**RAC技术分享**

吴鹏 2017.5.23

目录：

1. **ReactiveCocoa简介**

**2. ReactiveCocoa作用**

**3. ReactiveCocoa编程思想**

**4. ReactiveCocoa常见类。**

4.1 **RACSiganl**

4.2 **RACSubscriber**

4.3 **RACDisposable**

4.4 **RACSubject**

4.5 **RACTuple**

4.6 **RACSequence**

4.7 **RACCommand**

4.8 **RACMulticastConnection**

4.9 **RACScheduler**

**5.ReactiveCocoa常见操作方法介绍**

5.1 ReactiveCocoa操作须知

5.2 ReactiveCocoa操作思想

5.3 ReactiveCocoa核心方法

5.4 ReactiveCocoa操作方法之映射

5.5 ReactiveCocoa操作方法之组合

5.6 ReactiveCocoa操作方法之过滤

5.7 ReactiveCocoa操作方法之秩序

5.8 ReactiveCocoa操作方法之线程

5.9 ReactiveCocoa操作方法之时间

5.10 ReactiveCocoa操作方法之重复

**1.ReactiveCocoa简介**

ReactiveCocoa（简称为RAC）,是由Github开源的一个应用于iOS和OS开发的新框架,Cocoa是苹果整套框架的简称，因此很多苹果框架喜欢以Cocoa结尾。

**2.ReactiveCocoa作用**

ReactiveCocoa为事件提供了很多处理方法，而且利用RAC处理事件很方便，可以把要处理的事情，和监听的事情的代码放在一起，这样非常方便我们管理，就不需要跳到对应的方法里。非常符合我们开发中高聚合，低耦合的思想。

**3.ReactiveCocoa编程思想**

ReactiveCocoa结合了几种编程风格：

函数式编程（Functional Programming）：是把操作尽量写成一系列嵌套的函数或者方法调用。

函数式编程特点：每个方法必须有返回值（本身对象）,把函数或者Block当做参数,block参数（需要操作的值）block返回值（操作结果）

响应式编程（Reactive Programming）：不需要考虑调用顺序，只需要知道考虑结果，类似于蝴蝶效应，产生一个事件，会影响很多东西，这些事件像流一样的传播出去，然后影响结果，借用面向对象的一句话，万物皆是流。

代表：KVO运用

所以，你可能听说过ReactiveCocoa被描述为函数响应式编程（FRP）框架。

以后使用RAC解决问题，就不需要考虑调用顺序，直接考虑结果，把每一次操作都写成一系列嵌套的方法中，使代码高聚合，方便管理。

**4.ReactiveCocoa常见类。**

4.1 **RACSiganl**:信号类,一般表示将来有数据传递，只要有数据改变，信号内部接收到数据，就会马上发出数据。

注意：

* 信号类(RACSiganl)，只是表示当数据改变时，信号内部会发出数据，它本身不具备发送信号的能力，而是交给内部一个订阅者去发出。
* 默认一个信号都是冷信号，也就是值改变了，也不会触发，只有订阅了这个信号，这个信号才会变为热信号，值改变了才会触发。

如何订阅信号：调用信号RACSignal的subscribeNext就能订阅。

// RACSignal使用步骤：

// 1.创建信号 + (RACSignal \*)createSignal:(RACDisposable \* (^)(id<RACSubscriber> subscriber))didSubscribe

// 2.订阅信号,才会激活信号. - (RACDisposable \*)subscribeNext:(void (^)(id x))nextBlock

// 3.发送信号 - (void)sendNext:(id)value

// RACSignal底层实现：

// 1.创建信号，首先把didSubscribe保存到信号中，还不会触发。

// 2.当信号被订阅，也就是调用signal的subscribeNext:nextBlock

// 2.2 subscribeNext内部会创建订阅者subscriber，并且把nextBlock保存到subscriber中。

// 2.1 subscribeNext内部会调用siganl的didSubscribe

// 3.siganl的didSubscribe中调用[subscriber sendNext:@1];

// 3.1 sendNext底层其实就是执行subscriber的nextBlock

// 1.创建信号

RACSignal \*siganl = [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

// block调用时刻：每当有订阅者订阅信号，就会调用block。

// 2.发送信号

[subscriber sendNext:@1];

// 如果不在发送数据，最好发送信号完成，内部会自动调用[RACDisposable disposable]取消订阅信号。

[subscriber sendCompleted];

return [RACDisposable disposableWithBlock:^{

// block调用时刻：当信号发送完成或者发送错误，就会自动执行这个block,取消订阅信号。

// 执行完Block后，当前信号就不在被订阅了。

NSLog(@"信号被销毁");

}];

}];

// 3.订阅信号,才会激活信号.

[siganl subscribeNext:^(id x) {

// block调用时刻：每当有信号发出数据，就会调用block.

NSLog(@"接收到数据:%@",x);

}];

4.2 **RACSubscriber**:表示订阅者的意思，用于发送信号，这是一个协议，不是一个类，只要遵守这个协议，并且实现方法才能成为订阅者。通过create创建的信号，都有一个订阅者，帮助他发送数据。

4.3 **RACDisposable**:用于取消订阅或者清理资源，当信号发送完成或者发送错误的时候，就会自动触发它。

使用场景:不想监听某个信号时，可以通过它主动取消订阅信号。

4.4 **RACSubject**:信号提供者，自己可以充当信号，又能发送信号。

* 使用场景:通常用来代替代理，有了它，就不必要定义代理了。

RACReplaySubject:重复提供信号类，RACSubject的子类。

* RACReplaySubject与RACSubject区别:

RACReplaySubject可以先发送信号，在订阅信号，RACSubject就不可以。

* 使用场景一:如果一个信号每被订阅一次，就需要把之前的值重复发送一遍，使用重复提供信号类。
* 使用场景二:可以设置capacity数量来限制缓存的value的数量,即只缓充最新的几个值。
* RACSubject和RACReplaySubject简单使用:

// RACSubject使用步骤

// 1.创建信号 [RACSubject subject]，跟RACSiganl不一样，创建信号时没有block。

// 2.订阅信号 - (RACDisposable \*)subscribeNext:(void (^)(id x))nextBlock

// 3.发送信号 sendNext:(id)value

// RACSubject:底层实现和RACSignal不一样。

// 1.调用subscribeNext订阅信号，只是把订阅者保存起来，并且订阅者的nextBlock已经赋值了。

// 2.调用sendNext发送信号，遍历刚刚保存的所有订阅者，一个一个调用订阅者的nextBlock。

// 1.创建信号

RACSubject \*subject = [RACSubject subject];

// 2.订阅信号

[subject subscribeNext:^(id x) {

// block调用时刻：当信号发出新值，就会调用.

NSLog(@"第一个订阅者%@",x);

}];

[subject subscribeNext:^(id x) {

// block调用时刻：当信号发出新值，就会调用.

NSLog(@"第二个订阅者%@",x);

}];

// 3.发送信号

[subject sendNext:@"1"];

// RACReplaySubject使用步骤:

// 1.创建信号 [RACSubject subject]，跟RACSiganl不一样，创建信号时没有block。

// 2.可以先订阅信号，也可以先发送信号。

// 2.1 订阅信号 - (RACDisposable \*)subscribeNext:(void (^)(id x))nextBlock

// 2.2 发送信号 sendNext:(id)value

// RACReplaySubject:底层实现和RACSubject不一样。

// 1.调用sendNext发送信号，把值保存起来，然后遍历刚刚保存的所有订阅者，一个一个调用订阅者的nextBlock。

// 2.调用subscribeNext订阅信号，遍历保存的所有值，一个一个调用订阅者的nextBlock

// 如果想当一个信号被订阅，就重复播放之前所有值，需要先发送信号，在订阅信号。

// 也就是先保存值，在订阅值。

// 1.创建信号

RACReplaySubject \*replaySubject = [RACReplaySubject subject];

// 2.发送信号

[replaySubject sendNext:@1];

[replaySubject sendNext:@2];

// 3.订阅信号

[replaySubject subscribeNext:^(id x) {

NSLog(@"第一个订阅者接收到的数据%@",x);

}];

// 订阅信号

[replaySubject subscribeNext:^(id x) {

NSLog(@"第二个订阅者接收到的数据%@",x);

}];

* RACSubject替换代理

// 需求:

// 1.给当前控制器添加一个按钮，modal到另一个控制器界面

// 2.另一个控制器view中有个按钮，点击按钮，通知当前控制器

步骤一：在第二个控制器.h，添加一个RACSubject代替代理。

@interface TwoViewController : UIViewController

@property (nonatomic, strong) RACSubject \*delegateSignal;

@end

步骤二：监听第二个控制器按钮点击

@implementation TwoViewController

- (IBAction)notice:(id)sender {

// 通知第一个控制器，告诉它，按钮被点了

// 通知代理

// 判断代理信号是否有值

if (self.delegateSignal) {

// 有值，才需要通知

[self.delegateSignal sendNext:nil];

}

}

@end

步骤三：在第一个控制器中，监听跳转按钮，给第二个控制器的代理信号赋值，并且监听.

@implementation OneViewController

- (IBAction)btnClick:(id)sender {

// 创建第二个控制器

TwoViewController \*twoVc = [[TwoViewController alloc] init];

// 设置代理信号

twoVc.delegateSignal = [RACSubject subject];

// 订阅代理信号

[twoVc.delegateSignal subscribeNext:^(id x) {

NSLog(@"点击了通知按钮");

}];

// 跳转到第二个控制器

[self presentViewController:twoVc animated:YES completion:nil];

}

@end

4.5 **RACTuple**:元组类,类似NSArray,用来包装值.(好比数据库表里的各列对应不同的属性，那么它的每一行对应一个元组).

4.6 **RACSequence**:RAC中的集合类，用于代替NSArray,NSDictionary,可以使用它来快速遍历数组和字典。

使用场景：1.字典转模型

RACSequence和RACTuple简单使用

// 1.遍历数组

NSArray \*numbers = @[@1,@2,@3,@4];

// 这里其实是三步

// 第一步: 把数组转换成集合RACSequence numbers.rac\_sequence

// 第二步: 把集合RACSequence转换RACSignal信号类,numbers.rac\_sequence.signal

// 第三步: 订阅信号，激活信号，会自动把集合中的所有值，遍历出来。

[numbers.rac\_sequence.signal subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

// 2.遍历字典,遍历出来的键值对会包装成RACTuple(元组对象)

NSDictionary \*dict = @{@"name":@"xmg",@"age":@18};

[dict.rac\_sequence.signal subscribeNext:^(RACTuple \*x) {

// 解包元组，会把元组的值，按顺序给参数里面的变量赋值

RACTupleUnpack(NSString \*key,NSString \*value) = x;

// 相当于以下写法

// NSString \*key = x[0];

// NSString \*value = x[1];

NSLog(@"%@ %@",key,value);

}];

// 3.字典转模型

// 3.1 OC写法

NSString \*filePath = [[NSBundle mainBundle] pathForResource:@"flags.plist" ofType:nil];

NSArray \*dictArr = [NSArray arrayWithContentsOfFile:filePath];

NSMutableArray \*items = [NSMutableArray array];

for (NSDictionary \*dict in dictArr) {

FlagItem \*item = [FlagItem flagWithDict:dict];

[items addObject:item];

}

// 3.2 RAC写法

NSString \*filePath = [[NSBundle mainBundle] pathForResource:@"flags.plist" ofType:nil];

NSArray \*dictArr = [NSArray arrayWithContentsOfFile:filePath];

NSMutableArray \*flags = [NSMutableArray array];

\_flags = flags;

// rac\_sequence注意点：调用subscribeNext，并不会马上执行nextBlock，而是会等一会。

[dictArr.rac\_sequence.signal subscribeNext:^(id x) {

// 运用RAC遍历字典，x：字典

FlagItem \*item = [FlagItem flagWithDict:x];

[flags addObject:item];

}];

NSLog(@"%@", NSStringFromCGRect([UIScreen mainScreen].bounds));

// 3.3 RAC高级写法:

NSString \*filePath = [[NSBundle mainBundle] pathForResource:@"flags.plist" ofType:nil];

NSArray \*dictArr = [NSArray arrayWithContentsOfFile:filePath];

// map:映射的意思，目的：把原始值value映射成一个新值

// array: 把集合转换成数组

// 底层实现：当信号被订阅，会遍历集合中的原始值，映射成新值，并且保存到新的数组里。

NSArray \*flags = [[dictArr.rac\_sequence map:^id(id value) {

return [FlagItem flagWithDict:value];

}] array];

4.7 **RACCommand**:RAC中用于处理事件的类，可以把事件如何处理,事件中的数据如何传递，包装到这个类中，他可以很方便的监控事件的执行过程。

使用场景:监听按钮点击，网络请求

RACCommand简单使用

// 一、RACCommand使用步骤:

// 1.创建命令 initWithSignalBlock:(RACSignal \* (^)(id input))signalBlock

// 2.在signalBlock中，创建RACSignal，并且作为signalBlock的返回值

// 3.执行命令 - (RACSignal \*)execute:(id)input

// 二、RACCommand使用注意:

// 1.signalBlock必须要返回一个信号，不能传nil.

// 2.如果不想要传递信号，直接创建空的信号[RACSignal empty];

// 3.RACCommand中信号如果数据传递完，必须调用[subscriber sendCompleted]，这时命令才会执行完毕，否则永远处于执行中。

// 4.RACCommand需要被强引用，否则接收不到RACCommand中的信号，因此RACCommand中的信号是延迟发送的。

// 三、RACCommand设计思想：内部signalBlock为什么要返回一个信号，这个信号有什么用。

// 1.在RAC开发中，通常会把网络请求封装到RACCommand，直接执行某个RACCommand就能发送请求。

// 2.当RACCommand内部请求到数据的时候，需要把请求的数据传递给外界，这时候就需要通过signalBlock返回的信号传递了。

// 四、如何拿到RACCommand中返回信号发出的数据。

// 1.RACCommand有个执行信号源executionSignals，这个是signal of signals(信号的信号),意思是信号发出的数据是信号，不是普通的类型。

// 2.订阅executionSignals就能拿到RACCommand中返回的信号，然后订阅signalBlock返回的信号，就能获取发出的值。

// 五、监听当前命令是否正在执行executing

// 六、使用场景,监听按钮点击，网络请求

// 1.创建命令

RACCommand \*command = [[RACCommand alloc] initWithSignalBlock:^RACSignal \*(id input) {

NSLog(@"执行命令");

// 创建空信号,必须返回信号

// return [RACSignal empty];

// 2.创建信号,用来传递数据

return [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

[subscriber sendNext:@"请求数据"];

// 注意：数据传递完，最好调用sendCompleted，这时命令才执行完毕。

[subscriber sendCompleted];

return nil;

}];

}];

// 强引用命令，不要被销毁，否则接收不到数据

\_conmmand = command;

// 3.订阅RACCommand中的信号

[command.executionSignals subscribeNext:^(id x) {

[x subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

}];

// RAC高级用法

// switchToLatest:用于signal of signals，获取signal of signals发出的最新信号,也就是可以直接拿到RACCommand中的信号

[command.executionSignals.switchToLatest subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

// 4.监听命令是否执行完毕,默认会来一次(会先检测到之前的状态是执行完成)，可以直接跳过，skip表示跳过第一次信号。

[[command.executing skip:1] subscribeNext:^(id x) {

if ([x boolValue] == YES) {

// 正在执行

NSLog(@"正在执行");

}else{

// 执行完成

NSLog(@"执行完成");

}

}];

// 5.执行命令

[self.conmmand execute:@1];

4.8 **RACMulticastConnection**:用于当一个信号，被多次订阅时，为了保证创建信号时，避免多次调用创建信号中的block，造成副作用，可以使用这个类处理。

使用注意:RACMulticastConnection通过RACSignal的-publish或者-muticast:方法创建.

RACMulticastConnection简单使用:

// RACMulticastConnection使用步骤:

// 1.创建信号 + (RACSignal \*)createSignal:(RACDisposable \* (^)(id<RACSubscriber> subscriber))didSubscribe

// 2.创建连接 RACMulticastConnection \*connect = [signal publish];

// 3.订阅信号,注意：订阅的不在是之前的信号，而是连接的信号。 [connect.signal subscribeNext:nextBlock]

// 4.连接 [connect connect]

// RACMulticastConnection底层原理:

// 1.创建connect，connect.sourceSignal -> RACSignal(原始信号) connect.signal -> RACSubject

// 2.订阅connect.signal，会调用RACSubject的subscribeNext，创建订阅者，而且把订阅者保存起来，不会执行block。

// 3.[connect connect]内部会订阅RACSignal(原始信号)，并且订阅者是RACSubject

// 3.1.订阅原始信号，就会调用原始信号中的didSubscribe

// 3.2 didSubscribe，拿到订阅者调用sendNext，其实是调用RACSubject的sendNext

// 4.RACSubject的sendNext,会遍历RACSubject所有订阅者发送信号。

// 4.1 因为刚刚第二步，都是在订阅RACSubject，因此会拿到第二步所有的订阅者，调用他们的nextBlock

// 需求：假设在一个信号中发送请求，每次订阅一次都会发送请求，这样就会导致多次请求。

// 解决：使用RACMulticastConnection就能解决.

// 1.创建请求信号

RACSignal \*signal = [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

NSLog(@"发送请求");

return nil;

}];

// 2.订阅信号

[signal subscribeNext:^(id x) {

NSLog(@"接收数据");

}];

// 2.订阅信号

[signal subscribeNext:^(id x) {

NSLog(@"接收数据");

}];

// 3.运行结果，会执行两遍发送请求，也就是每次订阅都会发送一次请求

// RACMulticastConnection:解决重复请求问题

// 1.创建信号

RACSignal \*signal = [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

NSLog(@"发送请求");

[subscriber sendNext:@1];

return nil;

}];

// 2.创建连接

RACMulticastConnection \*connect = [signal publish];

// 3.订阅信号，

// 注意：订阅信号，也不能激活信号，只是保存订阅者到数组，必须通过连接,当调用连接，就会一次性调用所有订阅者的sendNext:

[connect.signal subscribeNext:^(id x) {

NSLog(@"订阅者一信号");

}];

[connect.signal subscribeNext:^(id x) {

NSLog(@"订阅者二信号");

}];

// 4.连接,激活信号

[connect connect];

4.9 **RACScheduler**: 信号调度器，是一个线性执行队列，rac中的信号可以在 RACScheduler 上执行任务、发送结果，底层用GCD封装的。

4.9.1 包含6个实例化方法：都是用来生成一个queue的

//是一个会立即执行传入的代码块的调度器

+ (RACScheduler \*)immediateScheduler;

//immediateScheduler底层实现就是生成了一个RACImmediateScheduler的单例。

RACImmediateScheduler 是继承自RACScheduler

//获得主线程调度器

+ (RACScheduler \*)mainThreadScheduler;

//获取当前线程调度器

+ (RACScheduler \*)currentScheduler;

//以下3个方法实质都是是创建了一个 Global Dispatch Queue，对应的属于Concurrent Dispatch Queue。

+(RACScheduler\*)schedulerWithPriority:(RACSchedulerPriority)priority name:(NSString \*)name;//方法可以指定线程的优先级和名字

+ (RACScheduler \*)schedulerWithPriority:(RACSchedulerPriority)priority; //方法只能执行优先级，名字为默认的

+ (RACScheduler \*)scheduler;//方法创建出来的queue的优先级是默认的，名字也是默认的

4.9.2 几个常用的用于调度、执行任务的方法:

//会立刻执行传入的 block

- (RACDisposable \*)schedule:(**void** (^)(**void**))block;

//会将当前线程休眠到指定时间后执行 block

- (RACDisposable \*)after:(NSDate \*)date schedule:(**void** (^)(**void**))block;

//当前线程休眠date时间之后执行，然后每隔interval时间重复执行, leeway这个参数是为dispatch source指定一个期望的定时器事件精度，让系统能够灵活地管理并唤醒内核。例如系统可以使用leeway值来提前或延迟触发定时器，使其更好地与其它系统事件结合。创建自己的定时器时，应该尽量指定一个leeway值。不过就算指定leeway值为0，也不能完完全全期望定时器能够按照精确的纳秒来触发事件

- (RACDisposable \*)after:(NSDate \*)date repeatingEvery:(NSTimeInterval)interval withLeeway:(NSTimeInterval)leeway schedule:(**void** (^)(**void**))block;

4.9.3 RACScheduler的使用举例：

//1.线程的切换：

**deliverOn：**在设置的调度中发送信号值,但操作封包依然在原来的调度里进行

//创建信号

RACSignal \* signal = [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

NSLog(@"sendTestSignal%@",[NSThread currentThread]);

[subscriber sendNext:@1];

[subscriber sendCompleted];

return [RACDisposable disposableWithBlock:^{

}];

}];

//订阅信号

//要想放在主线程执行只要将[RACScheduler scheduler]更换为[RACScheduler mainThreadScheduler]

[[self.testSignal deliverOn:[RACScheduler scheduler]] subscribeNext:^(id x) {

NSLog(@"receiveSignal%@",[NSThread currentThread]);

}];

打印结果如下

2016-09-02 09:48:59.697 Signal processing[1686:22894] sendTestSignal<NSThread: 0x7fb373c0bb80>{number = 1, name = main}

2016-09-02 09:48:59.697 Signal processing[1686:24680] receiveSignal<NSThread: 0x7fb373e07070>{number = 3, name = (null)}

**subscribeOn**：在设置的调度中发送信号和执行封包里的操作

[[[RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

NSLog(@"sendSignal%@",[NSThread currentThread]);

[subscriber sendNext:@1];

return [RACDisposable disposableWithBlock:^{

}];

}] subscribeOn:[RACScheduler scheduler]] subscribeNext:^(id x) {

NSLog(@"receiveSignal%@",[NSThread currentThread]);

}];

打印结果如下

2016-09-02 09:54:47.819 Signal processing[1778:54504] sendSignal<NSThread: 0x7fde7adb4e00>{number = 2, name = (null)}

2016-09-02 09:54:47.819 Signal processing[1778:54504] receiveSignal<NSThread: 0x7fde7adb4e00>{number = 2, name = (null)}

2.与时间相关的操作会用到

timeout：超时，可以让一个信号在一定的时间后，自动报错。（我们项目中网络请求超时问题没有处理）

RACSignal \*signal = [[RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

return nil;

}] timeout:1 onScheduler:[RACScheduler currentScheduler]];

[signal subscribeNext:^(id x) {

NSLog(@"%@",x);

} error:^(NSError \*error) {

// 1秒后会自动调用

NSLog(@"%@",error);

}];

interval 定时：每隔一段时间发出信号

[[RACSignal interval:1 onScheduler:[RACScheduler currentScheduler]] subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

在RACSignal+Operations很作操作方法中都会调用RACScheduler的实例化方法，我们外部直接去调用的比较少，大多都被封装进去了，比如：

- (RACSignal \*)delay:(NSTimeInterval)interval；

+ (RACSignal \*)interval:(NSTimeInterval)interval onScheduler:(RACScheduler \*)scheduler

- (RACSignal \*)timeout:(NSTimeInterval)interval onScheduler:(RACScheduler \*)scheduler { }

- (RACSignal \*)deliverOn:(RACScheduler \*)scheduler { }

- (RACSignal \*)subscribeOn:(RACScheduler \*)scheduler { }

RACScheduler底层原理参考：

<http://www.open-open.com/lib/view/open1487381068466.html>

<http://www.jianshu.com/p/980ffdf3ed8c>

**5.ReactiveCocoa常见操作方法介绍**

5.1 ReactiveCocoa操作须知:

所有的信号（RACSignal）都可以进行操作处理，因为所有操作方法都定义在RACStream.h中，因此只要继承RACStream就有了操作处理方法

5.2ReactiveCocoa操作思想:

* 运用的是Hook（钩子）思想，Hook是一种用于改变API(应用程序编程接口：方法)执行结果的技术.
* Hook用处：截获API调用的技术。

Hook原理：在每次调用一个API返回结果之前，先执行你自己的方法，改变结果的输出。

5.3ReactiveCocoa核心方法**bind**

// 假设想监听文本框的内容，并且在每次输出结果的时候，都在文本框的内容拼接一段文字“输出：”

// 方式一:在返回结果后，拼接。

[\_textField.rac\_textSignal subscribeNext:^(id x) {

NSLog(@"输出:%@",x);

}];

// 方式二:在返回结果前，拼接，使用RAC中bind方法做处理。

// bind方法参数:需要传入一个返回值是RACStreamBindBlock的block参数

// RACStreamBindBlock是一个block的类型，返回值是信号，参数（value,stop），因此参数的block返回值也是一个block。

// RACStreamBindBlock:

// 参数一(value):表示接收到信号的原始值，还没做处理

// 参数二(\*stop):用来控制绑定Block，如果\*stop = yes,那么就会结束绑定。

// 返回值：信号，做好处理，在通过这个信号返回出去，一般使用RACReturnSignal,需要手动导入头文件RACReturnSignal.h。

// bind方法使用步骤:

// 1.传入一个返回值RACStreamBindBlock的block。

// 2.描述一个RACStreamBindBlock类型的bindBlock作为block的返回值。

// 3.描述一个返回结果的信号，作为bindBlock的返回值。

// 注意：在bindBlock中做信号结果的处理。

// 底层实现:

// 1.源信号调用bind,会重新创建一个绑定信号。

// 2.当绑定信号被订阅，就会调用绑定信号中的didSubscribe，生成一个bindingBlock。

// 3.当源信号有内容发出，就会把内容传递到bindingBlock处理，调用bindingBlock(value,stop)

// 4.调用bindingBlock(value,stop)，会返回一个内容处理完成的信号（RACReturnSignal）。

// 5.订阅RACReturnSignal，就会拿到绑定信号的订阅者，把处理完成的信号内容发送出来。

// 注意:不同订阅者，保存不同的nextBlock，看源码的时候，一定要看清楚订阅者是哪个。

// 这里需要手动导入#import <ReactiveCocoa/RACReturnSignal.h>，才能使用RACReturnSignal。

[[\_textField.rac\_textSignal bind:^RACStreamBindBlock{

// 什么时候调用:

// block作用:表示绑定了一个信号.

return ^RACStream \*(id value, BOOL \*stop){

// 什么时候调用block:当信号有新的值发出，就会来到这个block。

// block作用:做返回值的处理

// 做好处理，此处可以实现一些比较猥琐的操作，然后通过信号返回出去.

return [RACReturnSignal return:[NSString stringWithFormat:@"输出:%@",value]];

};

}] subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

5.4 ReactiveCocoa操作方法之映射(**flattenMap,Map**)

* 1. flattenMap，Map用于把源信号内容映射成新的内容。

flattenMap简单使用

// 监听文本框的内容改变，把结构重新映射成一个新值.

// flattenMap作用:把源信号的内容映射成一个新的信号，信号可以是任意类型。

// flattenMap使用步骤:

// 1.传入一个block，block类型是返回值RACStream，参数value

// 2.参数value就是源信号的内容，拿到源信号的内容做处理

// 3.包装成RACReturnSignal信号，返回出去。

// flattenMap底层实现:

// 0.flattenMap内部调用bind方法实现的,flattenMap中block的返回值，会作为bind中bindBlock的返回值。

// 1.当订阅绑定信号，就会生成bindBlock。

// 2.当源信号发送内容，就会调用bindBlock(value, \*stop)

// 3.调用bindBlock，内部就会调用flattenMap的block，flattenMap的block作用：就是把处理好的数据包装成信号。

// 4.返回的信号最终会作为bindBlock中的返回信号，当做bindBlock的返回信号。

// 5.订阅bindBlock的返回信号，就会拿到绑定信号的订阅者，把处理完成的信号内容发送出来。

[[\_textField.rac\_textSignal flattenMap:^RACStream \*(id value) {

// block什么时候 : 源信号发出的时候，就会调用这个block。

// block作用 : 改变源信号的内容。

// 返回值：绑定信号的内容.

return [RACReturnSignal return:[NSString stringWithFormat:@"输出:%@",value]];

}] subscribeNext:^(id x) {

// 订阅绑定信号，每当源信号发送内容，做完处理，就会调用这个block。

NSLog(@"%@",x);

}];

Map简单使用:

// 监听文本框的内容改变，把结构重新映射成一个新值.

// Map作用:把源信号的值映射成一个新的值

// Map使用步骤:

// 1.传入一个block,类型是返回对象，参数是value

// 2.value就是源信号的内容，直接拿到源信号的内容做处理

// 3.把处理好的内容，直接返回就好了，不用包装成信号，返回的值，就是映射的值。

// Map底层实现:

// 0.Map底层其实是调用flatternMap,Map中block中的返回的值会作为flatternMap中block中的值。

// 1.当订阅绑定信号，就会生成bindBlock。

// 3.当源信号发送内容，就会调用bindBlock(value, \*stop)

// 4.调用bindBlock，内部就会调用flattenMap的block

// 5.flattenMap的block内部会调用Map中的block，把Map中的block返回的内容包装成返回的信号。

// 5.返回的信号最终会作为bindBlock中的返回信号，当做bindBlock的返回信号。

// 6.订阅bindBlock的返回信号，就会拿到绑定信号的订阅者，把处理完成的信号内容发送出来。

[[\_textField.rac\_textSignal map:^id(id value) {

// 当源信号发出，就会调用这个block，修改源信号的内容

// 返回值：就是处理完源信号的内容。

return [NSString stringWithFormat:@"输出:%@",value];

}] subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

* FlatternMap和Map的区别
  + 1.FlatternMap中的Block返回信号。
  + 2.Map中的Block返回对象。
  + 3.开发中，如果信号发出的值不是信号，映射一般使用Map
  + 4.开发中，如果信号发出的值是信号，映射一般使用FlatternMap。
* 总结：signalOfsignals用FlatternMap。

// 创建信号中的信号

RACSubject \*signalOfsignals = [RACSubject subject];

RACSubject \*signal = [RACSubject subject];

[[signalOfsignals flattenMap:^RACStream \*(id value) {

// 当signalOfsignals的signals发出信号才会调用

return value;

}] subscribeNext:^(id x) {

// 只有signalOfsignals的signal发出信号才会调用，因为内部订阅了bindBlock中返回的信号，也就是flattenMap返回的信号。

// 也就是flattenMap返回的信号发出内容，才会调用。

NSLog(@"%@aaa",x);

}];

// 信号的信号发送信号

[signalOfsignals sendNext:signal];

// 信号发送内容

[signal sendNext:@1];

5.5 ReactiveCocoa操作方法之组合。

**concat**:按一定顺序拼接信号，当多个信号发出的时候，有顺序的接收信号。

RACSignal \*signalA = [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

[subscriber sendNext:@1];

[subscriber sendCompleted];

return nil;

}];

RACSignal \*signalB = [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

[subscriber sendNext:@2];

return nil;

}];

// 把signalA拼接到signalB后，signalA发送完成，signalB才会被激活。

RACSignal \*concatSignal = [signalA concat:signalB];

// 以后只需要面对拼接信号开发。

// 订阅拼接的信号，不需要单独订阅signalA，signalB

// 内部会自动订阅。

// 注意：第一个信号必须发送完成，第二个信号才会被激活

[concatSignal subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

// concat底层实现:

// 1.当拼接信号被订阅，就会调用拼接信号的didSubscribe

// 2.didSubscribe中，会先订阅第一个源信号（signalA）

// 3.会执行第一个源信号（signalA）的didSubscribe

// 4.第一个源信号（signalA）didSubscribe中发送值，就会调用第一个源信号（signalA）订阅者的nextBlock,通过拼接信号的订阅者把值发送出来.

// 5.第一个源信号（signalA）didSubscribe中发送完成，就会调用第一个源信号（signalA）订阅者的completedBlock,订阅第二个源信号（signalB）这时候才激活（signalB）。

// 6.订阅第二个源信号（signalB）,执行第二个源信号（signalB）的didSubscribe

// 7.第二个源信号（signalA）didSubscribe中发送值,就会通过拼接信号的订阅者把值发送出来.

**then**:用于连接两个信号，当第一个信号完成，才会连接then返回的信号。

* // 注意使用then，之前信号的值会被忽略掉.
* // 底层实现：1、先过滤掉之前的信号发出的值。2.使用concat连接then返回的信号

[[[RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

[subscriber sendNext:@1];

[subscriber sendCompleted];

return nil;

}] then:^RACSignal \*{

return [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

[subscriber sendNext:@2];

return nil;

}];

}] subscribeNext:^(id x) {

// 只能接收到第二个信号的值，也就是then返回信号的值

NSLog(@"%@",x);

}];

**merge**:把多个信号合并为一个信号，任何一个信号有新值的时候就会调用

注意区别concat：concat的信号有先后顺序，必须要前面的信号先发出后面的信号才会被激活

//创建多个信号

RACSignal \*= = [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

[subscriber sendNext:@1];

return nil;

}];

RACSignal \*signalB = [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

[subscriber sendNext:@2];

return nil;

}];

// 合并信号,任何一个信号发送数据，都能监听到.

RACSignal \*mergeSignal = [signalA merge:signalB];

[mergeSignal subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

// 底层实现：

// 1.合并信号被订阅的时候，就会遍历所有信号，并且发出这些信号。

// 2.每发出一个信号，这个信号就会被订阅

// 3.也就是合并信号一被订阅，就会订阅里面所有的信号。

// 4.只要有一个信号被发出就会被监听。

**zipWith**:把两个信号压缩成一个信号，只有当两个信号同时发出信号内容时，并且把两个信号的内容合并成一个元组，才会触发压缩流的next事件。

RACSignal \*signalA = [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

[subscriber sendNext:@1];

return nil;

}];

RACSignal \*signalB = [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

[subscriber sendNext:@2];

return nil;

}];

// 压缩信号A，信号B

RACSignal \*zipSignal = [signalA zipWith:signalB];

[zipSignal subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

// 底层实现:

// 1.定义压缩信号，内部就会自动订阅signalA，signalB

// 2.每当signalA或者signalB发出信号，就会判断signalA，signalB有没有发出个信号，有就会把最近发出的信号都包装成元组发出。

**combineLatest**:将多个信号合并起来，并且拿到各个信号的最新的值,必须每个合并的signal至少都有过一次sendNext，才会触发合并的信号。

RACSignal \*signalA = [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

[subscriber sendNext:@1];

return nil;

}];

RACSignal \*signalB = [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

[subscriber sendNext:@2];

return nil;

}];

RACSignal \*combineSignal = [signalA combineLatestWith:signalB];

[combineSignal subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

* // 底层实现：
* // 1.当组合信号被订阅，内部会自动订阅signalA，signalB,必须两个信号都发出内容，才会被触发。
* // 2.并且把两个信号组合成元组发出。

**Zip和****combineLatest区别：**

Zip的合并行为是按顺序取出各个信号，然后合并成元组后信号后发出。比如有信号A和信号B，合并信号为zipSingle,zipSingle发出的条件必须是A和B都有对应顺序下的最新值发出。比如A发了信号1和信号2,而B只发了信号3，那么此时组合信号只会触发一次，因为A虽说发了2次信号，但是B只发了一次，第一次的信号从A和B中都能取到，所以会有组合信号发出，第二次B信号取不到，故第二次的组合信号不成立，不会触发组合信号，只有B再发送一次信号时能能触发第二次组合信号。

CombineLatest的合并行为是，只要A和B两个子信号都触发过一次以上，就会触发该组合信号，并且后续只要A或者B中的一个信号有新信号发出都会触发该组合信号，并且该组合信号合并的规则是一直取A或者B中当前最新的值合并成元组。

实际开发中，我们下拉刷新需要等到所有接口返回再刷新UI的时候应该用zip而不是combinLatest.

**reduce**聚合:用于信号发出的内容是元组，把信号发出元组的值聚合成一个值

RACSignal \*signalA = [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

[subscriber sendNext:@1];

return nil;

}];

RACSignal \*signalB = [RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

[subscriber sendNext:@2];

return nil;

}];

// 常见的用法，（先组合在聚合）。combineLatest:(id<NSFastEnumeration>)signals reduce:(id (^)())reduceBlock

// reduce中的block简介:

// reduceblcok中的参数，有多少信号组合，reduceblcok就有多少参数，每个参数就是之前信号发出的内容

// reduceblcok的返回值：聚合信号之后的内容。

RACSignal \*reduceSignal = [RACSignal combineLatest:@[signalA,signalB] reduce:^id(NSNumber \*num1 ,NSNumber \*num2){

return [NSString stringWithFormat:@"%@ %@",num1,num2];

}];

[reduceSignal subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

// 底层实现:

// 1.订阅聚合信号，每次有内容发出，就会执行reduceblcok，把信号内容转换成reduceblcok返回的值。

5.6 ReactiveCocoa操作方法之过滤。

* **filter**:过滤信号，使用它可以获取满足条件的信号.

// 过滤:

// 每次信号发出，会先执行过滤条件判断.

[\_textField.rac\_textSignal filter:^BOOL(NSString \*value) {

return value.length > 3;

}];

* **ignore**:忽略完某些值的信号.

// 内部调用filter过滤，忽略掉ignore的值

[[\_textField.rac\_textSignal ignore:@"1"] subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

* **distinctUntilChanged**:当上一次的值和当前的值有明显的变化就会发出信号，否则会被忽略掉。

// 过滤，当上一次和当前的值不一样，就会发出内容。

// 在开发中，刷新UI经常使用，只有两次数据不一样才需要刷新

[[\_textField.rac\_textSignal distinctUntilChanged] subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

* **take**:从开始一共取N次的信号

// 1、创建信号

RACSubject \*signal = [RACSubject subject];

// 2、处理信号，订阅信号

[[signal take:1] subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

// 3.发送信号

[signal sendNext:@1];

[signal sendNext:@2];

* **takeLast**:取最后N次的信号,前提条件，订阅者必须调用完成，因为只有完成，就知道总共有多少信号.

// 1、创建信号

RACSubject \*signal = [RACSubject subject];

// 2、处理信号，订阅信号

[[signal takeLast:1] subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

// 3.发送信号

[signal sendNext:@1];

[signal sendNext:@2];

[signal sendCompleted];//必须调完成

* **takeUntil**:(RACSignal \*):获取信号直到某个信号执行完成

// 监听文本框的改变直到当前对象被销毁

[\_textField.rac\_textSignal takeUntil:self.rac\_willDeallocSignal];

* **skip**:(NSUInteger):跳过几个信号,不接受。

// 表示输入第一次，不会被监听到，跳过第一次发出的信号

[[\_textField.rac\_textSignal skip:1] subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

* **switchToLatest**:用于signalOfSignals（信号的信号），有时候信号也会发出信号，会在signalOfSignals中，获取signalOfSignals发送的最新信号。

RACSubject \*signalOfSignals = [RACSubject subject];

RACSubject \*signal = [RACSubject subject];

// 获取信号中信号最近发出信号，订阅最近发出的信号。

// 注意switchToLatest：只能用于信号中的信号

[signalOfSignals.switchToLatest subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

[signalOfSignals sendNext:signal];

[signal sendNext:@1];

5.7 ReactiveCocoa操作方法之秩序。

* **doNext**: 执行Next之前，会先执行这个Block
* **doCompleted**: 执行sendCompleted之前，会先执行这个Block

[[[[RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

[subscriber sendNext:@1];

[subscriber sendCompleted];

return nil;

}] doNext:^(id x) {

// 执行[subscriber sendNext:@1];之前会调用这个Block

NSLog(@"doNext");;

}] doCompleted:^{

// 执行[subscriber sendCompleted];之前会调用这个Block

NSLog(@"doCompleted");;

}] subscribeNext:^(id x) {

NSLog(@"%@",x);

}];

5.8 ReactiveCocoa操作方法之线程。

**deliverOn**: 内容传递切换到制定线程中，副作用在原来线程中,把在创建信号时block中的代码称之为副作用。

**subscribeOn**: 内容传递和副作用都会切换到制定线程中。

上文**RACScheduler**均有代码演示。

5.9 ReactiveCocoa操作方法之时间。

**timeout**：超时，可以让一个信号在一定的时间后，自动报错。

**interval** 定时：每隔一段时间发出信号

**delay** 延迟发送next。

上文均有代码演示。

5.10 ReactiveCocoa操作方法之重复。

**retry**重试 ：只要失败，就会重新执行创建信号中的block,直到成功.

\_\_block int i = 0;

[[[RACSignal createSignal:^RACDisposable \*(id<RACSubscriber> subscriber) {

if (i == 10) {

[subscriber sendNext:@1];

}else{

NSLog(@"接收到错误");

[subscriber sendError:nil];

}

i++;

return nil;

}] retry] subscribeNext:^(id x) {

NSLog(@"%@",x);

} error:^(NSError \*error) {

}];

**replay**重放：当一个信号被多次订阅,反复播放内容（类似的还有**replayLast，replayLazily**）

具体用法比较：见文档。

**throttle**节流:当某个信号发送比较频繁时，可以使用节流，在某一段时间不发送信号内容，过了一段时间获取信号的最新内容发出。

需求场景：

在我们做搜索框的时候，有时候需求的时实时搜索，即用户每每输入字符，view都要求展现搜索结果。这时如果用户搜索的字符串较长，那么由于网络请求的延时可能造成UI显示错误，并且多次不必要的请求还会加大服务器的压力，这显然是不合理的，此时我们就需要用到节流。

[[[self.textFild rac\_textSignal] throttle:1] subscribeNext:^(id x) {

NSLog(@"%@", x);

}];

主要技术参考资料：

<http://www.jianshu.com/p/87ef6720a096>

<http://www.jianshu.com/p/e10e5ca413b7>