Table of Contents

1.Map与**FlatMap**

2.Observable与**Alamofire**

3RxSwift Runtime分析**(**利用**OC**消息转发实现**IOS**消息拦截**)**

4.RxSwift中的**KVO**、**delegate**、**binding**的使用

5.RxSwift基础概念

6.RxSwift示例实战

7.RxSwift实现一个**UITableView**

Map与FlatMap

flatMap() 和 map() 有一个相同点：它也是把传入的参数转化之后返回另一个对象。但需要注意，和 map() 不同的是， flatMap() 中返回的是个 Observable 对象，并且这个 Observable 对象并不是被直接发送到了 Subscriber 的回调方法中。 flatMap() 的原理是这样的：

1. 使用传入的事件对象创建一个 Observable 对象；
2. 并不发送这个 Observable, 而是将它激活，于是它开始发送事件；
3. 每一个创建出来的 Observable 发送的事件，都被汇入同一个 Observable ，而这个 Observable 负责将这些事件统一交给 Subscriber 的回调方法。这三个步骤，把事件拆成了两级，通过一组新创建的 Observable 将初始的对象『铺平』之后通过统一路径分发了下去。而这个『铺平』就是 flatMap() 所谓的 flat。

Observable与Alamofire

Observable

Observable也就是一个被观察的对象，是一个事件序列，订阅者可以订阅它，监测事件的发生(Next\Complete\Error)。是不是听到这里就感觉这货有点像KVO？

热信号vs冷信号

信号分两种，热信号在它创建的时候就开始推送事件，这意味着如果后面有订阅者来的时候，就可能会错过一些事件。而冷信号则不会，只有在它被订阅的时候，它才会发送事件，这可以保证后面即使有订阅者中途加入的时候也能收到完整的事件序列。

获得Observable

要得到一个Observable有两种方式，一种是RxSwift已经提供了的（这里你可能需要引入RxCocoa），一种是自己创建。比如你有一个UITextfield，你要订阅其text的变化，就可以通过textfield.rx_text来获得这个Observable的对象。如果要自己创建的话，可以利用它提供的create函数来创建，create接受函数闭包作为参数，比如我的项目里用到了Alamofire，我如果想让它的response是一个observable的，我可以这样写：

```
    func fetchClubList(fromPage page:Int) -> Observable<[ClubTab
leCellViewModel]>{

        return create{ (observer ) -> Disposable in
            Alamofire.request(Method.GET, "http://www.mocky.io/v
2/560a046995e00cc208981280").responseArray{[unowned self]
                (results:[ClubModel]?, err:ErrorType?) -> Void
in
                    if let _ = err{
                        observer.on(Event.Error(err!))
                    }else{
                        if let cellVMs = (results?.map{ClubTable
CellViewModel(club: $0)}){
                            if page == 0{
                                self.cellViewModels.removeAll()
                            }
                            self.cellViewModels.appendContentsOf
(cellVMs)

                            observer.on(Event.Next(self.cellView
Models))
                            observer.on(Event.Completed)
                        }else{
                            //TODO: error handle
                        }
                    }
                }

            return NopDisposable.instance
        }
    }
```

这样就创建了一个Observable<[ClubModel]>，当他上面有事件发生的时候，我们可以做相应的操作。除了create，还可以通过never()创建一个什么都不发送的空序列，用empty()创建一个空序列但会发送.Complete等等。

信号的操作

常用的操作符有map，flatMap，subscribe，doOn，retry等等，具体的用法在它的文档中有具体说明。比如在我的demo里，需要根据页码加载列表数据。可以像这样做，在我的UIViewController里先创建一个属性：

```
var page : Variable<Int> = Variable(0)
```

接下来订阅这个page

```
    self.page
        .throttle(0.5, MainScheduler.sharedInstance)
        .doOn{ [unowned self] value in
            self.refreshCtrl.enabled = false
        }
        .flatMap {[unowned self] in
            self.viewModel.fetchClubList(fromPage: $0)
                .doOn{ [unowned self] in
                    self.refreshCtrl.endRefreshing()
                    self.refreshCtrl.enabled = true
                }
                .retry()
                .catchErrorJustReturn([])
        }
        .bindTo(self.clubList.rx_itemsWithCellIdentifier("clubCe
ll")){(_, viewModel, cell: ClubListCell) in
            cell.viewModel = viewModel
        }
        .addDisposableTo(self.dispose)
```

这段代码里实现了在page发生变化的时候自动去请求服务端的数据，然后刷新tableview。throttle保证了事件序列发送的频率不会过快，doOn在每次.Next到来之前让我们有机会做些额外的处理，页码变化通过flatMap触发API调用，然后把得到的数据绑定到每个cell上面。

通过RxSwift整个代码量减少了很多，也变得更加易读，是不是感觉很棒！现在了解的也不够多，欢迎大家一起来交流

```
func getSprites() -> Observable<AnyObject?> {
        return create{ observer in
            var request = Alamofire.request(.GET, "https://epoqu
ecore.com/sprites", parameters: nil)
                .response({ (request, response, data, error) in
                    if((error) != nil){
                        sendError(observer, error!);
                    }else{
                        sendNext(observer, data);
                        sendCompleted(observer);
                    }
                });
            return AnonymousDisposable {
                request.cancel()
            }
        }
    }
```

RxSwift Runtime分析(利用OC消息转发实现IOS消息拦截)

简要介绍：这是一篇介绍IOS消息拦截的文章，来源于对RxSwift源码的分析，其原理是利用Object-c的消息转发(forwardInvocation:)来实现(ReactiveCocoa中也是这个原理，而且是RXSwift借鉴的RAC和MAZeroingWeakRef)，阅读本文章需要对OC的runtime有一定的了解，并且对函数式响应编程(FRP)框架RAC或者RxSwift有一定的了解，不然会很迷惑的。

特别说明：如果想深入到代码层次来理解这个机制，可以参照着RxSwift框架中RxCocoa模块代码的_RXObjCRuntime.h和_RXObjCRuntime.m文件来理解，使用端的代码可以参考RxCocoa中的NSObject+Rx.swift文件的publicfuncrx_sentMessage(selector:Selector) ->Observable<[AnyObject]>方法（以及publicvarrx_deallocating:Observable<()>），并配合着文中提供的那张大图来仔细研读。

本文结构：文章总共包括两个部分：1）理论部分；2）源码剖析部分。这两个部分都会配置上一些图，帮助大家理解，最后会把用OmniGraffle绘制的图上传上来。

理论部分：

至于什么是函数式相应编程(FRP)以及有什么好处此处不做介绍，ReactiveCocoa和RxSwift(结合MVVM)的基本用法也不做介绍，下面只结合RXSwift源码对这俩框架中的其中一个技术做剖析，也就是本文的主题，如何利用Object-c的消息转发(forwardInvocation:)机制来实现IOS的消息拦截机制！

1. 首先介绍下最终实现的效果，就是可以为任何OC类(排除CF类和已经实现类似KVO机制的类)的方法添加一个切面(AOP)，这个切面可以获取到原方法的参数列表，加上自己的逻辑，这时就可以做到OC在执行某一个方法的时候，加上自己的逻辑，这一点很想Java中spring的AOP，不过没那么强大，在RXSwift中主要利用这个工作来给使用者提供一个钩子(hook)，并结合RXSwift中信号流的概念，在方法被调用时，以数据流的方式发送出来。

2. 在继续下面之前，有必要先简要说说apple实现KVO的原理(先推荐大家看一下Mike Ash的一篇KVO的文章，还有Mike Ash的一篇NSNotificationCenter的文章，也可以看看NSNotificationCenter 与 KVO 的实现比较的一篇文章)，我简

要描述下，KVO的实现，是借助了OC的动态性语言的特点，利用runtime在程序运行时动态的为ClassA添加一个子类，暂且叫_KVO_ClassA,把ClassA设置为其父类，重写要监听的属性pro的setPro方法，在重写的这个方法中，赋值前后发出相应的通知，并且把用户当前要监听的对象的isa指针设置为_KVO_ClassA，这样，当使用者调用a.pro=xxx的时候，就会顺着isa指针找到_KVO_ClassA类的被重写的setPro方法，这样就做到了KVO的特性(KVO中还重写了class函数，使得当使用者调用[a class]时返回的还是ClassA，而不是a的isa指针指向的_KVO_ClassA，这里有点欺骗了使用者)。

3. 这里的原理跟这个有一点相似的地方，就是也是通过给当前类ClassB(假设有有一个方法叫selector)添加一个子类*RX_namespace_ClassB*（*_RX_namespace*是命名前缀）,这个类继承ClassB，并且添加一个_RX_namespace_selector方法，并且把_RX_namespace_selector的实现设置为原selector的实现，然后再把原selector的实现设置为_objc_msgForward,_objc_msgForward是runtime的消息转发环节的入口，这样当使用者调用[a selector]是，就进入了OC runtime的消息转发环节；

4. 这里还有一个地方就是，会为每一个rx临时类(暂且把_RX_namespace_ClassB这一群类叫做rx临时类，把_RX_namespace_selector一类的方法叫做rx临时方法)新增或者替换forwardInvocation:的实现（其实还有另外三个，不过这个最重要，其他三个是respondsToSelector:, class, methodSignatureForSelector: ;其中class就是类似KVO的那种欺骗使用者的效果），在forwardInvocation:的新实现中，调用static BOOL RX_forward_invocation(id self, NSInvocation *invocation)方法；

```
// ##非常重要的方法##
// 这个方法的调用时机：对于通用方法，采用把中间类的这个方法的实现修改为_objc_msgForward，从而触发了消息转发环节；
//                  在前面的准备环节中，会调用swizzleForwardInvocation方法，来替换forwardInvocation:方法，
//                  替换的结果是调用RX_forward_invocation，如果返回NO(该类没有中间方法)，则调用父类方法或者调用原始方法
// 在这个方法中，先处理个性化的RXMessageSentObserver，然后在调用中间方法
static BOOL RX_forward_invocation(id __nonnull __unsafe_unretained self, NSInvocation *invocation) {
    SEL originalSelector = RX_selector(invocation.selector);

    id<RXMessageSentObserver> messageSentObserver = objc_getAssociatedObject(self, originalSelector);

    if (messageSentObserver != nil) {
        NSArray *arguments = RX_extract_arguments(invocation);
        [messageSentObserver messageSentWithParameters:arguments];
    }

    if ([self respondsToSelector:originalSelector]) {
        invocation.selector = originalSelector;
        [invocation invokeWithTarget:self];
        return YES;
    }

    return NO;
}
```

5. 在进入整个环节之前（也就是开启监听方法的入口），还有一个步骤，就是给原类实例对象添加一个关联属性，这个关联属性的key就是_RX_namespace_selector，属性值value是名为MessageSentObservable实例对象的钩子，这样在d步骤中RX_forward_invocation的实现中以key为_RX_namespace_selector获取这个钩子，获取到钩子之后，调用钩子的-(void)messageSentWithParameters:(NSArray*)parameters方法，把原方法的参数以数组的方式传递出去，最后在把invocation的方法SEL设置为_RX_namespace_selector，调用[invocation invokeWithTarget:self]，还记得d中的方法实现替换的步骤吗？那一步，把_RX_namespace_selector的实现设置为原方法的实现，这样，就实现了不破坏现场的特性了；至此通用情形下得原理就结束了。

6. 因为OC的消息转发环节，不是直接调用方法的实现，而是绕了一个圈子，所以效率上肯定是有折扣的，所以RXSwift中间又加了一层优化机制，下面简述下优化机制过程原理：

7. 再讲述优化机制前，有必要简要说下RXSwift中强大的宏的运用（这里宏的运用是为了生成函数或者category，从而减少代码的量），这里到处使用到了宏，大部分都是利用宏来动态生成代码，比如说生成函数，生成category等等；比如说d中提到的“新增或者替换forwardInvocation:”的方法-(BOOL)swizzleForwardInvocation:(Class **nonnull)class error:(NSError **nonnull)error就是下面这个宏生成的：

```
// 替换转发方法forwardInvocation:的实现，使得其调用RX_forward_invocation方法
SWIZZLE_INFRASTRUCTURE_METHOD(
    void,
    swizzleForwardInvocation,
    ,
    @selector(forwardInvocation:),
    FORWARD_BODY,
    NSInvocationRef
)
```

（这个宏生成的category会生成一个方法-(BOOL)swizzle_void_id:(Class **nonnull)class selector:(SEL)selector error:(NSError ** nonnull)error）

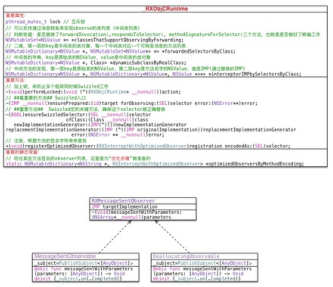
> 下面优化环节要用到的category也都是宏生成的，文章后面的图中，会给出例子；在每一个category中，都有load方法，load方法又都是在类被加载时就执行，举例子来说吧，宏SWIZZLE_OBSERVE_METHOD(void, id)针对的是为返回值为void，有一个参数为id类型的方法签名的一类生成个RXInterceptWithOptimizedObserver，在load方法中，将这个observer根据方法的签名为key添加到optimizedObserversByMethodEncoding中（这俩东西下面会讲解到）；

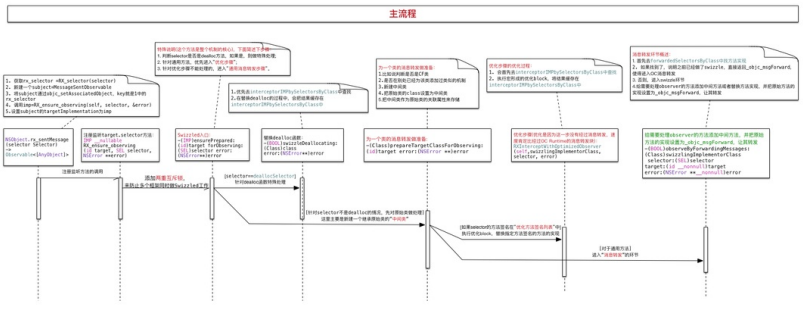
RxSwift中存在一个全局静态变量optimizedObserversByMethodEncoding，这个变量就是存储RXInterceptWithOptimizedObserver列表，而RXInterceptWithOptimizedObserver的定义是typedef BOOL (^RXInterceptWithOptimizedObserver)(RXObjCRuntime ***nonnull self, Class*** *nonnull class, SEL* ***nonnull selector, NSError ***** *nonnull error);*，当需要监听某一个方法的执行时，首先会根据这个方法的方法签名，到*optimizedObserversByMethodEncoding*中查找，找不到就进入*c,d,e*三个对应的通用消息转发环节中，找到了，就执行*g*中宏生成的*swizzle_void_id*方法，这个方法中是将原方法新增或者替换一个*block*生成的方法实现，这个*block*中，首先根据*_RX_namespace_selector*来找*RXMessageSentObserver*钩子对象，获取到钩子之后，调用钩子的*-(void)messageSentWithParameters:(NSArray*)parameters方法，再调用msgSend(&superInfo, selector, id_0)或者originalImplementationTyped(self, selector, id_0)来保持现场完整性，这样就把selector给拦截了，别切不破坏现场，而且也没有用到OC的消息转发，只有第一次使用时需要方法实现的替换，后续的效率肯定高。

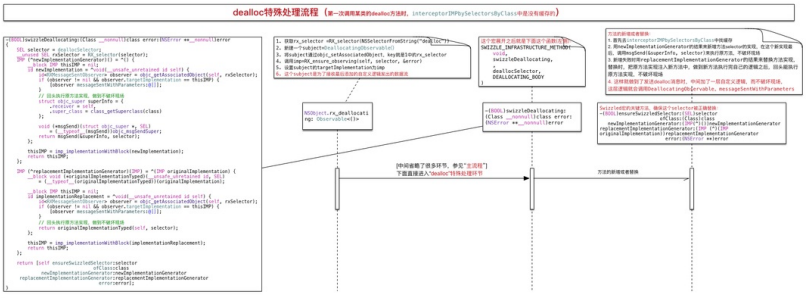
当需要监听一个对象的dealloc方法的调用时，RXSwift中还针对dealloc方法做了特殊化处理，这个过程跟“优化环节”很像，在此不再表述，文章后面的图中有说明。

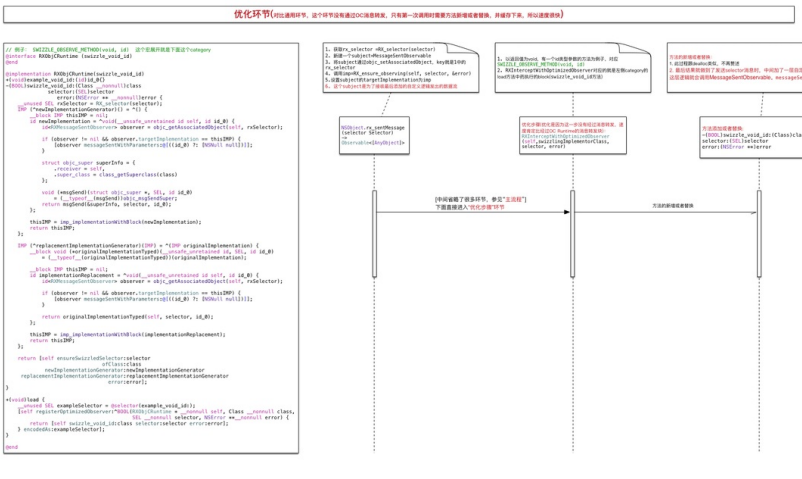
源码剖析部分

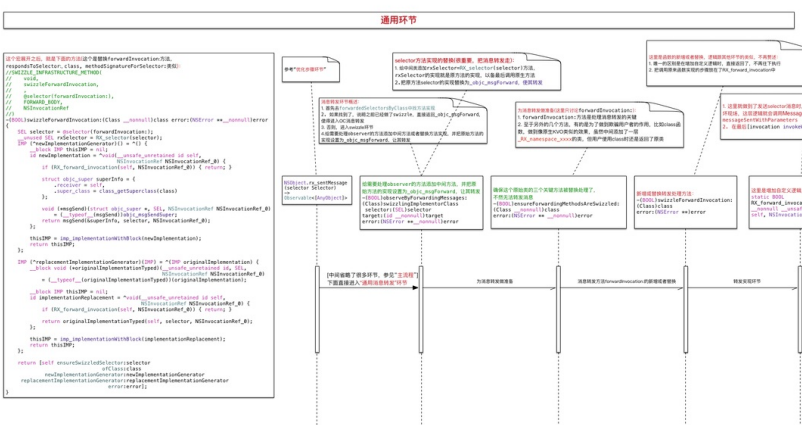
- 上面一大堆的原理过程，看着难免有些枯燥，下面给出一张大图（小弟文档工地太差，画图很不规范，大家就凑合着看吧，意思应该还是表达清楚了），这个图中有对源码中关键部分，比如说宏，关键方法等做解读(这个图片太大，下载下来时出现模糊的情况，我另外在github上存了一份，可以去那儿下载高清版，以及OMniGraffe格式的文件，地址，另外，手机简书APP打开的图片是高清的，保存下来放在电脑上，效果不错)











- 针对宏，有必要给出两个展开的过程例子，那就选“理论部分中提到的两个关键宏”，其他的宏都是类似展开(展开过程就不详细写了，因为写起来，排版很难看，展开过程也很繁琐，有兴趣的可以私聊)，下面只给出几个宏展开之后的结果：

替换转发方法forwardInvocation:的实现的宏：

```
// 替换转发方法forwardInvocation:的实现，使得其调用RX_forward_invocation方法
SWIZZLE_INFRASTRUCTURE_METHOD(
    void,
    swizzleForwardInvocation,
    ,
    @selector(forwardInvocation:),
    FORWARD_BODY,
    NSInvocationRef
)

-(BOOL)swizzleForwardInvocation:(Class __nonnull)class error:(NSError **__nonnull)error
{
    SEL selector = @selector(forwardInvocation:);
    __unused SEL rxSelector = RX_selector(selector);
    IMP (^newImplementationGenerator)() = ^() {
        __block IMP thisIMP = nil;
        id newImplementation = ^void(__unsafe_unretained id self, NSInvocationRef NSInvocationRef_0) {
            if (RX_forward_invocation(self, NSInvocationRef_0)) { return; }

            struct objc_super superInfo = {
                .receiver = self,
                .super_class = class_getSuperclass(class)
            };

            void (*msgSend)(struct objc_super *, SEL, NSInvocationRef NSInvocationRef_0)
                = (__typeof__(msgSend))objc_msgSendSuper;
            return msgSend(&superInfo, selector, NSInvocationRef_0);
        };

        thisIMP = imp_implementationWithBlock(newImplementation);
        return thisIMP;
    };

    IMP (^replacementImplementationGenerator)(IMP) = ^(IMP originalImplementation) {
        __block void (*originalImplementationTyped)(__unsafe_unretained id, SEL, NSInvocationRef NSInvocationRef_0)
            = (__typeof__(originalImplementationTyped))(originalImplementation);

        __block IMP thisIMP = nil;
        id implementationReplacement = ^void(__unsafe_unretained id self, NSInvocationRef NSInvocationRef_0) {
            if (RX_forward_invocation(self, NSInvocationRef_0)) { return; }

            return originalImplementationTyped(self, selector, NSInvocationRef_0);
        };

        thisIMP = imp_implementationWithBlock(implementationReplacement);
        return thisIMP;
    };

    return [self ensureSwizzledSelector:selector
                                ofClass:class
              newImplementationGenerator:newImplementationGenerator
     replacementImplementationGenerator:replacementImplementationGenerator
                                  error:error];
}
```

大家可以看出下面编译报错了，那是因为宏生成的方法和我展开之后的方法名字一样

SWIZZLE_OBSERVE_METHOD(void,id)宏（两次截屏）：

```objc
SWIZZLE_OBSERVE_METHOD(void, id)

@interface RXObjCRuntime (swizzle_void_id)
@end

@implementation RXObjCRuntime(swizzle_void_id)
+(void)example_void_id:(id)id_0{}
-(BOOL)swizzle_void_id:(Class __nonnull)class
              selector:(SEL)selector
                 error:(NSError ** __nonnull)error {
    __unused SEL rxSelector = RX_selector(selector);
    IMP (^newImplementationGenerator)() = ^() {
        __block IMP thisIMP = nil;
        id newImplementation = ^void(__unsafe_unretained id self, id id_0) {
            id<RXMessageSentObserver> observer = objc_getAssociatedObject(self, rxSelector);

            if (observer != nil && observer.targetImplementation == thisIMP) {
                [observer messageSentWithParameters:@[((id_0) ?: [NSNull null])]];
            }

            struct objc_super superInfo = {
                .receiver = self,
                .super_class = class_getSuperclass(class)
            };

            void (*msgSend)(struct objc_super *, SEL, id id_0)
                = (__typeof__(msgSend))objc_msgSendSuper;
            return msgSend(&superInfo, selector, id_0);
        };

        thisIMP = imp_implementationWithBlock(newImplementation);
        return thisIMP;
    };

    IMP (^replacementImplementationGenerator)(IMP) = ^(IMP originalImplementation) {
        __block void (*originalImplementationTyped)(__unsafe_unretained id, SEL, id id_0)
            = (__typeof__(originalImplementationTyped))(originalImplementation);

        __block IMP thisIMP = nil;
        id implementationReplacement = ^void(__unsafe_unretained id self, id id_0) {
            id<RXMessageSentObserver> observer = objc_getAssociatedObject(self, rxSelector);

            if (observer != nil && observer.targetImplementation == thisIMP) {
                [observer messageSentWithParameters:@[((id_0) ?: [NSNull null])]];
            }

            return originalImplementationTyped(self, selector, id_0);
        };

        thisIMP = imp_implementationWithBlock(implementationReplacement);
        return thisIMP;
    };

    return [self ensureSwizzledSelector:selector
                                ofClass:class
              newImplementationGenerator:newImplementationGenerator
     replacementImplementationGenerator:replacementImplementationGenerator
                                  error:error];
}
```

编译错误，是因为宏生成的category跟我展开的名字是一样的

备注：这里说的RXSwift其实严格来说是RXSwift中的RxCocoa子框架；

关于RxSwift中把Delegate代理、KVO，Notification转化为流的原理后续文章会给出，这些相对于本文章中原理稍简单些。

RxSwift中的KVO、delegate、binding的使用

State 状态

允许修改的语言很容易访问全局状态并修改它，未受控制的状态的修改很可能会导致程序的混乱崩溃(combinatorial explosion),但是从另一方面来说，强类型语言能都写出更高效的代码，这中方式就是尽可能地保持状态简单，并使用单项数据流来模型化数据，这就是Rx的闪光处

Bindings 绑定

当你写UI app的时候，理想情况就是用户界面可以随着状态的改变而改变，并且不会出现不一致的情况，这就是所谓的binding。

```
observable.combineLatest(firstName.rx_text,lastName.rx_text) { $
0 + " " + $1 }
            .map { "Greeting \\($0)" }
            .bindTo(greetingLabel.rx_text)
```

> 官方建议使用.adddisposableTo(disposeBag) 即使对于简单地bindings 来说这并不是必要的

Retries

我们很期望APIs 不会失败，但是这并不能如我们所愿，下面我们就来看一个：

```
func doSomethingIncredible(forWho: String) throws -> IncredibleT
hing
```

这个函数如果执行失败很难再次retry，即使是retry了，这也会产生很多的transient states，这并不是我们想要的，Rx 实现起来就很简单

```
doSomethingIncredible("me")
          .retry(3)
```

Transient State

在写 async 程序的时候会有许多的 瞬态问题，最典型的就是输入框的自动搜索，先输入ab ,发送一次请求，再输入c,又会发送一次请求，之前的请求可能需要cancel掉，或者使用另外一个变量引用。 另外一个问题就是，如果请求失败，就需要大量的retry 逻辑处理,如果成功，之前的retry 也需要清理。 如果我们在触发请求之前能有一丢丢时间的间隔，那就会非常完美，毕竟有些输入操作需要较长的时间。 还有一个问题就是在搜索请求执行过程中，屏幕上显示啥呢？失败了，又要显示啥？(公司产品要求，你懂得，处理起来极其繁琐)，下面就是Rx 大展拳脚的时候了

```
searchTextField.rx_text
  .throttle(0.3, scheduler: MainScheduler.instance)
  .distinctUntilChanged()
  .flatMapLatest { query in
      API.getSearchResults(query)
          .retry(3)
          .startWith([]) // clears results on new search term
          .catchErrorJustReturn([])
  }
  .subscribeNext { results in
    // bind to ui
  }
```

throttle 节流，在设定的时间内多次发送请求，仅仅触发最后一次

distinctUntilChanged 求异去同，比较前后两个值，如果相同则不会触发，不同则触发

startWith 开始设定的值，可以认为是占位符或者placeHolder

整合网络请求

你想一同发送两个请求，当两个请求都成功后，将两者的结果整合起来处理，这... 好伤脑筋啊！！！ 没关系，zip 帮你实现

```
let userRequest: Observable<User> = API.getUser("me")
let friendsRequest: Observable<Friends> = API.getFriends("me")

  Observable.zip(userRequest, friendsRequest) { user, friends in
      return (user, friends)
    }
    .subscribeNext { user, friends in
        // bind them to user interface
    }
```

zip 将两个信号合并成一个信号，并压缩成一个元组返回，前提是两个信号均成功
还有个问题是，这些请求是在后台，绑定还没有在主线程发生，这就要用到 observeOn

```
let userRequest: Observable<User> = API.getUser("me")
  let friendsRequest: Observable<[Friend]> = API.getFriends("me"
)

  Observable.zip(userRequest, friendsRequest) { user, friends in
      return (user, friends)
    }
    .observeOn(MainScheduler.instance)
    .subscribeNext { user, friends in
        // bind them to user interface
    }
```

轻松整合RX

实现自己的observable,那眞是太简单了，(so easy，老板再也不担心我写不出代码了)

```
extension NSURLSession {
    public func rx_response(request: NSURLRequest) -> Observable
<(NSData, NSURLResponse)> {
        return Observable.create { observer in
            let task = self.dataTaskWithRequest(request) { (data
, response, error) in
                guard let response = response, data = data else
{
                    observer.on(.Error(error ?? RxCocoaURLError.
Unknown))
                    return
                }

                guard let httpResponse = response as? NSHTTPURLR
esponse else {
                    observer.on(.Error(RxCocoaURLError.NonHTTPRe
sponse(response: response)))
                    return
                }

                observer.on(.Next(data, httpResponse))
                observer.on(.Completed)
            }

            task.resume()

            return AnonymousDisposable {
                task.cancel()
            }
        }
    }
}
```

综合处理 (Compositional disposal)

设想一下几个场景： 在tableView上展示一个模糊的image，这个image首先需要获取，然后解码，然后模糊处理

1. 在cell 退出了可显示区域，整个操作可以取消

2. 当用户快速滑动cell,cell 进入可现实区域，不会立刻去获取image。cell仅仅是昙花一现,这需要发送很多的request 和 cancel 操作。
3. 我们可以限制并发数量

以上场景如果可以优化并满足要求，改多好，是吧！让我们看看 Rx是怎么做的

```
// this is conceptual solution
let imageSubscription = imageURLs
    .throttle(0.2, scheduler: MainScheduler.instance)
    .flatMapLatest { imageURL in
        API.fetchImage(imageURL)
    }
    .observeOn(operationScheduler)
    .map { imageData in
        return decodeAndBlurImage(imageData)
    }
    .observeOn(MainScheduler.instance)
    .subscribeNext { blurredImage in
        imageView.image = blurredImage
    }
    .addDisposableTo(reuseDisposeBag)
```

代理(Delegates)

代理一般用于回调和作为一种观察机制。 传统的delegate 在设置setter 方法时，不会触发初始值，因此你需要通过其他的途径来读取初始值。 RxCocoa 不仅提供了UIKit Class的封装，而且还提供了一套通用机制-DelegateProxy,使你能够封装你自己的 delegate 并作为可观察的Sequence 暴漏出来。 来看一下整合后的UISearchbar

> It uses delegate as a notification mechanism to create an Observable that immediately returns current search text upon subscription, and then emits changed search values.

```
extension UISearchBar {

    public var rx_delegate: DelegateProxy {
        return proxyForObject(RxSearchBarDelegateProxy.self, sel
f)
    }

    public var rx_text: Observable<String> {
        return defer { [weak self] in
            let text = self?.text ?? ""

            return self?.rx_delegate.observe("searchBar:textDidC
hange:") ?? empty()
                    .map { a in // a 包含了searchbar:textDidChang
e:的参数，第一个是Searchbar,第二个是值
                        return a[1] as? String ?? ""
                    }
                    .startWith(text)
        }
    }
}
```

RxSearchBarDelegateProxy 可在这里找到 here

下面是该API的使用

```
searchBar.rx_text
    .subscribeNext { searchText in
        print("Current search text '\\(searchText)'")
    }
```

通知 (Notificatioins)

通知可以注册过个观察者，但是他们也是未知类型的，值需要从userInfo中提取。冗余的形式：

```
let initialText = object.text

doSomething(initialText)

// ....

func controlTextDidChange(notification: NSNotification) {
    doSomething(object.text)
}
```

你可以使用rx_notification 来创建一个观察序列，减少逻辑和重复代码的散播。

```
 let subscription = notificationCenter.rx_notification("testNoti
fication", object: targetObject)
            .subscribeNext { n in
                numberOfNotifications += 1
        }
```

KVO

KVO 是一个很方便的观察机制，但是也不是没有缺点，他最大的缺点就是让人模糊的内存管理。 在观察一个对象的一个属性时，该对象必须要比注册的KVO observer 活的时间长，都则会遇到crash

```
`TickTock` was deallocated while key value observers were still
registered with it. Observation info was leaked, and may even be
come mistakenly attached to some other object.
```

另外还有一套规则你还得遵守，否则结果有可能很诡异。还的实现一个笨拙的方法

```
-(void)observeValueForKeyPath:(NSString *)keyPath
                     ofObject:(id)object
                       change:(NSDictionary *)change
                      context:(void *)context
```

RxCocoa提供了一套很方便的观察序列--rx_observe 和 rx_observeWeakly 使用方法：

```
view.rx_observe(CGRect.self, "frame")
    .subscribeNext { (frame: CGRect?) in
        print("Got new frame \\(frame)")
    }
```

or

```
someSuspiciousViewController.rx_observeWeakly(Bool.self, "behavi
ngOk")
    .subscribeNext { (behavingOk: Bool?) in
        print("Cats can purr? \\(behavingOk)")
    }
```

RxSwift基础概念

SupportCode

在进入正题之前，先看下项目里的 SupportCode.swift ，主要为 playground 提供了两个便利函数。

一个是 example 函数，专门用来写示例代码的，统一输出 log 便于标记浏览，同时还能保持变量不污染全局：

```
public func example(description: String, action: () -> ()) {
    print("\n--- \(description) example ---")
    action()
}
```

另一个是 delay 函数，通过 dispatch_after 用来演示延时的：

```
public func delay(delay:Double, closure:()->()) {
    dispatch_after(
        dispatch_time(
            DISPATCH_TIME_NOW,
            Int64(delay * Double(NSEC_PER_SEC))
        ),
        dispatch_get_main_queue(), closure)
}
```

Introduction

主要介绍了 Rx 的基础： Observable 。 Observable 是观察者模式中被观察的对象，相当于一个事件序列 (GeneratorType) ，会向订阅者发送新产生的事件信息。事件信息分为三种：

- .Next(value) 表示新的事件数据。
- .Completed 表示事件序列的完结。
- .Error 同样表示完结，但是代表异常导致的完结。（打个岔：协议命名，想起

来上午汤哥在微博说的一段话：

另外，我觉得 protocol 名字用形容词会更加语义分明，比如 Swift : Flyable, Killable, Visible。全用名词的话显得比较生硬，比如 Swift : Head, Wings, Ass。

empty

empty 是一个空的序列，它只发送 .Completed 消息。

```
example("empty") {
    let emptySequence: Observable<Int> = empty()
    let subscription = emptySequence
        .subscribe { event in
            print(event)
        }
}
--- empty example ---
Completed
```

never

never 是没有任何元素、也不会发送任何事件的空序列。

```
example("never") {
    let neverSequence: Observable<String> = never()
    let subscription = neverSequence
        .subscribe { _ in
            print("This block is never called.")
        }
}
--- never example ---
```

just

just 是只包含一个元素的序列，它会先发送 .Next(value) ，然后发送 .Completed 。

```
example("just") {
    let singleElementSequence = just(32)
    let subscription = singleElementSequence
        .subscribe { event in
            print(event)
        }
}
--- just example ---
Next(32)
Completed
```

sequenceOf

sequenceOf 可以把一系列元素转换成事件序列。

```
example("sequenceOf") {
    let sequenceOfElements/* : Observable<Int> */ = sequenceOf(0
, 1, 2, 3)
    let subscription = sequenceOfElements
        .subscribe { event in
            print(event)
        }
}
--- sequenceOf example ---
Next(0)
Next(1)
Next(2)
Next(3)
Completed
```

from

from 是通过 asObservable() 方法把 Swift 中的序列 (SequenceType) 转换成事件序列。

```
example("from") {
    let sequenceFromArray = [1, 2, 3, 4, 5].asObservable()
    let subscription = sequenceFromArray
        .subscribe { event in
            print(event)
        }
}
--- from example ---
Next(1)
Next(2)
Next(3)
Next(4)
Next(5)
Completed
```

create

create 可以通过闭包创建序列，通过 .on(e: Event) 添加事件。

```
example("create") {
    let myJust = { (singleElement: Int) -> Observable<Int> in
        return create { observer in
            observer.on(.Next(singleElement))
            observer.on(.Completed)
            return NopDisposable.instance
        }
    }
    let subscription = myJust(5)
        .subscribe { event in
            print(event)
        }
}
--- create example ---
Next(5)
Completed
```

failWith

failWith 创建一个没有元素的序列，只会发送失败 (.Error) 事件。

```
example("failWith") {
    let error = NSError(domain: "Test", code: -1, userInfo: nil)
    let erroredSequence: Observable<Int> = failWith(error)
    let subscription = erroredSequence
        .subscribe { event in
            print(event)
        }
}
--- failWith example ---
Error(Error Domain=Test Code=-1 "The operation couldn’t be compl
eted. (Test error -1.)")
```

deferred

deferred 会等到有订阅者的时候再通过工厂方法创建 Observable 对象，每个订阅者订阅的对象都是内容相同而完全独立的序列。

```
example("deferred") {
    let deferredSequence: Observable<Int> = deferred {
        print("creating")
        return create { observer in
            print("emmiting")
            observer.on(.Next(0))
            observer.on(.Next(1))
            observer.on(.Next(2))
            return NopDisposable.instance
        }
    }
    print("go")
    deferredSequence
        .subscribe { event in
            print(event)
    }
    deferredSequence
        .subscribe { event in
            print(event)
        }
}
--- deferred example ---
go
creating
emmiting
Next(0)
Next(1)
Next(2)
creating
emmiting
Next(0)
Next(1)
Next(2)
```

为什么需要 defferd 这样一个奇怪的家伙呢？其实这相当于是一种延时加载，因为在添加监听的时候数据未必加载完毕，例如下面这个例子：

```
example("TestDeferred") {
    var value: String? = nil
    var subscription: Observable<String?> = just(value)
    // got value
    value = "Hello!"
    subscription.subscribe { event in
        print(event)
    }
}
--- TestDeferred example ---
Next(nil)
Completed
```

如果使用 deffered 则可以正常显示想要的数据：

```
example("TestDeferred") {
    var value: String? = nil
    var subscription: Observable<String?> = deferred {
        return just(value)
    }
    // got value
    value = "Hello!"
    subscription.subscribe { event in
        print(event)
    }
}
--- TestDeferred example ---
Next(Optional("Hello!"))
Completed
```

Subjects

接下来是关于 Subject 的内容。 Subject 可以看做是一种代理和桥梁。它既是订阅者又是订阅源，这意味着它既可以订阅其他 Observable 对象，同时又可以对它的订阅者们发送事件。

如果把 Observable 理解成不断输出事件的水管，那 Subject 就是套在上面的水龙头。它既怼着一根不断出水的水管，同时也向外面输送着新鲜水源。如果你直接用水杯接着水管的水，那可能导出来什么王水胶水完全把持不住；如果你在水龙头下面接着水，那你可以随心所欲的调成你想要的水速和水温。

在开始下面的代码之前，先定义一个辅助函数用于输出数据：

```
func writeSequenceToConsole<O: ObservableType>(name: String, seq
uence: O) {
    sequence
        .subscribe { e in
            print("Subscription: \(name), event: \(e)")
        }
}
```

PublishSubject

PublishSubject 会发送订阅者从订阅之后的事件序列。

```
example("PublishSubject") {
    let subject = PublishSubject<String>()
    writeSequenceToConsole("1", sequence: subject)
    subject.on(.Next("a"))
    subject.on(.Next("b"))
    writeSequenceToConsole("2", sequence: subject)
    subject.on(.Next("c"))
    subject.on(.Next("d"))
}
--- PublishSubject example ---
Subscription: 1, event: Next(a)
Subscription: 1, event: Next(b)
Subscription: 1, event: Next(c)
Subscription: 2, event: Next(c)
Subscription: 1, event: Next(d)
Subscription: 2, event: Next(d)
```

ReplaySubject

ReplaySubject 在新的订阅对象订阅的时候会补发所有已经发送过的数据队列，bufferSize 是缓冲区的大小，决定了补发队列的最大值。如果 bufferSize 是1，那么新的订阅者出现的时候就会补发上一个事件，如果是2，则补两个，以此类推。

```
example("ReplaySubject") {
    let subject = ReplaySubject<String>.create(bufferSize: 1)
    writeSequenceToConsole("1", sequence: subject)
    subject.on(.Next("a"))
    subject.on(.Next("b"))
    writeSequenceToConsole("2", sequence: subject)
    subject.on(.Next("c"))
    subject.on(.Next("d"))
}
--- ReplaySubject example ---
Subscription: 1, event: Next(a)
Subscription: 1, event: Next(b)
Subscription: 2, event: Next(b) // 补了一个 b
Subscription: 1, event: Next(c)
Subscription: 2, event: Next(c)
Subscription: 1, event: Next(d)
Subscription: 2, event: Next(d)
```

BehaviorSubject

BehaviorSubject 在新的订阅对象订阅的时候会发送最近发送的事件，如果没有则发送一个默认值。

```
example("BehaviorSubject") {
    let subject = BehaviorSubject(value: "z")
    writeSequenceToConsole("1", sequence: subject)
    subject.on(.Next("a"))
    subject.on(.Next("b"))
    writeSequenceToConsole("2", sequence: subject)
    subject.on(.Next("c"))
    subject.on(.Completed)
}
--- BehaviorSubject example ---
Subscription: 1, event: Next(z)
Subscription: 1, event: Next(a)
Subscription: 1, event: Next(b)
Subscription: 2, event: Next(b)
Subscription: 1, event: Next(c)
Subscription: 2, event: Next(c)
Subscription: 1, event: Completed
Subscription: 2, event: Completed
```

Variable

Variable 是基于 BehaviorSubject 的一层封装，它的优势是：不会被显式终结。即：不会收到 .Completed 和 .Error 这类的终结事件，它会主动在析构的时候发送 .Complete 。

```
example("Variable") {
    let variable = Variable("z")
    writeSequenceToConsole("1", sequence: variable)
    variable.value = "a"
    variable.value = "b"
    writeSequenceToConsole("2", sequence: variable)
    variable.value = "c"
}
--- Variable example ---
Subscription: 1, event: Next(z)
Subscription: 1, event: Next(a)
Subscription: 1, event: Next(b)
Subscription: 2, event: Next(b)
Subscription: 1, event: Next(c)
Subscription: 2, event: Next(c)
Subscription: 1, event: Completed
Subscription: 2, event: Completed
```

Transform

我们可以对序列做一些转换，类似于 Swift 中 CollectionType 的各种转换。在以前的坑中曾经提到过，可以参考：函数式的函数。

map

map 就是对每个元素都用函数做一次转换，挨个映射一遍。

```
example("map") {
    let originalSequence = sequenceOf(1,2,3)
    originalSequence
        .map { $0 * 2 }
        .subscribe { print($0) }
}
--- map example ---
Next(2)
Next(4)
Next(6)
Completed
```

flatMap

map 在做转换的时候很容易出现『升维』的情况，即：转变之后，从一个序列变成了一个序列的序列。

什么是『升维』？在集合中我们可以举这样一个例子，我有一个好友列表 [p1, p2, p3]，那么如果要获取我好友的好友的列表，可以这样做：

```
myFriends.map { $0.getFriends() }
```

结果就成了 [[p1-1, p1-2, p1-3], [p2-1], [p3-1, p3-2]] ，这就成了好友的好友列表的列表了。这就是一个『升维』的例子。

在 Swift 中，我们可以用 flatMap 过滤掉 map 之后的 nil 结果。在 Rx 中， flatMap 可以把一个序列转换成一组序列，然后再把这一组序列『拍扁』成一个序列。

```
example("flatMap") {
    let sequenceInt = sequenceOf(1, 2, 3)
    let sequenceString = sequenceOf("A", "B", "--")
    sequenceInt
        .flatMap { int in
            sequenceString
        }
        .subscribe {
            print($0)
        }
}
--- flatMap example ---
Next(A)
Next(B)
Next(--)
Next(A)
Next(B)
Next(--)
Next(A)
Next(B)
Next(--)
Completed
```

scan

scan 有点像 reduce ，它会把每次的运算结果累积起来，作为下一次运算的输入值。

```
example("scan") {
    let sequenceToSum = sequenceOf(0, 1, 2, 3, 4, 5)
    sequenceToSum
        .scan(0) { acum, elem in
            acum + elem
        }
        .subscribe {
            print($0)
        }
}
--- scan example ---
Next(0)
Next(1)
Next(3)
Next(6)
Next(10)
Next(15)
Completed
```

Filtering

除了上面的各种转换，我们还可以对序列进行过滤。

filter

filter 只会让符合条件的元素通过。

```
example("filter") {
    let subscription = sequenceOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
        .filter {
            $0 % 2 == 0
        }
        .subscribe {
            print($0)
        }
}
--- filter example ---
Next(0)
Next(2)
Next(4)
Next(6)
Next(8)
Completed
```

distinctUntilChanged

distinctUntilChanged 会废弃掉重复的事件。

```
example("distinctUntilChanged") {
    let subscription = sequenceOf(1, 2, 3, 1, 1, 4)
        .distinctUntilChanged()
        .subscribe {
            print($0)
        }
}
--- distinctUntilChanged example ---
Next(1)
Next(2)
Next(3)
Next(1)
Next(4)
Completed
```

take

take 只获取序列中的前 n 个事件，在满足数量之后会自动 .Completed 。

```
example("take") {
    let subscription = sequenceOf(1, 2, 3, 4, 5, 6)
        .take(3)
        .subscribe {
            print($0)
        }
}
--- take example ---
Next(1)
Next(2)
Next(3)
Completed
```

Combining

这部分是关于序列的运算，可以将多个序列源进行组合拼装成一个新的事件序列。

startWith

startWith 会在队列开始之前插入一个事件元素。

```
example("startWith") {
    let subscription = sequenceOf(4, 5, 6)
        .startWith(3)
        .subscribe {
            print($0)
        }
}
--- startWith example ---
Next(3)
Next(4)
Next(5)
Next(6)
Completed
```

combineLatest

如果存在两条事件队列，需要同时监听，那么每当有新的事件发生的时候，combineLatest 会将每个队列的最新的一个元素进行合并。

```
example("combineLatest 1") {
    let intOb1 = PublishSubject<String>()
    let intOb2 = PublishSubject<Int>()
    combineLatest(intOb1, intOb2) {
        "\($0) \($1)"
        }
        .subscribe {
            print($0)
        }
    intOb1.on(.Next("A"))
    intOb2.on(.Next(1))
    intOb1.on(.Next("B"))
    intOb2.on(.Next(2))
}
--- combineLatest 1 example ---
Next(A 1)
Next(B 1)
Next(B 2)
```

zip

zip 人如其名，就是合并两条队列用的，不过它会等到两个队列的元素一一对应地凑齐了之后再合并，正如百折不挠的米斯特菜所提醒的， zip 就像是拉链一样，两根拉链拉着拉着合并到了一根上：

```
example("zip 1") {
    let intOb1 = PublishSubject<String>()
    let intOb2 = PublishSubject<Int>()
    zip(intOb1, intOb2) {
        "\($0) \($1)"
        }
        .subscribe {
            print($0)
        }
    intOb1.on(.Next("A"))
    intOb2.on(.Next(1))
    intOb1.on(.Next("B"))
    intOb1.on(.Next("C"))
    intOb2.on(.Next(2))
}
--- zip 1 example ---
Next(A 1)
Next(B 2)
```

merge

merge 就是 merge 啦，把两个队列按照顺序组合在一起。

```
example("merge 1") {
    let subject1 = PublishSubject<Int>()
    let subject2 = PublishSubject<Int>()
    sequenceOf(subject1, subject2)
        .merge()
        .subscribeNext { int in
            print(int)
        }
    subject1.on(.Next(1))
    subject1.on(.Next(2))
    subject2.on(.Next(3))
    subject1.on(.Next(4))
    subject2.on(.Next(5))
}
--- merge 1 example ---
1
2
3
4
5
```

switch

当你的事件序列是一个事件序列的序列 (Observable>) 的时候，（可以理解成二维序列？），可以使用 switch 将序列的序列平铺成一维，并且在出现新的序列的时候，自动切换到最新的那个序列上。和 merge 相似的是，它也是起到了将多个序列『拍平』成一条序列的作用。

```
example("switchLatest") {
    let var1 = Variable(0)
    let var2 = Variable(200)
    // var3 is like an Observable<Observable<Int>>
    let var3 = Variable(var1)
    let d = var3
        .switchLatest()
        .subscribe {
            print($0)
        }
    var1.value = 1
    var1.value = 2
    var1.value = 3
    var1.value = 4
    var3.value = var2
    var2.value = 201
    var1.value = 5
    var3.value = var1
    var2.value = 202
    var1.value = 6
}
--- switchLatest example ---
Next(0)
Next(1)
Next(2)
Next(3)
Next(4)
Next(200)
Next(201)
Next(5)
Next(6)
```

注意，虽然都是『拍平』，但是和 flatmap 是不同的， flatmap 是将一条序列变成另一条序列，而这变换过程会让维度变高，所以需要『拍平』，而 switch 是将本来二维的序列（序列的序列）拍平成了一维的序列。

Error Handling

在事件序列中，遇到异常也是很正常的事情，有以下几种处理异常的手段。

catchError

catchError 可以捕获异常事件，并且在后面无缝接上另一段事件序列，丝毫没有异常的痕迹。

```
example("catchError 1") {
    let sequenceThatFails = PublishSubject<Int>()
    let recoverySequence = sequenceOf(100, 200)
    sequenceThatFails
        .catchError { error in
            return recoverySequence
        }
        .subscribe {
            print($0)
        }
    sequenceThatFails.on(.Next(1))
    sequenceThatFails.on(.Next(2))
    sequenceThatFails.on(.Error(NSError(domain: "Test", code: 0,
 userInfo: nil)))
}
--- catchError 1 example ---
Next(1)
Next(2)
Next(100)
Next(200)
Completed
```

retry

retry 顾名思义，就是在出现异常的时候会再去从头订阅事件序列，妄图通过『从头再来』解决异常。

```
example("retry") {
    var count = 1 // bad practice, only for example purposes
    let funnyLookingSequence: Observable<Int> = create { observe
r in
        let error = NSError(domain: "Test", code: 0, userInfo: n
il)
        observer.on(.Next(0))
        observer.on(.Next(1))
        if count < 2 {
            observer.on(.Error(error))
            count++
        }
        observer.on(.Next(2))
        observer.on(.Completed)
        return NopDisposable.instance
    }
    funnyLookingSequence
        .retry()
        .subscribe {
            print($0)
        }
}
--- retry example ---
Next(0)
Next(1)
Next(0)
Next(1)
Next(2)
Completed
```

Utility

这里列举了针对事件序列的一些方法。

subscribe

subscribe 在前面已经接触过了，有新的事件就会触发。

```
example("subscribe") {
    let sequenceOfInts = PublishSubject<Int>()
    sequenceOfInts
        .subscribe {
            print($0)
        }
    sequenceOfInts.on(.Next(1))
    sequenceOfInts.on(.Completed)
}
--- subscribe example ---
Next(1)
Completed
```

subscribeNext

subscribeNext 也是订阅，但是只订阅 .Next 事件。

```
example("subscribeNext") {
    let sequenceOfInts = PublishSubject<Int>()
    sequenceOfInts
        .subscribeNext {
            print($0)
        }
    sequenceOfInts.on(.Next(1))
    sequenceOfInts.on(.Completed)
}
--- subscribeNext example ---
1
```

subscribeCompleted

subscribeCompleted 是只订阅 .Completed 完成事件。

```
example("subscribeCompleted") {
    let sequenceOfInts = PublishSubject<Int>()
    sequenceOfInts
        .subscribeCompleted {
            print("It's completed")
        }
    sequenceOfInts.on(.Next(1))
    sequenceOfInts.on(.Completed)
}
--- subscribeCompleted example ---
It's completed
```

subscribeError

subscribeError 只订阅 .Error 失败事件。

```
example("subscribeError") {
    let sequenceOfInts = PublishSubject<Int>()
    sequenceOfInts
        .subscribeError { error in
            print(error)
        }
    sequenceOfInts.on(.Next(1))
    sequenceOfInts.on(.Error(NSError(domain: "Examples", code: -
1, userInfo: nil)))
}
--- subscribeError example ---
Error Domain=Examples Code=-1 "The operation couldn’t be complet
ed. (Examples error -1.)"
```

doOn

doOn 可以监听事件，并且在事件发生之前调用。

```
example("doOn") {
    let sequenceOfInts = PublishSubject<Int>()
    sequenceOfInts
        .doOn {
            print("Intercepted event \($0)")
        }
        .subscribe {
            print($0)
        }
    sequenceOfInts.on(.Next(1))
    sequenceOfInts.on(.Completed)
}
--- doOn example ---
Intercepted event Next(1)
Next(1)
Intercepted event Completed
Completed
```

Conditional

我们可以对多个事件序列做一些复杂的逻辑判断。

takeUntil

takeUntil 其实就是 take ，它会在终于等到那个事件之后触发 .Completed 事件。

```
example("takeUntil") {
    let originalSequence = PublishSubject<Int>()
    let whenThisSendsNextWorldStops = PublishSubject<Int>()
    originalSequence
        .takeUntil(whenThisSendsNextWorldStops)
        .subscribe {
            print($0)
        }
    originalSequence.on(.Next(1))
    originalSequence.on(.Next(2))
    whenThisSendsNextWorldStops.on(.Next(1))
    originalSequence.on(.Next(3))
}
--- takeUntil example ---
Next(1)
Next(2)
Completed
```

takeWhile

takeWhile 则是可以通过状态语句判断是否继续 take 。

```
example("takeWhile") {
    let sequence = PublishSubject<Int>()
    sequence
        .takeWhile { int in
            int < 2
        }
        .subscribe {
            print($0)
        }
    sequence.on(.Next(1))
    sequence.on(.Next(2))
    sequence.on(.Next(3))
}
--- takeWhile example ---
Next(1)
Completed
```

Aggregate

我们可以对事件序列做一些集合运算。

concat

concat 可以把多个事件序列合并起来。

```
example("concat") {
    let var1 = BehaviorSubject(value: 0)
    let var2 = BehaviorSubject(value: 200)
    // var3 is like an Observable<Observable<Int>>
    let var3 = BehaviorSubject(value: var1)
    let d = var3
        .concat()
        .subscribe {
            print($0)
        }
    var1.on(.Next(1))
    var1.on(.Next(2))
    var3.on(.Next(var2))
    var2.on(.Next(201))
    var1.on(.Next(3))
    var1.on(.Completed)
    var2.on(.Next(202))
}
--- concat example ---
Next(0)
Next(1)
Next(2)
Next(3)
Next(201)
Next(202)
```

reduce

这里的 reduce 和 CollectionType 中的 reduce 是一个意思，都是指通过对一系列数据的运算最后生成一个结果。

```
example("reduce") {
    sequenceOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
        .reduce(0, +)
        .subscribe {
            print($0)
        }
}
--- reduce example ---
Next(45)
Completed
```

Connectable

坑待填，Xcode 里这个操场跑不起来了。

更新：评论区有同学提醒，跑不起来的原因参考：Using NSTimer in swift playground。

Next

Run the playground in your Xcode!

RxSwift示例实战

Intro

这部分主要是学习 RxSwift 项目中的示例项目，了解 RxSwift 在实际 iOS 开发中的正确打开方式。

Demo1: GitHub Signup

第一个示例是 GitHub 注册账号的例子。输入用户名、密码、重复密码，然后提交注册。

username

在注册流程中，用户名校验是一个很常见的功能。我们一般需要对用户名做如下检查和流程：

- 是否不为空
- 是否不包含非法字符
- 是否没有被注册过
- 以上均通过，联网注册
- 是否成功连接上服务器
- 服务器是否正确处理并返回结果

要通过 RxSwift 实现以上流程，可以划分成如下几步。

rx_text

如果想监听文字输入，我们最好能有一个 Observable 的对象不断地给我们发送新的输入值。rx_text 是 RxSwift 针对 Cocoa 库做的各种封装中的一个，可以简单看下它的定义：

```
extension UITextField {
    /**
    Reactive wrapper for `text` property.
    */
    public var rx_text: ControlProperty<String> {
        return rx_value(getter: { [weak self] in
            self?.text ?? ""
        }, setter: { [weak self] value in
            self?.text = value
        })
    }
}
```

返回的 ControlProperty 遵循 ControlPropertyType 协议：

```
public protocol ControlPropertyType : ObservableType, ObserverTy
pe {

    /**
    - returns: `ControlProperty` interface
    */
    func asControlProperty() -> ControlProperty<E>
}
```

可以看到，它既是一个可被订阅者 (ObservableType)，又是一个订阅者 (ObserverType)，也就是说它是一个 Subject 对象。由于它是 Observable 的，所以我们可以通过 map 把它转换成想要的输出，例如下面这段代码会实现『每输入一个字就在控制台输出当前内容』的功能（记得 RAC 的系列教程似乎也是这个节奏）：

```
    @IBOutlet weak var usernameOutlet: UITextField!
    override func viewDidLoad() {
        super.viewDidLoad()
        let username = usernameOutlet.rx_text
        username.subscribeNext {
            print($0)
        }
}
```

validate

接下来就是验证用户名的阶段了。首先需要一个方法，将用户名转换成一串流。为什么是流？为了统一口径。先看代码：

```
typealias ValidationResult = (valid: Bool?, message: String?)
func validateUsername(username: String) -> Observable<Validation
Result> {
    // 如果用户名为空
    if username.characters.count == 0 {
        return just((false, nil))
    }

    // 如果用户名中出现非法字符
    if username.rangeOfCharacterFromSet(NSCharacterSet.alphanume
ricCharacterSet().invertedSet) != nil {
        return just((false, "Username can only contain numbers o
r digits"))
    }

    // 加载中的值
    let loadingValue = (valid: nil as Bool?, message: "Checking
availabilty ..." as String?)

    // 校验用户名
    return API.usernameAvailable(username)
        .map { available in
            if available {
                return (true, "Username available")
            }
            else {
                return (false, "Username already taken")
            }
        }
        .startWith(loadingValue)    // 在加载结果之前插入 加载中 这个
状态的事件值
}
```

梳理一下：

- 如果用户名为空，返回 (false, nil) 完事儿
- 如果用户名有非法字符，返回 (false, "Username can only contain numbers or digits") 完事儿
- 如果本地检查没问题，发给服务器检查，先发送一个事件 loadingValue 表示正

在加载，加载成功再发送结果事件

这就是我所理解的『统一口径』。虽然本地检查分分钟就能给你个结果，但是如果统一都用『流』来表述，外部处理起来会简单得多。不用管具体的结果是什么，只需要知道是一个 Observable 对象，并且随之而来的是一串事件，就足够了。这一串事件，有可能只有一个，提示用户名不能为空；也有可能有很多，先提示正在加载，然后再提示注册成功。在外部来看，这就是一个事件流。

试想一下如果用 UIKit 的那一套来写这个流程，肯定要监听个 valueChanged 事件然后在委托方法里先判断 A ，如果不符合就刷新 UI ；再判断 B ，不符合就刷新 UI ；最后发请求给服务器，刷新 UI 提示等待，然后加载完了再刷新个 UI 。。。

switch

把上面的验证代码用到项目中大概就是这样：

```
let usernameValidation = username
    .map { username in
        return validationService.validateUsername(username)
    }
```

如果把每次的文字改动事件用 v 来表示，那整个事件序列应该是这样的：

----V------V--V---V---V---- 而用了 validateUsername 之后，它会把以前的值转换成一个新的序列，就成了这样：

----|------|--|---|---|---- | | | | | V V V V V | | | | | | V | V | | | | | 瞬间变成了二维世界了，我们可以用 switch 来『降维』这里我们选用 switchLastest ：

```
let usernameValidation = username
    .map { username in
        return validationService.validateUsername(username)
    }
    .switchLatest()
```

不妨看一下 ObservableType 的定义：

```
extension ObservableType where E : ObservableType {
    public func switchLatest() -> Observable<E.E> {
        return Switch(sources: self.asObservable())
    }
}
```

这个扩展是针对『自己是 ObservableType 且自己监听的事件也是 ObservableType』这一类对象的。通过 switchLatest 方法我们可以将前面的二维结构梳理成一维结构，每次自动切换到新的序列的最新事件上。

----|------|--|---|---|---- V |||||------|||||V||||V|||||--|||||V|||||---|||||V|||||V|||||---|||||V 执行结果如下：

```
let usernameValidation = username
        .map { username in
            return validationService.validateUsername(username)
        }
        .switchLatest()
        .subscribe {
            print("Event: - \($0)")
        }
----- Output -----
// 输入 12!
Event: - Next((Optional(false), Optional("Username can only cont
ain numbers or digits")))
// 输出 abc
Event: - Next((nil, Optional("Checking availabilty ...")))
Event: - Next((Optional(false), Optional("Username already taken
")))
```

replay

我们在 map 里调用了 validateUsername 方法，这会导致如果有多个订阅者的话，会重复调用多次。

例如这个例子：

```
let sequenceOfElements = sequenceOf(0).map { r -> Int in
    print("MAP")    // twice
    return r * 2
}
let subscription = sequenceOfElements
    .subscribe { event in
        print("1 - \(event)")
}
let subscription2 = sequenceOfElements
    .subscribe { event in
        print("2 - \(event)")
}
--- map example ---
MAP
1 - Next(0)
1 - Completed
MAP
2 - Next(0)
2 - Completed
```

我一开始很疑惑，明明订阅的是 map 之后的队列，但是为什么 subscribe 每次都会重新 map 一次呢？

经提醒之后又去仔细翻阅了入门文档 Getting Started ，看到了下面这段话：

> Every subscriber upon subscription usually generates it's own separate sequence of elements. Operators are stateless by default. There is vastly more stateless operators then stateful ones.

这么一想就说得通了，一切都是为了 stateless 。如果 subscribe 的是 map 后的结果，那就意味着需要多存储一个状态，而状态的增加往往意味着复杂度的指数级增长。

为了解决这个多次订阅会多次执行的问题，我们需要 shareReplay ，看下这个示例：

```
let sequenceOfInts = PublishSubject<Int>()
let a = sequenceOfInts.map{ i -> Int in
    print("MAP---\(i)")
    return i * 2
}.shareReplay(3)
let b = a.subscribeNext {
    print("--1--\($0)")
}
sequenceOfInts.on(.Next(1))
sequenceOfInts.on(.Next(2))
let c = a.subscribeNext {
    print("--2--\($0)")
}
sequenceOfInts.on(.Next(3))
sequenceOfInts.on(.Next(4))
let d = a.subscribeNext {
    print("--3--\($0)")
}
sequenceOfInts.on(.Completed)
--- shareReplay example ---
MAP---1
--1--2
MAP---2
--1--4
--2--2
--2--4
MAP---3
--1--6
--2--6
MAP---4
--1--8
--2--8
--3--4
--3--6
--3--8
```

shareReplay 会返回一个新的事件序列，它监听底层序列的事件，并且通知自己的订阅者们。不过和传统的订阅不同的是，它是通过『重播』的方式通知自己的订阅者。就像是过目不忘的看书，但是每次都只记得最后几行的内容，在有人询问的时

候就背诵出来。从上面的例子可以看到，通过 shareReplay 订阅的 map 并不会调用多次。所以我们也可以把它应用到 validateUsername 上：

```
let usernameValidation = username
    .map { username in
        return validationService.validateUsername(username)
    }
    .switchLatest()
    .shareReplay(1)
```

这样就不会出现『多次订阅导致重复地检查用户名是否可用』的情况了。

usernameAvailable

前面梳理了基本的用户名校验流程，接下来看下联网检测这部分是如何实现的。

联网检测用户名是否可用主要是访问用户名对应的 github 地址然后查看是否是 404，如果不是那就说明已经被注册了。核心代码如下：

```
func usernameAvailable(username: String) -> Observable<Bool> {
    let URL = NSURL(string: "https://github.com/\(URLEscape(user
name))")!
    let request = NSURLRequest(URL: URL)
    return self.URLSession.rx_response(request)
        .map { (maybeData, maybeResponse) in
            if let response = maybeResponse as? NSHTTPURLRespons
e {
                return response.statusCode == 404
            }
            else {
                return false
            }
        }
        .observeOn(self.dataScheduler)
        .catchErrorJustReturn(false)
}
```

和前面的 rx_value 相似， rx_response 是针对 NSURLSession 的扩展。通过 observeOn 将监听事件绑定在了 dataScheduler 上。最后 catchErrorJustReturn(false) 表明如果出现异常就返回个 false 。

Scheduler 是一种 Rx 里的任务运行机制，类似的 gcd 里的 dispatch queue 。可以通过 observeOn 切换 scheduler ：

```
sequence1
    .observeOn(backgroundScheduler)
    .map { n in
      println("This is performed on background scheduler")
    }
    .observeOn(MainScheduler.sharedInstance)
    .map { n in
      println("This is performed on main scheduler")
    }
```

password

密码的检测相比较用户名而言就简单很多，核心代码如下：

```
func validatePassword(password: String) -> ValidationResult {
    let numberOfCharacters = password.characters.count
    if numberOfCharacters == 0 {
        return (false, nil)
    }
    if numberOfCharacters < minPasswordCount {
        return (false, "Password must be at least \(minPasswordC
ount) characters")
    }
    return (true, "Password acceptable")
}
```

注意这里返回了 ValidationResult ，因为所有校验都是本地完成的。

接下来就是重复密码的校验，这部分比较有意思，通过 combineLatest 将两个序列合并起来：

```
let repeatPasswordValidation = combineLatest(password, repeatPas
sword) { (password, repeatedPassword) in
        validationService.validateRepeatedPassword(password, rep
eatedPassword: repeatedPassword)
    }
    .shareReplay(1)
```

然后 validateRepeatedPassword 方法如下：

```
func validateRepeatedPassword(password: String, repeatedPassword
: String) -> ValidationResult {
    if repeatedPassword.characters.count == 0 {
        return (false, nil)
    }

    if repeatedPassword == password {
        return (true, "Password repeated")
    }
    else {
        return (false, "Password different")
    }
}
```

这几个例子基本都是把事件序列进行组装然后『外包』给其他对象去处理。

bindValidationResultToUI

检查也检查好了，接下来的就是更新 UI 了，用户名非法、两次密码不一致，这些都需要通过刷新 UI 告知用户。也就是说，需要把前面定义的『事件流』和『用户界面』绑定起来。看下这个绑定的方法：

```
func bindValidationResultToUI(source: Observable<ValidationResul
t>,
    validationErrorLabel: UILabel) {
    source
        .subscribeNext { v in
            let validationColor: UIColor

            if let valid = v.valid {
                validationColor = valid ? okColor : errorColor
            }
            else {
               validationColor = UIColor.grayColor()
            }

            validationErrorLabel.textColor = validationColor
            validationErrorLabel.text = v.message ?? ""
        }
        .addDisposableTo(disposeBag)
}
```

在这里出现了 addDisposableTo(disposeBag)，在此需要解释一下 disposing 的相关概念。

一个事件流的终结除了前面了解的各种事件之外，还有一种方法，就是 dispose，释放掉所有的资源。比如这个例子：

```
let subscription = interval(0.3, scheduler)
    .subscribe { (e: Event<Int64>) in
        println(e)
    }
NSThread.sleepForTimeInterval(2)
subscription.dispose()
----- Dispose Sample -----
0
1
2
3
4
5
```

然而 dispose 方法是不推荐使用的，推荐使用更好的解决方案， DisposeBag 就是一个。addDisposableTo(disposeBag) 有点像是 ARC ，先把分配的资源统一丢到袋子里 (有点像是 autoreleasepool) ，然后当 disposeBag 销毁的时候就一起销毁这些资源。在代码里可以看到，只要有 subscribe 的基本在最后都会兜上一个 .addDisposableTo(disposeBag) 用来处理资源自动销毁的问题。

signupEnabled

检查完毕之后，如果所有条件都符合，那就需要把 Signup 按钮高亮，高亮的逻辑是把多个数据流合并在了一起：

```
let signupEnabled = combineLatest(
     usernameValidation,
     passwordValidation,
     repeatPasswordValidation,
     signingProcess
 ) { un, p, pr, signingState in
     return (un.valid ?? false) && (p.valid ?? false) && (pr.val
id ?? false) && signingState != SignupState.SigningUp
 }
```

在基本的流都构建完毕的情况下，各种需求更多的是对流的组合拼装。比如这里就再次用到了 usernameValidation 这个流，还好前面有 shareReplay 罩着，我们想复用多少次都没问题。

signingProcess

在点击注册按钮之后，就是具体的注册流程了，注册流程的代码是这样的：

```
let signingProcess = combineLatest(username, password) { ($0, $1
) }
    .sampleLatest(signupSampler)
    .map { (username, password) in
        return API.signup(username, password: password)
    }
    .switchLatest()
    .startWith(SignupState.InitialState)
    .shareReplay(1)
```

这里有个 sampleLatest ，在了解它之前先要了解什么是 sample 。

sample

sample 就是一次『采样』，当收到采样事件的时候，就会从事件队列中取出一个事件作为『样本』，并发送到事件流里。如果下一次又要采样了，就会从两次采样之间的事件队列中选择最后一个事件，如果两次采集之间没有新的事件就不会进行任何操作。

可以看下这个例子帮助理解：

```
let s = PublishSubject<Int>()
let o = PublishSubject<String>()
let subscription = s
    .sample(o)
    .subscribe { event in
        print(event)
}
s.on(.Next(1))
o.on(.Next("A"))
s.on(.Next(2))
s.on(.Next(3))
o.on(.Next("B"))
o.on(.Next("C"))
--- sample example ---
Next(1)
Next(3)
```

sampleLatest 就是，即使两次采样期间没有新的事件也没关系，取整个队列的最后一个事件作为输出。还是上面那个例子：

```
let s = PublishSubject<Int>()
let o = PublishSubject<String>()
let subscription = s
    .sampleLatest(o)
    .subscribe { event in
        print(event)
}
s.on(.Next(1))
o.on(.Next("1"))
s.on(.Next(2))
s.on(.Next(3))
o.on(.Next("2"))
o.on(.Next("3"))
--- sample example ---
Next(1)
Next(3)
Next(3)
```

所以上面的注册流程代码也就可以理解了：

- 先把 username 和 password 绑起来
- 将注册按钮的点击事件作为一个触发点，每次点击都会获取最新的账号密码走下面的流程
- 调用 API.signup 进行注册
- 将 map 之后的二维队列拍平，切换到最新的队列上
- 将状态置为初始状态
- 通过 shareReplay 避免重复订阅导致的反复执行的问题

signup

项目里的注册功能只是一个 mock 而已，并没有真的访问 API ：

```
func signup(username: String, password: String) -> Observable<Si
gnupState> {
    // this is also just a mock
    let signupResult = SignupState.SignedUp(signedUp: arc4random
() % 5 == 0 ? false : true)
    return [just(signupResult), never()]
        .concat()
        .throttle(2, MainScheduler.sharedInstance)
        .startWith(SignupState.SigningUp)
}
```

在这里可以看到 never() 的正确打开方式：用于无限等待。 concat 将上面两个序列首尾拼接起来，然后 throttle 等价于 debounce ：如果两个事件的时间间隔小于某个特定值，就会忽视掉前面一个。通过 never + throttle 伪造了一种等待加载2秒然后返回注册结果的错觉。

disposeBag

定义了事件流之后，我们就可以通过 subscribeNext 来刷新 UI 了：

```
signingProcess
    .subscribeNext { [unowned self] signingResult in
        switch signingResult {
        case .SigningUp:
            self.signingUpOulet.hidden = false
        case .SignedUp(let signed):
            self.signingUpOulet.hidden = true

            let alertView: UIAlertView

            if signed {
                alertView = UIAlertView(title: "GitHub", message
: "Mock signed up to GitHub", delegate: nil, cancelButtonTitle:
"OK")
            }
            else {
                alertView = UIAlertView(title: "GitHub", message
: "Mock signed up failed", delegate: nil, cancelButtonTitle: "OK
")
            }

            alertView.show()
        default:
            self.signingUpOulet.hidden = true
        }
    }
    .addDisposableTo(disposeBag)
```

注意，每一次 subscribe 都要及时回收资源，在示例代码中是都通过 addDisposableTo(disposeBag) 统一处理了。在 disposeBag 重新赋值的时候就会自动清理资源。

项目中一共有三个地方调用了 `disposeBag = DisposeBag()` ：

定义变量的时候：

```
var disposeBag = DisposeBag()
```

viewDidLoad 里：

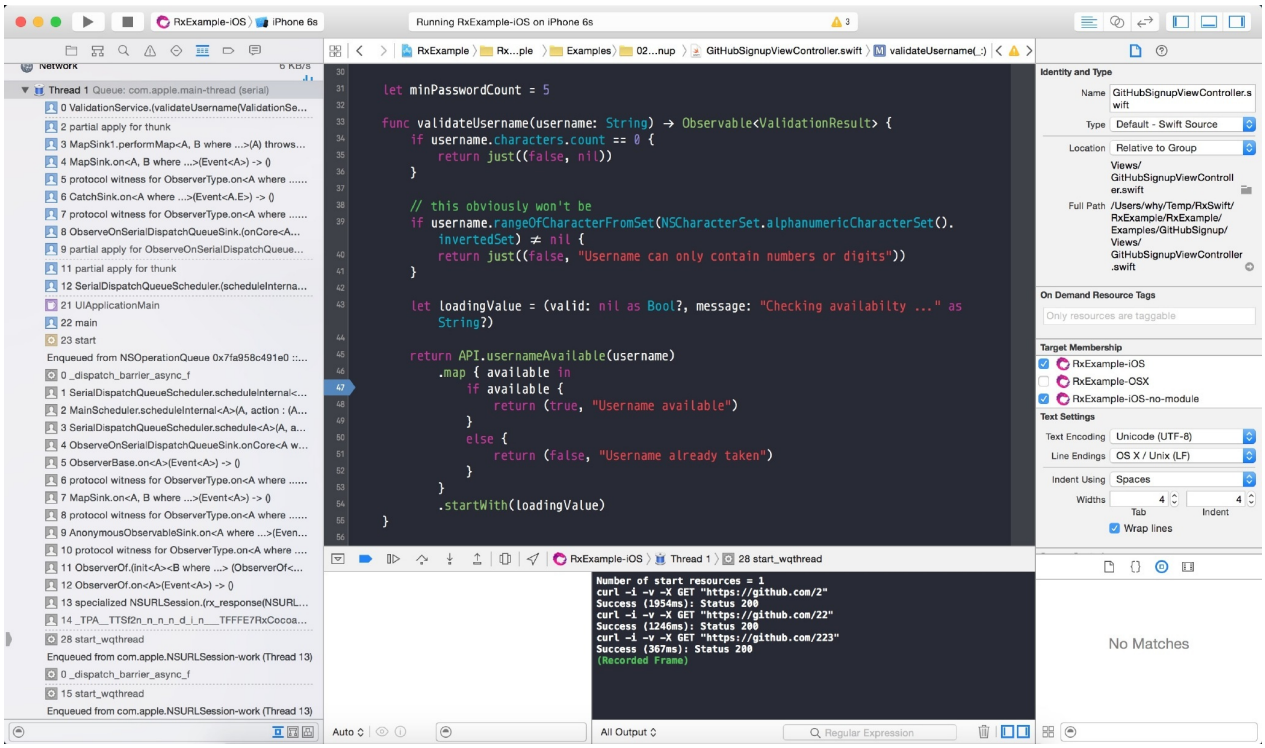
```
override func viewDidLoad() {
    super.viewDidLoad()
    self.disposeBag = DisposeBag()
    ...
}

willMoveToParentViewController 里：
// This is one of the reasons why it's a good idea for disposal
to be detached from allocations.
// If resources weren't disposed before view controller is being
 deallocated, signup alert view
// could be presented on top of wrong screen or crash your app i
f it was being presented while
// navigation stack is popping.
// This will work well with UINavigationController, but has an a
ssumption that view controller will
// never be readded as a child view controller.
// It it was readded UI wouldn't be bound anymore.
override func willMoveToParentViewController(parent: UIViewContr
oller?) {
    if let parent = parent {
        assert(parent.isKindOfClass(UINavigationController), "Pl
ease read comments")
    }
    else {
        self.disposeBag = DisposeBag()
    }
}
```

在 UINavigationController 中，这样的代码没有问题，但是当把这个 view controller 作为 child view controller 添加到其他界面的时候，会直接走到断言处。原因是在 child view controller 中，会先调用 viewDidLoad 再调用 willMoveToParentViewController ，好不容易绑定好的界面和事件流，结果直接 self.disposeBag = DisposeBag() 就给解绑了，自然出了问题。

未完待续

Next

各种异步各种回调的好处是整个应用行云流水让人感觉十分舒适，坏处是和 RAC 一样断点调试基本就是噩梦：



RxSwift实现一个UITableView

前言

因为和同事突然决定要在项目里使用MVVM架构 + 响应式编程 + Swift，最近一直在撸RxSwift。由于没有很完善的中文教程和文档，所以学习的过程中遇到了很多坑，比如一个简单的实现UITableView就搞了好久... 于是决定把自己遇到的坑都记录下来，顺便翻译一些有用的英文材料，给后来踩坑的人留下一些经验。

今天要介绍的就是UITableView在RxSwift中的使用方法，我也是Google了好些资料，最后找到了这篇《Implement a UITableView in RxSwift》博文，按照上面的例子实现了UITableView。今天要讲的UITableView的用法也是主要翻译这篇博文，加上自己的一些改良。

长话短说，让我们开始吧。

RxSwift是什么

RxSwift是一个针对于Swift语言的响应式编程框架，旨在使异步操作和事件/数据流的实现变的简单。这里不做过多的介绍，直接进入教程。

示例

使用Xcode新建一个工程，并把语言选择为Swift。然后添加RxSwift框架到你的工程里。你可以使用CocoaPods来管理三方库。 你的Podflie文件看起来应该是这样的：

```
source 'https://github.com/CocoaPods/Specs.git'
platform :ios, ‘8.1’
use_frameworks!

target 'RxTableView' do
pod 'RxSwift'
pod 'RxCocoa'
end
```

注意！确保你安装了**RxDataSources**这个三方库！

RxDataSources是使用RxSwift对UITableView和UICollectionView的数据源做了一层包装。作者一开始在尝试的时候就没有包含这个库，结果一启动就Crash，一启动就Crash，无限循环... 最可恶的是官方给的Example里面没有用Pods加入这个库，而是手动放到工程里的，没仔细看目录结构之前都不知道有这个鬼东西...

所以，实际上，你的Podfile文件里还要加上RxDataSources，这样你的Podflie看起来应该是这样的：

```
source 'https://github.com/CocoaPods/Specs.git'
platform :ios, ‘8.1’
use_frameworks!

target 'RxTableView' do
pod 'RxSwift'
pod 'RxCocoa'
pod 'RxDataSources'
end
```

接着，新建一个ViewController，并给它添加一个UITableView，你的代码看起来应该是这样的：

```
import UIKit
import RxCocoa
import RxSwift
import RxDataSources

class RxTableViewController: UIViewController {
    let tableView: UITableView = UITableView(frame: UIScreen.mai
nScreen().bounds, style: .Plain)
    let reuseIdentifier = "\(TableViewCell.self)"


override func viewDidLoad() {
    super.viewDidLoad()
    view.addSubview(tableView)
    tableView.registerClass(TableViewCell.self, forCellReuseIden
tifier: reuseIdentifier)

    }
}
```

很好，现在我们要开始实现 UITableViewDelegate和UITableViewDataSource方法了，对吧？ 哈，其实不需要这样做。我们使用了响应式的方法来编程，就不在需要写这些数据源和代理方法了。现在你要确保你的控制器里 `import RxSwift` 和 `RxDataSource` 两个模块就可以了，我们使用RxSwift来配置我们的TableView。

细心的你应该会发现作者自定义了一个TableviewCell，这个后面再提。

现在我们创建一个Model，来代表简书的用户对象，它有关注、粉丝、昵称三个属性。

```
import Foundation

struct User {
    let followersCount: Int
    let followingCount: Int
    let screenName: String
}
```

现在你会不会感到困惑，为什么我们的Model使用了一个Struct而不是一个Class呢？ 作者和你一样困惑。呃... 作者翻译的这篇博文的原作者就是这样写的，而且原作者创建的ViewModel文件还包含了UIKit模块，实际上MVVM模式下ViewModel是最好不要包含UI相关的元素的。不过我们先不要在意这些细节，毕竟我们这篇的目的是研究如何使用UITableView不是。

回到我们的ViewController文件，声明这样一个属性：

```
let dataSource = RxTableViewSectionedReloadDataSource<SectionMod
el<String, User>>()
```

RxDataSources类指定了我们的数据源包括哪些内容。SectionModel带有一个String作为section的名字，User类作为item的类型。如果你不太明白的话，可以按住⌘并点击对象的声明来查看它内部的实现是怎样的。

现在我们创建一个ViewModel类用来传递我们的数据源。出于给控制器减负的目的，我们要避免直接在控制器里处理Model类。你的ViewModel看上去应该是这样的：

```
import Foundation
import RxSwift
import RxDataSources

class ViewModel: NSObject {

}
```

让我们为ViewModel类添加一个获取数据的功能。在你的真实的应用中，你的数据更可能是通过网络请求解析JSON数据而获得来的，在我们的例子中，我们先写一段假的数据。

ViewModel的方法看起来应该是这样的：

```
import Foundation
import RxSwift
import RxDataSources

class ViewModel: NSObject {

    func getUsers() -> Observable<[SectionModel<String, User>]>
{
    return Observable.create { (observer) -> Disposable in
        let users = [User(followersCount: 19_901_990, followingC
ount: 1990, screenName: "Marco Sun"),
            User(followersCount: 19_890_000, followingCount: 198
9, screenName: "Taylor Swift"),
            User(followersCount: 250_000, followingCount: 25, sc
reenName: "Rihanna"),
            User(followersCount: 13_000_000_000, followingCount:
 13, screenName: "Jolin Tsai"),
            User(followersCount: 25_000_000, followingCount: 25,
 screenName: "Adele")]
        let section = [SectionModel(model: "", items: users)]
        observer.onNext(section)
        observer.onCompleted()
        return AnonymousDisposable{}
    }
}

}
```

一个Observable是响应式编程里最重要也是最基本的概念。它是一组序列的值。这就是为什么异步操作来获取你的数据如此简单的原因。你可以连接多个Observable，然后等他们全部完成后再刷新数据（看不懂这段的要回去再研究下RxSwift的几个基础概念）。

我们刚刚定义的Observable是一个数组里面装了SectionModel对象，SectionModel里面包含了String型的标题和User型的item。还记得这个类吗？它实际上是我们的TableView的row的数据的容器。我们插入了五个假数据在数组里，并且创建了一个section。section的名字用了一个空的字符串。如果你想给section的区头加上title，你可以填充这个字符串并且在控制器里实现titleForHeaderInSection的方法。

之后我们告诉Observable我们的序列完成了，通过调用onCompleted()方法来编译我们的功能。返回AnonymousDisposable()来确保在函数返回后资源可以得到释放和清理。假如你用了网络请求，你可以在AnonymousDisposable()方法的闭包里取消所有的等待请求。关于事件序列和Disposable()在Getting Started guide里有很好的解释。

写好了ViewModel的逻辑之后，我们回到控制器文件然后把数据源串起来。首先，先创建两个常量ViewModel和DisposeBag。DisposeBag是在控制器销毁后来控制释放资源的。

```
let viewModel = ViewModel()
let disposeBag = DisposeBag()
```

在viewDidLoad()方法里，我们配置UItableViewCell然后绑定ViewModel给TableView的数据源。

```
override func viewDidLoad() {

    super.viewDidLoad()
    view.addSubview(tableView)
    tableView.registerClass(TableViewCell.self, forCellReuseIden
tifier: reuseIdentifier)

    dataSource.configureCell = {
        _, tableView, indexPath, user in
        let cell = tableView.dequeueReusableCellWithIdentifier(s
elf.reuseIdentifier, forIndexPath: indexPath) as! TableViewCell
        cell.tag = indexPath.row
        cell.user = user
        return cell
    }

    viewModel.getUsers()
        .bindTo(tableView.rx_itemsWithDataSource(dataSource))
        .addDisposableTo(disposeBag)
}
```

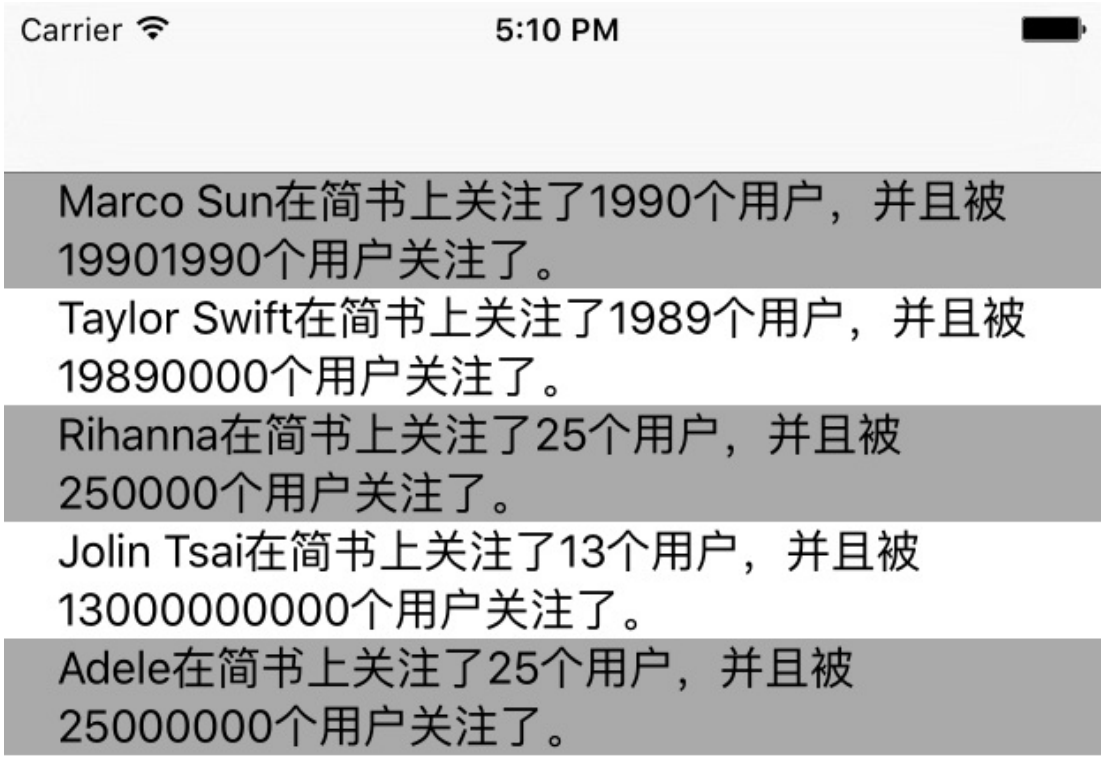
这里我们使用了自定义的TableViewCell类，并给它设置了一个user属性，TableViewCell类的内部应该是这样的：

```
import UIKit

class TableViewCell: UITableViewCell {

var user: User? {
    willSet {
        let string = "\(newValue!.screenName)在简书上关注了\(newVa
lue!.followingCount)个用户，并且被\(newValue!.followersCount)个用户
关注了。"
        backgroundColor = tag % 2 == 0 ? UIColor.lightGrayColor(
) : UIColor.whiteColor()
        textLabel?.text = string
        textLabel?.numberOfLines = 0
        }
    }

}
```

我们重写了cell的willSet方法来利用数据做UI展示。

在上面的viewDidLoad()方法中我们最后还使用了Observable的getUser方法来返回数据源。viewModel会返回SectionModel对象，tableview会自动的展示数据，真棒！~

在手机上展示的真实页面是这样的：


Carrier
5:10 PM
Marco Sun在简书上关注了1990个用户，并且被19901990个用户关注了。
Taylor Swift在简书上关注了1989个用户，并且被19890000个用户关注了。
Rihanna在简书上关注了25个用户，并且被250000个用户关注了。
Jolin Tsai在简书上关注了13个用户，并且被13000000000个用户关注了。
Adele在简书上关注了25个用户，并且被25000000个用户关注了。

RxDataSources的功能是很强大的。除了使用 `RxTableViewSectionedReloadDataSource` 我们还可以使用 `RxTableViewSectionedAnimatedDataSource` 来进行动画操作，它对UICollectionView也有很好的支持。

我们可以扩展这个例子让它有更多的功能，这个项目的Demo放在了Github上，如果你需要可以直接下载它来使用。