItems")
    archiver.finishEncoding()

    data.write(to: dataFilePath(), atomically: true)
}
```

实际上，`data.write(to:atomically:)`方法是有可能执行失败的，失败的时候，它会返回`false`，而现在我们忽略了这个事实。为了在保存按钮的`@IBAction`中处理这个问题，我们让`saveTodoItems`返回一个`Observable<Void>`：

```
func saveTodoItems() -> Observable<Void> {
    // ...

    return Observable.create({ observer in
        let result = data.write(
            to: self.dataFilePath(), atomically: true)

        if !result {
            observer.onError(SaveTodoError.canNotSaveToLocalFile)
        }
        else {
            observer.onCompleted()
        }

        return Disposables.create()
    })
}
```

思考：为什么在`create`的closure参数中无需使用`[weak self]`呢？

实现的逻辑很简单，根据`write`的返回值，向订阅者发送`onError`或`onCompleted`事件就好了。完成之后，我们来修改保存按钮的`@IBAction`部分。第一个修改的版本是这样的：

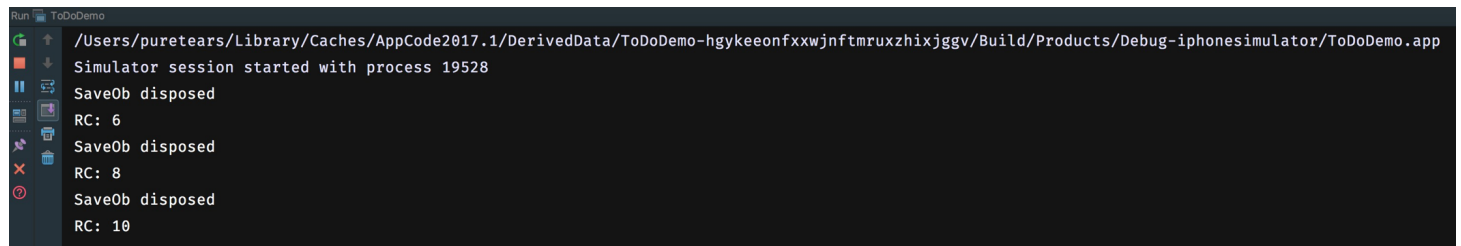
```
@IBAction func saveTodoList(_ sender: Any) {
    saveTodoItems().subscribe(
        onError: { [weak self] error in
            self?.flash(title: "Error",
                        message: error.localizedDescription)
        },
        onCompleted: { [weak self] in
            self?.flash(title: "Success",
                        message: "All Todos are saved on your phone.")
        },
        onDisposed: { print("SaveOb disposed") }
    ).addDisposableTo(bag)
}
```

其中，处理的逻辑很简单，我们只是根据订阅到的事件类型给用户弹了不同的Alert而已。但你能发现其中的问题么？其实，在之前的内容里我们提到过类似的场景：每当我们点击一次保存按钮，就往bag中添加了一个Disposable对象，但是由于TodoListViewController会一直存在，因此，bag中的对象一直也不会释放。于是，app占用的内存就会随着保存不断增长，为了观察这个效果，我们在saveTodoList最后，同样打印Resource计数：

```
@IBAction func saveTodoList(_ sender: Any) {
    // ...

    print("RC: \(RxSwift.Resources.total)")
}
```

重新执行一下，多次点击保存按钮，就能在控制台看到引用计数不断增长的结果了。



The screenshot shows the Xcode console output for a simulator session. The path is /Users/puretears/Library/Caches/AppCode2017.1/DerivedData/ToDoDemo-hgykeonfxxwjnftmruzhixjggv/Build/Products/Debug-iphonesimulator/ToDoDemo.app. The output shows the following sequence of messages:

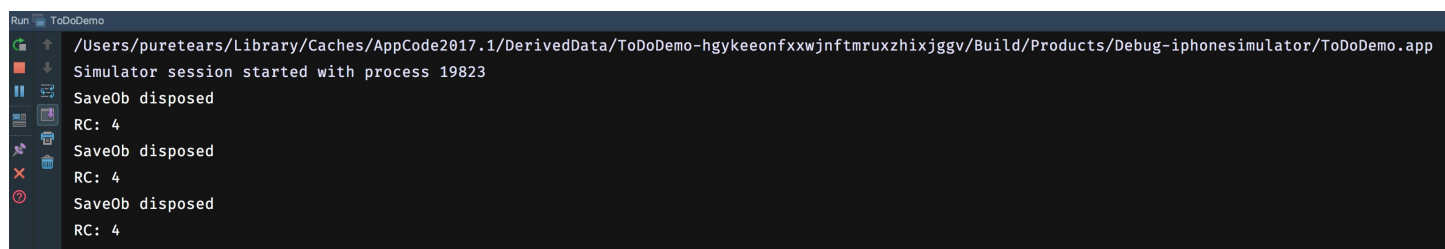
```
Simulator session started with process 19528
SaveOb disposed
RC: 6
SaveOb disposed
RC: 8
SaveOb disposed
RC: 10
```

但这显然不是我们期望的结果。因此，如果一个Controller会常驻在内存里不会释放，我们就不要把这种单次事件的订阅对象放到它的DisposeBag里。实际上，对于这种单次的事件序列，我们可以在订阅之后不做任何事情。因为订阅的Observable对象，一定会结束，要不就是正常的onCompleted，要不就是异常的onError，无论是哪种情况，在订阅到之后，Observable都会结束，订阅也随之会自动取消，分配给Observable的资源也就会被回收了。因此，直接把最后的addDisposableTo(bag)删除就好：

```
@IBAction func saveTodoList(_ sender: Any) {
    _ = saveTodoItems().subscribe(
        onError: { [weak self] error in
            self?.flash(title: "Success",
                        message: error.localizedDescription)
        },
        onCompleted: { [weak self] in
            self?.flash(title: "Success",
                        message: "All Todos are saved on your phone.")
        },
        onDisposed: { print("SaveOb disposed") }
    )

    print("RC: \(RxSwift.Resources.total)")
}
```

重新执行一次，现在，无论保存多少次，事件序列占用的资源都可以正常申请和回收了：

The screenshot shows the Xcode console for a project named 'ToDoDemo'. The path to the app is displayed at the top. The console output indicates that the simulator session started with process 19823. Subsequent log messages show 'SaveOb disposed' followed by 'RC: 4' multiple times, demonstrating that resources are properly released and managed after each save operation.

```
Run - ToDoDemo
/Users/puretears/Library/Caches/AppCode2017.1/DerivedData/ToDoDemo-hgykeonfxxwjnftmruzhixjggv/Build/Products/Debug-iphonesimulator/ToDoDemo.app
Simulator session started with process 19823
SaveOb disposed
RC: 4
SaveOb disposed
RC: 4
SaveOb disposed
RC: 4
```

以上，就是对保存Todo到本地功能的改造，这属于我们在一开始提到的第一种情况，处理用户交互的代码是同步执行的。可能你觉得这样的改动意义不大，甚至还有点儿更麻烦了。接下来，我们就来看另外一个场景：如何通过Observable封装异步执行的用户交互代码。

同步Todo到iCloud

把Todo同步到iCloud和核心功能是在*ToDoListViewConfigure.swift*中完成的。当然，当前我们的重点不是学习如何使用iCloud，而是在*syncTodoToCloud()*方法中下面的这段代码：

```
func syncTodoToCloud() {
    // ...

    plist.save(to: cloudUrl, for: .forOverwriting, completionHandler: {
        (success: Bool) -> Void in
        print(cloudUrl)

        if success {
            self.flash(title: "Success",
                        message: "All todos are synced to cloud.")
        } else {
            self.flash(title: "Failed",
                        message: "Sync todos to cloud failed")
        }
    })
}
```

可以看到，是否同步成功是通过调用`completionHandler`通知的，仿照之前的思路，我们可以让`syncTodoToCloud`返回一个`Observable<URL>`，其中的`URL`是iCloud保存在本地的路径：

```
func syncTodoToCloud() -> Observable<URL> {
    // ...

    return Observable.create({ observer in
        plist.save(to: cloudUrl, for: .forOverwriting,
                    completionHandler: { (success: Bool) -> Void in

            if success {
                observer.onNext(cloudUrl)
                observer.onCompleted()
            } else {
                observer.onError(SaveTodoError.cannotCreateFileOnCloud)
            }
        })

        return Disposables.create()
    })
}
```

在上面的代码里，在成功的时候，使用了`onNext`和`onCompleted`的组合，通知了同步成功事件，并结束了事件序列。当需要向`observer`返回内容的时候，就可以使用这样的形式。

要特别强调的是：`onCompleted`对于自定义`Observable`非常重要，通常我们要在`onNext`之后，自动跟一个`onCompleted`，以确保`Observable`资源可以正确回收。

完成后，我们来修改对应的订阅部分：

```
@IBAction func syncToCloud(_ sender: Any) {
    // Add sync code here
    _ = syncTodoToCloud().subscribe(
        onNext: {
            self.flash(title: "Success",
                message: "All todos are synced to: \($0)")
        },
        onError: {
            self.flash(title: "Failed",
                message: "Sync failed due to: \($0.localizedDescription)")
        },
        onDisposed: {
            print("SyncOb disposed")
        }
    )

    print("RC: \(RxSwift.Resources.total)")
}
```

逻辑上没什么好说的，跟订阅同步保存在本地的逻辑几乎相同。唯一不同的地方，在`subscribe`的closure参数里，我们没有使用`[weak self]`。实际上，这样完全没问题，因为`TodoListViewController`并不拥有`subscribe`返回的订阅对象。

What's next?

以上，就是自定义Observable在App开发中的实践。除了进一步体验`create` operator的用法之外，还有一个重要的内容是：当我们用`create`自定义一个单一事件序列的时候，并不用通过`DisposeBag`来自动回收Observable占用的资源。正确理解订阅对象和Controller之间的关系，是避免各种“诡异”bug非常重要的环节，所以，在继续之前，请务必确保已经理解了上面的内容。

接下来，我们还会陆续给Todo添加一些功能，以了解更多Rx operators是如何改变App开发流程的。

< [Prev: Todo II - 如何通过Subject传递数据](#)

☰ [Todo III - 自定义Observable统一用户交互处理](#)

[Next: 常用的忽略事件操作符](#) >

关于我们

想循序渐进的跟上最新的技术趋势？想不为了学点东西到处搜索？想找个伙伴一起啃原版技术经典书？技术之外，还想了解高效的工作流技巧？甚至，工作之余，想找点东西放松心情？没问题，我们用4K开发视频，配以详尽的技术文档，以及精心准备的广播节目，让你渴望成长的技术需求，也是一种享受。

Email Address

10@boxue.io

客户服务

📞 2085489246

相关链接

- › 版权声明
- › 用户隐私及服务条款
- › 京ICP备15057653号-1
- › 京公网安备 11010802020752号

关注我们

在任何你常用的社交平台上关注我们，并告诉我们你的任何想法和建议！



邮件列表

订阅泊学邮件列表以了解泊学视频更新以及最新活动，我们不会向任何第三方公开你的邮箱！

Email address

立即订阅

2019 © All Rights Reserved. Boxue is created by 10 ❤️ 11.

