ReactiveCocoa简介

简介

ReactiveCocoa（RAC）是一个将**函数响应式编程**范例带入Objective-C的开源库
由Josh Abernathy和Justin Spahr-Summers在对GitHub for Mac的开发过程中建立。
Mattt Thompson(AFN作者)这样评论RAC:

- 打破了苹果API排他性的盾牌。
- 一个为Objective-C构建新纪元的开源项目

版本

ReactiveCocoa社区也非常活跃，最新版为ReactiveCocoa 5.0.0-alpha.3，5.0.0-alpha.4在开发中。

版本	Objective-C支持	Swift支持	备注
RAC 2.5	Y	N	迭代周期长，OC最稳定版本
RAC 3.x	Y	1.2	开支全面支持Swift
RAC 4.x	Y	2.x	
RAC 5.x	Y	3.0	

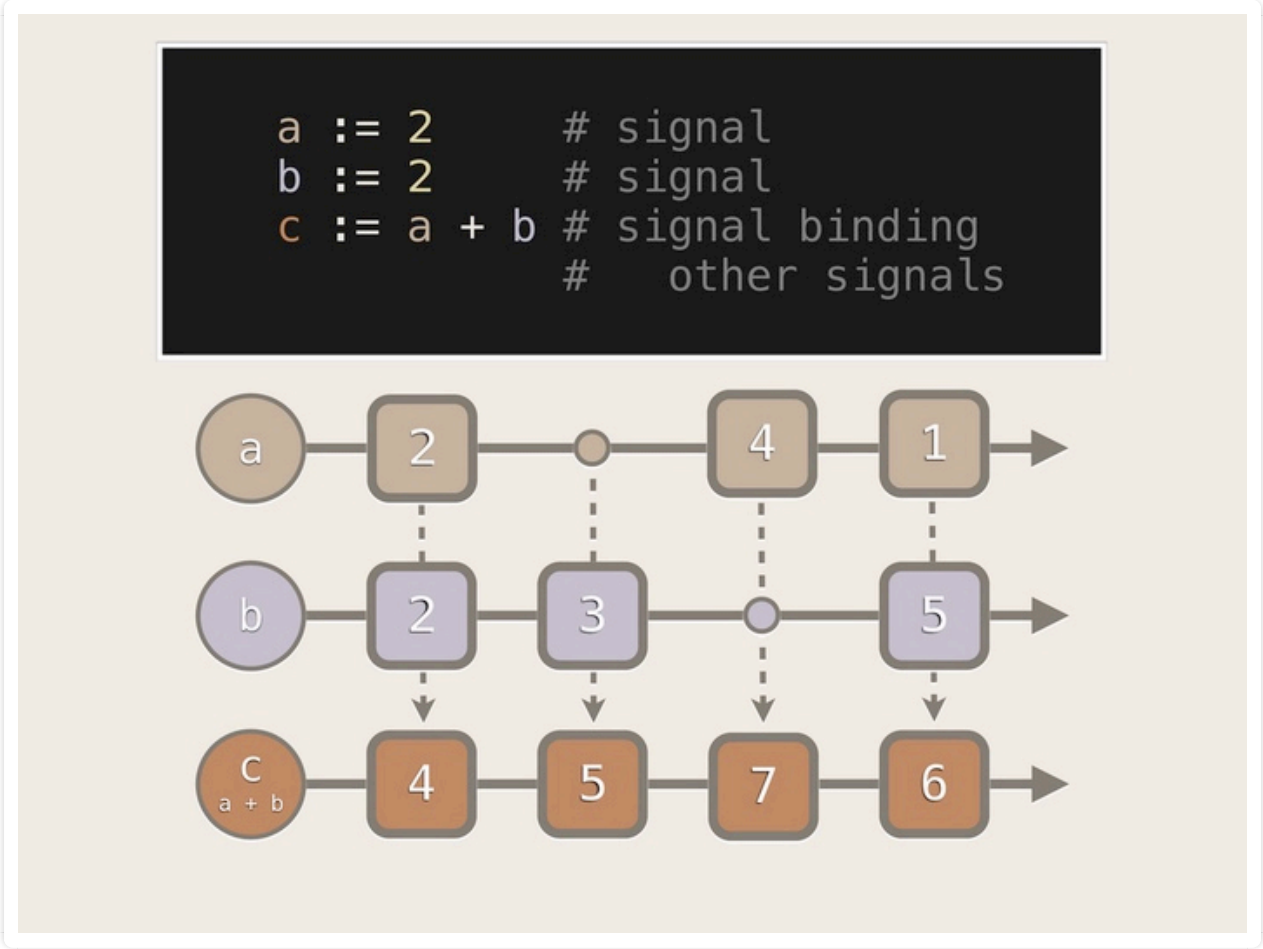
其他语言响应式框架：RxSwift、RxJava、Rx.Net、RxJS

编程思想

响应式编程

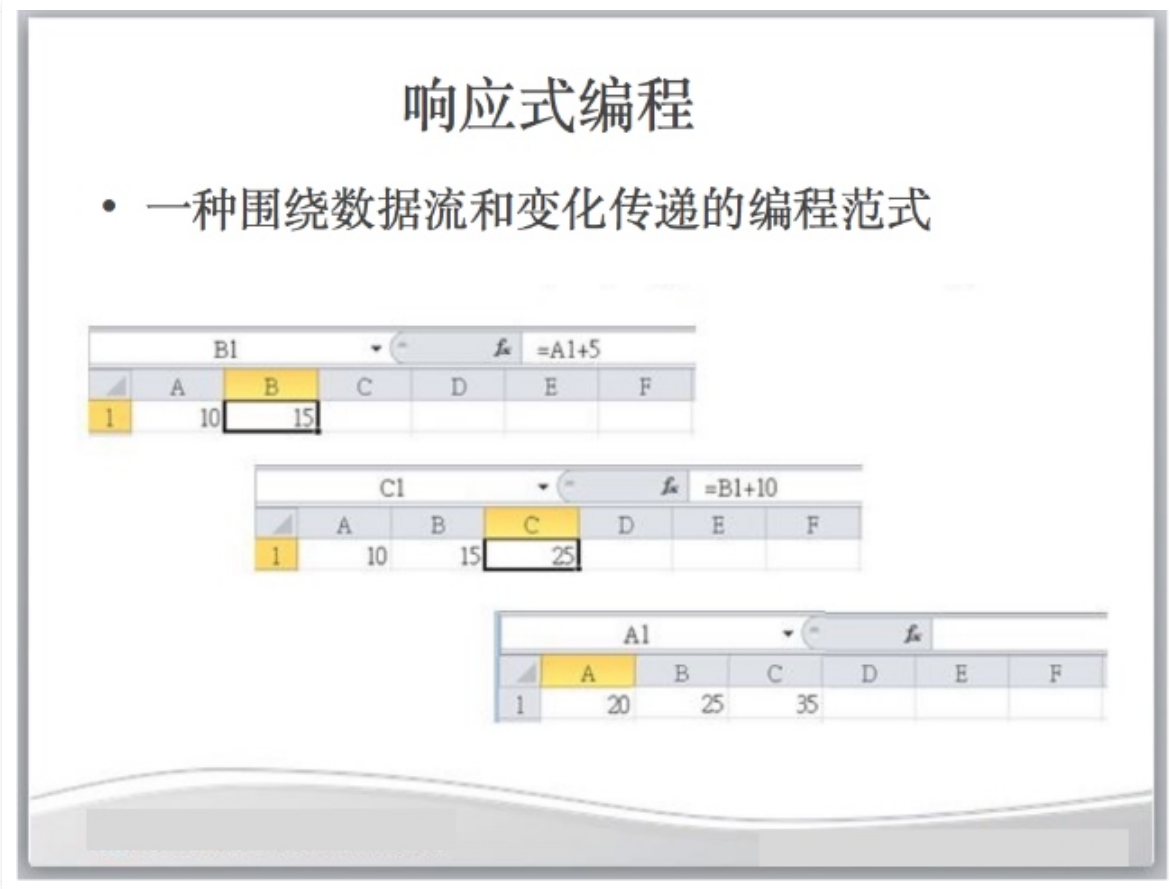
在计算机中，**响应式编程是一种面向数据流和变化传播的编程范式**。这意味着可以在编程语言中很方便地表达静态或动态的数据流，而相关的计算模型会自动将变化的值通过数据流进行传播。

例如，在命令式编程环境中，a=b+c表示将表达式的结果赋给a，而之后改变b或c的值不会影响a。但在响应式编程中，a的值会随着b或c的更新而更新。



响应式编程应用：

- Excel



- Autolayout

响应式编程特点

- 输入被视为"行为"，或者说一个随时间而变化的事件流
- 连续的、随时间而变化的值
- 按时间排序的离散事件序列

函数式编程

- 函数是”第一等公民”
- 闭包和高阶函数
- 不改变状态
- 递归
- 只用”表达式”，不用”语句”，没有副作用

为什么用RAC

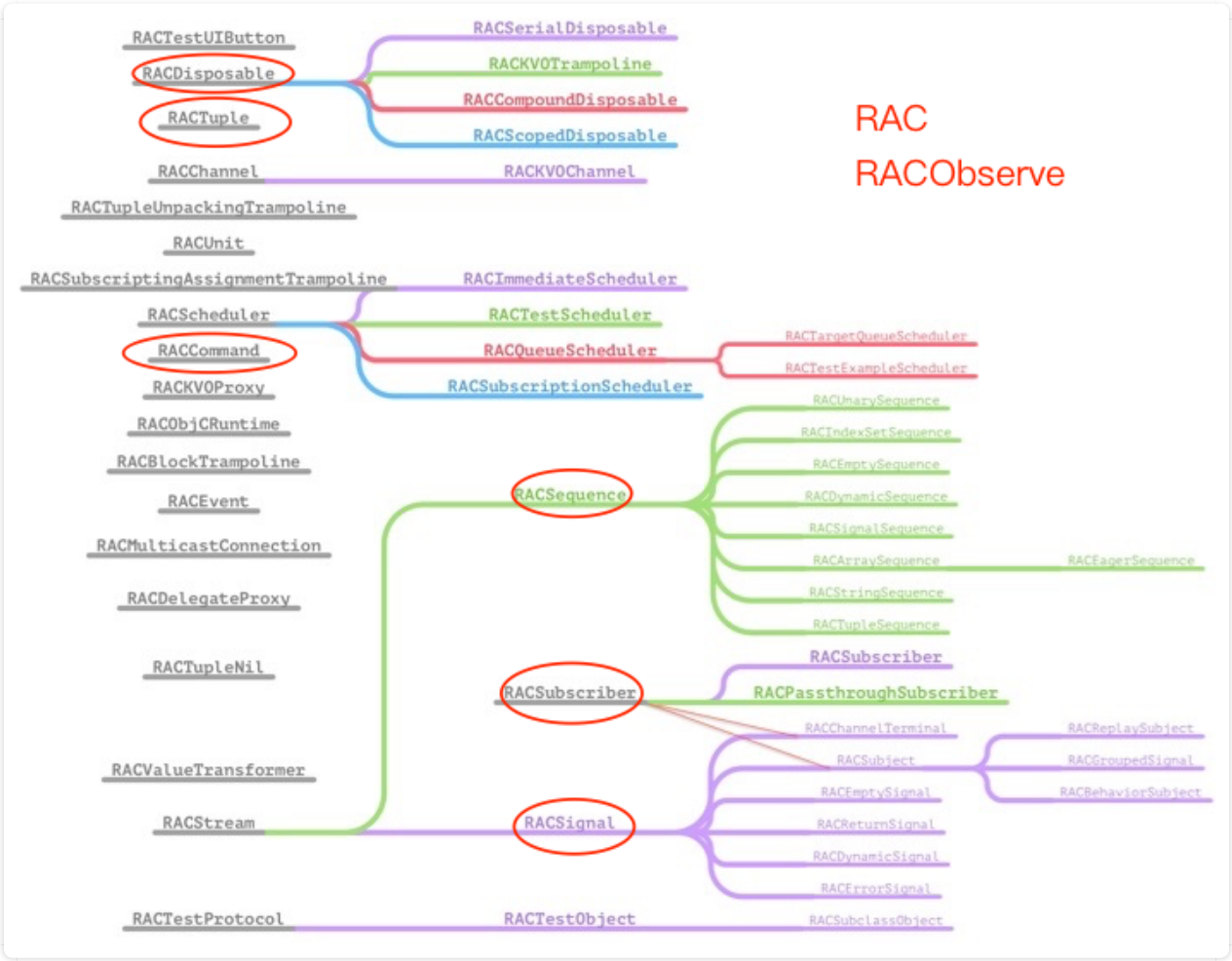
- 开发过程中，状态以及状态之间依赖过多,RAC更加有效率地处理事件流，而无需显式去管理状态。在OO或者过程式编程中，状态变化是最难跟踪，最头痛的事。
- 减少变量的使用，由于它跟踪状态和值的变化，因此不需要再申明变量不断地观察状态和更新值。
- 提供统一的消息传递机制，将OC中的通知，action，KVO以及其它所有UIControl事件的变化都进行监控，当变化发生时，就会传递事件和值
- 当值随着事件变换时，可以使用map，filter，reduce，zip，merge等流运算符便利地对值进行变换操作，实现复杂功能。
- 结合MVVM，更好的实现UI与业务逻辑分层，减少控制器代码，提高程序可测性

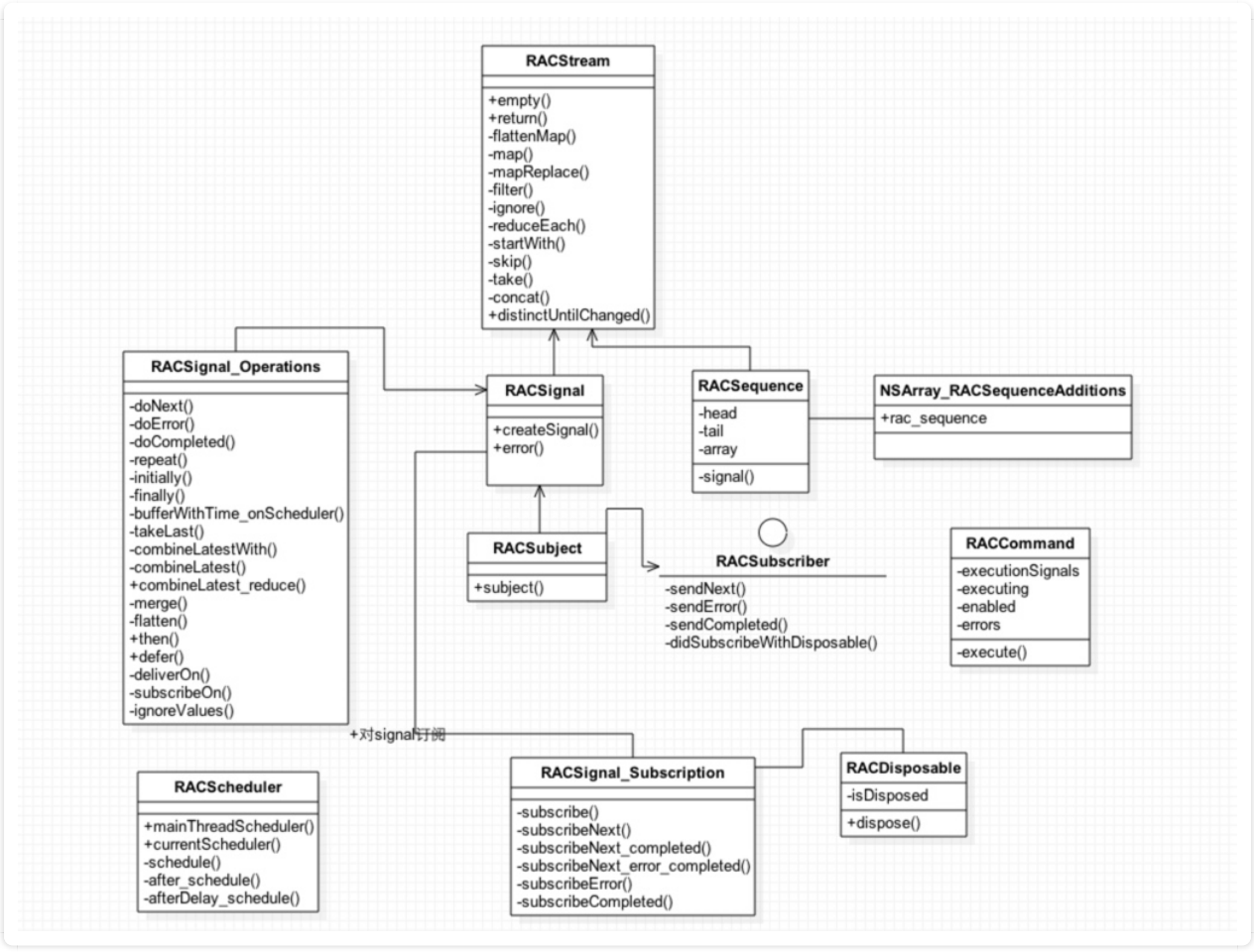
ReactiveCocoa框架概览

概览

ReactiveCocoa 是一个非常复杂的框架，主要有以下组件组成：

- 信号源：RACStream 及其子类；
- 订阅者：RACSubscriber 的实现类及其子类。
- 调度器：RACScheduler 及其子类
- 清洁工：RACDisposable 及其子类

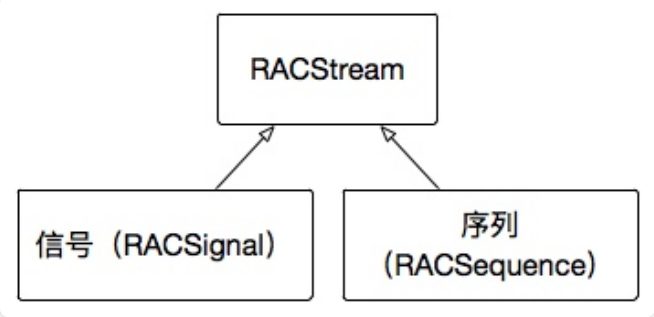

RAC
RACObserve
RACTestUIButton
RACDisposable
RACTuple
RACSerialDisposable
RACKVOTrampoline
RACCompoundDisposable
RACScopedDisposable
RACChannel
RACKVOChannel
RACTupleUnpackingTrampoline
RACUnit
RACSubscriptingAssignmentTrampoline
RACScheduler
RACCommand
RACKVOProxy
RACObjCRuntime
RACBlockTrampoline
RACEvent
RACMulticastConnection
RACDelegateProxy
RACTupleNil
RACValueTransformer
RACStream
RACImmediateScheduler
RACTestScheduler
RACQueueScheduler
RACSubscriptionScheduler
RACTargetQueueScheduler
RACTestExampleScheduler
RACSequence
RACUnarySequence
RACIndexSetSequence
RACEmptySequence
RACDynamicSequence
RACSignalSequence
RACArraySequence
RACEagerSequence
RACStringSequence
RACTupleSequence
RACSubscriber
RACSubscriber
RACPassthroughSubscriber
RACChannelTerminal
RACSubject
RACReplaySubject
RACGroupedSignal
RACBehaviorSubject
RACEmptySignal
RACSignal
RACReturnSignal
RACDynamicSignal
RACErrorSignal
RACTestProtocol
RACTestObject
RACSubclassObject


RACStream
+empty()
+return()
-flattenMap()
-map()
-mapReplace()
-filter()
-ignore()
-reduceEach()
-startWith()
-skip()
-take()
-concat()
+distinctUntilChanged()
RACSignal_Operations
-doNext()
-doError()
-doCompleted()
-repeat()
-initially()
-finally()
-bufferWithTime_onScheduler()
-takeLast()
-combineLatestWith()
-combineLatest()
+combineLatest_reduce()
-merge()
-flatten()
+then()
+defer()
-deliverOn()
-subscribeOn()
-ignoreValues()
RACSignal
+createSignal()
+error()
RACSequence
-head
-tail
-array
-signal()
NSArray_RACSequenceAdditions
+rac_sequence
RACSubject
+subject()
RACSubscriber
-sendNext()
-sendError()
-sendCompleted()
-didSubscribeWithDisposable()
RACCommand
-executionSignals
-executing
-enabled
-errors
-execute()
+对signal订阅
RACScheduler
+mainThreadScheduler()
+currentScheduler()
-schedule()
-after_schedule()
-afterDelay_schedule()
RACSignal_Subscription
-subscribe()
-subscribeNext()
-subscribeNext_completed()
-subscribeNext_error_completed()
-subscribeError()
-subscribeCompleted()
RACDisposable
-isDisposed
+dispose()

核心对象

RACStream

RACStream抽象类代表了一个流，是由一些对象值所组成的任意序列。
值可能马上或者在将来某个时刻变得可用，但是每个值必须按顺序检索。在检索流中的第二个值之前，你必须对流中的第一个值进行处理。
流是单子(monads)。
RACStream本身作用并不大。大部分流其实都被当做了信号或者序列。



RACSignal

一个RACSignal类对象代表了一个信号，是一个推动式（ push-driven ）的流。
信号通常都代表了将来会被送达的数据。比如一个方法被调用或者收到了数据，值在信号中被发送(sent)，信号将会把他们‘推给’任何订阅者。用户必须订阅一个信号来获取它里面的值。
信号会发送三种不同的事件给它的订阅者:

- next将会从流中获得一个新的值。RACStream只会操作此种类型的事件。和Cocoa中的集合类型不同，信号中包含一个nil也是合法的。
- error事件表明一个错误在信号发出之前产生。这个时间可能会包含一个NSError对象，表明发生了什么错误。错误必须被特殊处理–他们不被包含在流的值当中。
- completed事件表明这个信号成功完成了发送，这个流将不会再有更多地值。完成事件必须被特殊处理–它不被包含在流的值当中。

一个信号的完整生命周期中可能包含多个next事件，接着可能是一个error或者completed事件。

```
//1.创建信号
    RACSignal *signal = [RACSignal createSignal:^RACDisposable *(id<RACSubsc
riber> 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:@"Hello"];
        [subscriber sendNext:@"Reactive"];

        // 如果不再发送数据，最好发送信号完成，内部会自动调用[RACDisposable disposab
le]取消订阅信号。
        //  [subscriber sendCompleted];
       //  [subscriber sendNext:@"Cocoa"];
        return nil;
    }];
```

```
    //3.订阅信号，才会激活信号
    [signal subscribeNext:^(id x) {
        NSLog(@"%@",x);
    } error:^(NSError *error) {
        NSLog(@"%@",error);
    } completed:^{
        NSLog(@"completed");
    }];
```

RACSubject

一个subject，代表一个RACSubject类对象，是一种可以手动控制的信号。
Subjects可以被认为是一种"可变（mutable）"的信号，就像NSMutableArray和NSArray。
Subjects对于将非RAC的变得RAC非常有用。
比如，我们可以不用在block当中处理程序的回调，这些block可以将事件通过一个subject单例发送出去。这个subject可以以一个RACSignal的形式返回，并隐藏掉这些回调的实现细节。
RACSubject必须先订阅信号，再发送信号

```
//1.创建信号
RACSubject *subject = [RACSubject subject];

//2.1订阅信号
[subject subscribeNext:^(id x) {
   NSLog(@"第一个订阅者：%@",x);
}];

[subject subscribeNext:^(id x) {
   NSLog(@"第二个订阅者：%@",x);
}];

//2.2订阅错误
[subject subscribeError:^(NSError *error) {
   NSLog(@"订阅错误信息：%@",error);
}];

//3.发送信号
[subject sendNext:@"Hello"];
[subject sendNext:@"Ractive"];
[subject sendError:[NSError errorWithDomain:@"error message" code:0 userInfo
:nil]];
[subject sendNext:@"Cocoa"];
```

RACSequences

一个序列代表一个RACSequence对象,是一个 拉动式（pull-driven） 的流。
序列一种集合，类似于 NSArray。和数组不同的是，序列中的值默认是**懒演算**的。（比如，当他们被需要时），这样就能在序列中只有一部分的值被使用的时候提升性能。和Cocoa的集合类型一样，序列中也不能包含 nil。

RAC给Cocoa中的大部分集合类型都添加了 –rac_sequence 方法，来让他们像 RACSequences 一样被使用。

```
NSArray *array = @[@1,@2,@3,@4,@5,@6,@7,@8,@9];

    RACSequence *sequence = array.rac_sequence;
    RACSignal *signal = sequence.signal;
    [signal subscribeNext:^(id x) {
        NSLog(@"%@",x);
    }];

 NSDictionary *dic = @{@"id":@"1",@"name":@"Kobe",@"age":@32};
    [dic.rac_sequence.signal subscribeNext:^(RACTuple *x) {
//      NSLog(@"%@:%@",x.first,x.second);
//      NSLog(@"%@:%@",x[0],x[1]);
        RACTupleUnpack(NSString *key,NSString *value) = x;
        NSLog(@"%@:%@",key,value);
    }];
```

RACSubscription

一个订阅者可以是任何正在等待或者能够获取[信号]中的事件的对象。在RAC当中，订阅者是任何实现了RACSubscriber协议的对象。
通过–subscribeNext:error:completed:或者对应的方便方法可以产生一个订阅。
订阅会将信号retain，并会在信号完成发送或者产生错误的时候被析构(disposed)。订阅也可被手动析构

RACCommand

RACCommand对象，创建并订阅一个信号以响应一些动作。动作使得一些会产生副作用的行为变得十分简单，比如用户操作App。
通常情况下一个command被UI事件触发，比如一个按钮被点击。Commands也可以通过一个信号来自动的被禁用或者可用，而且这个禁用状态也可以通过将和这个command相关的UI控件禁用来体现。
RAC为UIButton、UIBarButtonItem来添加一个rac_command属性来自动支持这些行为。

```
RACSignal *validUserNameSignal = [[self.txtUserName rac_textSignal] map:^id(
NSString *value) {
        return @([value isValidPhoneNum]);
    }];
RACCommand *btnLoginCommand = [[RACCommand alloc] initWithEnabled:validUserN
ameSignal signalBlock:^RACSignal *(id input) {
        NSLog(@"%@",[input class]);
        return [[RACSignal empty] logAll];
    }];
self.btnLogin.rac_command = btnLoginCommand;
```

RACDisposable

RACDisposable类对象被用来取消任务和清除资源。
析构被用来取消对一个信号的订阅。当一个订阅被取消时，相应的订阅者将不会再收到 任何 来自这个信号的事件。并且，任何和这个订阅相关的工作（后台处理，网络请求，等等。）都会被取消，因为这些结果都已经不需要了。

Schedulers

一个scheduler代表一个RACScheduler类对象，是一个顺序执行信号的队列，来开展他们的工作或者传递它们的结果。
Schedulers和GCD类似，但是schedulers支持取消（通过析构），而且一定是顺序执行的。除了 +immediateScheduler之外，schedulers不提供任何同步执行的方法。这有助于避免死锁，我们鼓励使用signal operators而不是阻塞工作。
RACScheduler也和NSOperationQueue类似，但是schedulers并不支持任务的重新排序或者任务间的相互依赖。

RACTuple

RAC提供一些通用的类，以让值在流中被传递:
RACTuple 是一个小巧的，constant-sized的集合类型，并可以包含nil（以RACTupleNil表示）。它通常被用来合并多个流中的值。
RACUnit是一个'空'值单例，当一个流中没有有意义的值存在时，并且流没有被关闭时，流使用这个值来表示无意义。
RACEvent将信号事件转化为一个值。它主要被RACSignal中的-materialize 方法使用。

```
RACTuple *tuple = RACTuplePack(@"jack",@30,@YES);
NSLog(@"first value:%@,second value:%@,third value:%@",tuple.first,tuple.sec
ond,tuple.third);

RACTupleUnpack(NSString *name,NSNumber *age,NSNumber *sex) = tuple;
NSLog(@"name:%@,age:%@,sex:%@",name,age,sex);
```

RAC与RACObserve

RAC(TARGET, [KEYPATH, [NIL_VALUE]]):用于给某个对象的某个属性绑定。

```
RAC(self.btnLogin,enabled) = [[self.txtUserName rac_textSignal]
                                    map:^id(NSString *value) {
                                        return @([value isValidPhoneNum]);
                                    }];
等价于:
@weakify(self)
[[self.txtUserName rac_textSignal]
map:^id(NSString *value) {
    return @([value isValidPhoneNum]);
} subscribeNext:^(NSNumber *value) {
```

```
    @strongify(self)
    self.btnLogin.enable = [value boolValue];
}];
```

RACObserve(self, name):监听某个对象的某个属性,返回的是信号。

```
RAC(self.btnLogin,backgroundColor) = [RACObserve(self.btnLogin, enabled)
                                      map:^id(id value) {
                                          return [value boolValue] ? Hex
RGB(0x27aa28):HexRGB(0xa5dea5);
                                      }];
```

ReactiveCocoa应用

消息传递机制实现统一

Objective-C中的消息传递机制包括block、target-action、delegate、Notification、KVO等。
传统范式中，逻辑被放在了很多方法里，零碎地摆放在view controller里，逻辑比较分散，代码不易维护。
RAC使消息传递机制实现了统一，代码紧凑，逻辑更加清晰。

- target-action

```
[[self.btnLogin
    rac_signalForControlEvents:UIControlEventTouchUpInside]
    subscribeNext:^(UIButton *sender) {
        //do something you want
    }];
```

- delegate

```
[[self
    rac_signalForSelector:@selector(webViewDidStartLoad:)
    fromProtocol:@protocol(UIWebViewDelegate)]
    subscribeNext:^(id x) {
        // 实现 webViewDidStartLoad: 代理方法
    }];
```

关于 -rac_signalForSelector:fromProtocol订阅不执行的问题，参考：https://github.com/ReactiveCocoa/ReactiveCocoa/issues/1121

```
    [[[self rac_signalForSelector:@selector(tableView:didSelectRowAtIn
dexPath:) fromProtocol:@protocol(UITableViewDelegate) ]
      filter:^BOOL(RACTuple *value) {
          NSIndexPath *indexPath = value.second;
          return (indexPath.section == 0 && indexPath.row == 0);
      }]
     subscribeNext:^(RACTuple *value) {
         //do something
        }
     }];
    //在 -rac_signalForSelector:fromProtocol: 调用后重置代理:，否则代码不会
执行，
    self.tableView.delegate = nil;
    self.tableView.delegate = self;
```

- 通知

```
[[[NSNotificationCenter defaultCenter]
    rac_addObserverForName:kReachabilityChangedNotification object:nil
]
    subscribeNext:^(NSNotification *notification) {
        // 收到 kReachabilityChangedNotification 通知
    }];
```

- KVO

```
@weakify(self)
[RACObserve(self, orderDetailModel) subscribeNext:^(HROnlineRepairOrde
rDetailModel *value) {
    @strongify(self)
    [self.tableView reloadData];
}];
```

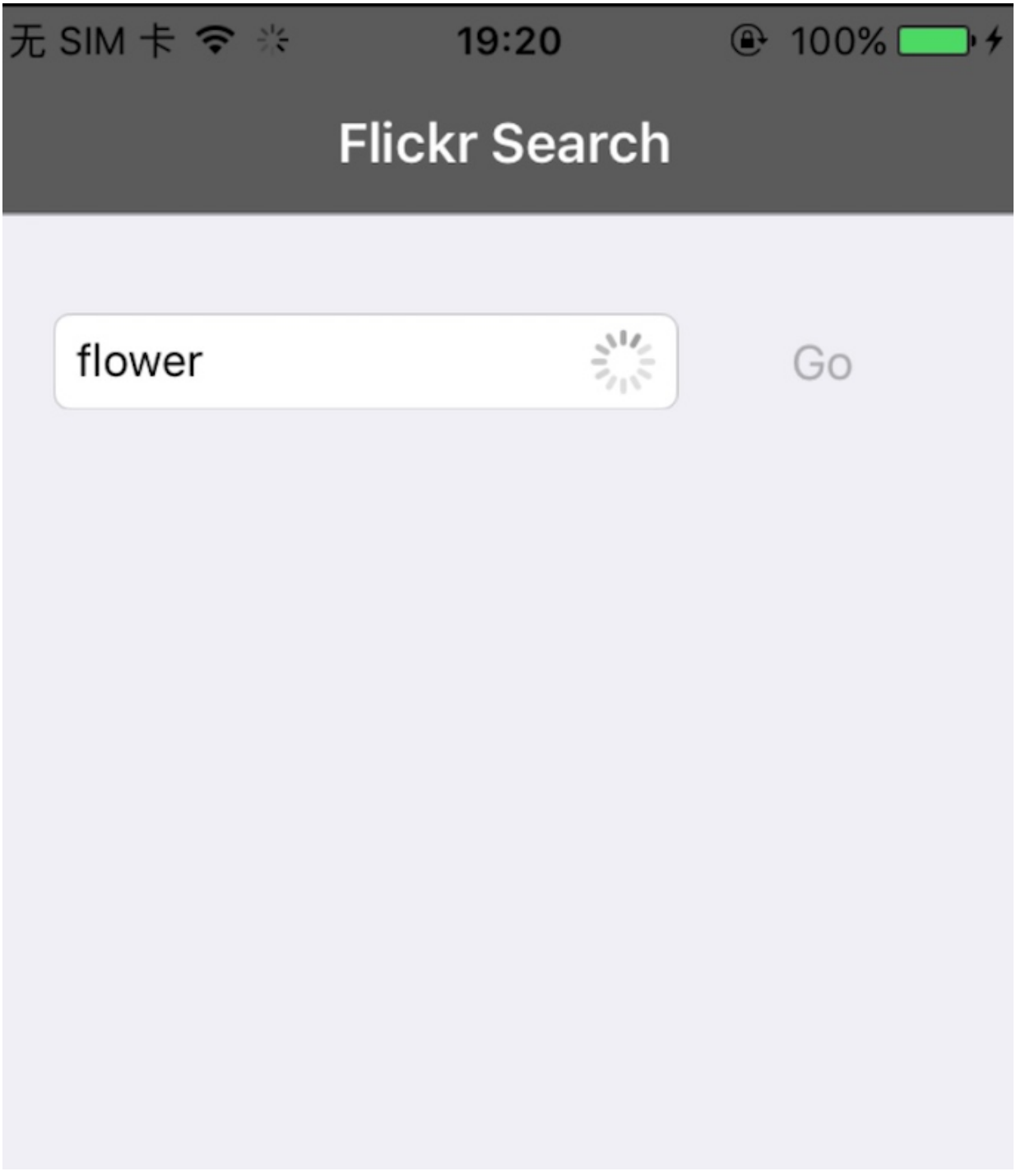
MutableArray添加、删除数据项，订阅方法不执行的解决办法

```
_repairItemArray = [NSMutableArray new];
//使用这种方式添加或删除数组项，否则数组中数据变化时，RACObserve无法响应，
//添加维修项
NSMutableArray *fromKVC = [self mutableArrayValueForKey:@"repairItemAr
ray"];
[fromKVC addObject:[HROnlineRepairItemModel new]];

//删除维修项
NSMutableArray *fromKVC = [self mutableArrayValueForKey:@"repairItemAr
ray"];
```

```
[fromKVC removeObjectAtIndex:headerIndex];

[RACObserve(self, repairItemArray) subscribeNext:^(NSArray *repairItem
s) {
    //do something
}];
```

参考：http://stackoverflow.com/questions/25162893/racobserve-for-value-update-in-nsmutablearray

事件绑定之RACCommand



控制器层：RWTFlickrSearchViewController

```
- (void)bindViewModel {

  self.searchButton.rac_command = self.viewModel.executeSearch;

  RAC(self.loadingIndicator, hidden) =
    [self.viewModel.executeSearch.executing not];

 RAC([UIApplication sharedApplication], networkActivityIndicatorVisible) = s
elf.viewModel.executeSearch.executing;

  [self.viewModel.executeSearch.executionSignals
    subscribeNext:^(id x) {
      [self.searchTextField resignFirstResponder];
    }];

  [self.viewModel.executeSearch.errors; subscribeNext:^(NSError *error) {
    UIAlertView *alert =
    [[UIAlertView alloc] initWithTitle:@"Connection Error"
                               message:@"There was a problem reaching Flickr
."
                              delegate:nil
                     cancelButtonTitle:@"OK"
                     otherButtonTitles:nil];
    [alert show];
  }];
}
```

ViewModel层：RWTFlickrSearchViewModel

```
- (void)initialize {

  RACSignal *validSearchSignal =
    [[RACObserve(self, searchText)
      map:^id(NSString *text) {
         return @(text.length > 3);
      }]
      distinctUntilChanged];

  self.executeSearch =
    [[RACCommand alloc] initWithEnabled:validSearchSignal
      signalBlock:^RACSignal *(id input) {
        return  [self executeSearchSignal];
      }];
 }
```

实现RACCommand属性的控件

- UIButton
- UIBarButtonItem

以上代码摘自：http://southpeak.github.io/2014/08/08/mvvm-tutorial-with-reactivecocoa-1/
关于RACCommand的使用参考：http://codeblog.shape.dk/blog/2013/12/05/reactivecocoa-essentials-understanding-and-using-raccommand

流运算符

常用的流运算符

序列和信号所共同使用的运算符被称之为流运算符。RAC提供的强大的运算符，基于这些运算符可实现许多复杂的功能。

名称	说明
filter	过滤信号，使用它可以获取满足条件的信号.
map	可以将一个流中的值映射成一个新的值,并返回一个带有新值的流
flattenMap	被用来转换每个流的值,然后再返回包含这个值的信号到一个新的流中.然后将对这个流降维,也就是说先-map:再-flatten.
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
combineLatest:reduce	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。
zip	Zips the values in the given streams to create RACTuples. The first value of each stream will be combined, then the second value, and so forth, until at least one of the streams is exhausted.
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
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
ignore	忽略某些值的信号.
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
take	从开始一共取N次的信号
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

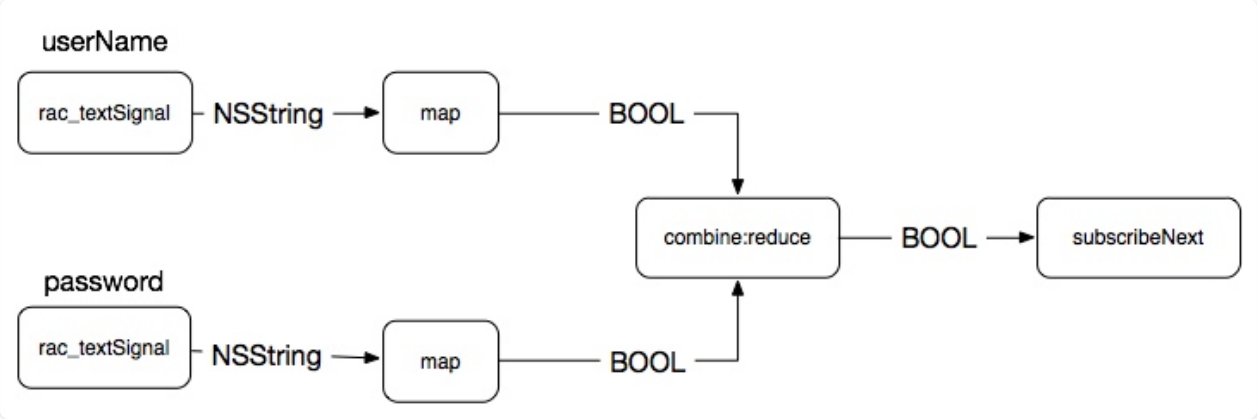
	成，就知道总共有多少信号
takeUntil	获取信号直到某个信号执行完成
skip	跳过几个信号,不接受
switchToLatest	用于signalOfSignals（信号的信号），有时候信号也会发出信号，会在signalOfSignals中，获取signalOfSignals发送的最新信号
doNext	给一个信号注入自定义操作,然后当自定义操作完成时返回一个信号
doCompleted	执行sendCompleted之前，会先执行这个Block
delay	延迟几秒后执行下一步操作
deliverOn	内容传递切换到制定线程中，副作用在原来线程中,把在创建信号时block中的代码称之为副作用
throttle	throttle操作只有在两次next事件间隔指定的时间时才会发送第二个next事件。
subscribeOn	内容传递和副作用都会切换到制定线程中
interval	定时：每隔一段时间发出信号

关于流运算符的图形化介绍，参考：http://neilpa.me/rac-marbles/#sampleOn

信号运算符基础用法


```
    RACSignal *validUserNameSignal = [[self.txtUserName rac_textSignal] map:
^id(NSString *value) {
        return @([value isValidPhoneNum]);
    }];

    RACSignal *validPasswordSignal = [[self.txtPassword rac_textSignal] map:
^id(NSString *value) {
        return @(value.length > 6);
    }];

    [[[RACSignal combineLatest:@[validUserNameSignal,validPasswordSignal] re
duce:^id(NSNumber *userName,NSNumber *password){
            return @([userName boolValue] && [password boolValue]);
        }] distinctUntilChanged]
        subscribeNext:^(id x) {
```

```
            self.btnLogin.enabled = [x boolValue];
            self.btnLogin.backgroundColor = [x boolValue] ? HexRGB(0x27aa28)
:HexRGB(0xa5dea5);
        }];     }
    }];
```



基于信号处理复杂网络请求问题

网络请求中常见的问题

- 有的需要一个接口返回后，根据返回的内容决定后续的接口访问，最终才能渲染。
- 有的需要几个接口都返回后才能统一渲染。
- 有的则是几个接口按照返回顺序依次渲染。

这就导致我们在处理这些逻辑的时候，需要很多的异步处理手段，利用一些中间变量来储存状态，每次返回的时候又判断这些状态决定渲染的逻辑。有的时候对于同时请求多个网络接口，某些出现了网络错误，异常处理也变得越来越复杂。
使用信号的组合等“高级”操作可以帮助我们解决很多的问题。例如flattenMap:可以解决接口串联的问题，zip:操作可以解决多个接口都返回后再渲染的问题，merge:操作可以解决依次渲染的问题。

- 创建网络请求信号

```
//获取维修空间
- (RACSignal *)requestRepairZoneSignal{
    @weakify(self)
    return [RACSignal createSignal:^RACDisposable *(id<RACSubscriber>
subscriber) {
        @strongify(self)
        NSDictionary *dic = @{@"czhth":[self getCurrentContractCode]};
     dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(5 * NSE
C_PER_SEC)), dispatch_get_main_queue(), ^{
            [ZRServiceRepairSubmitManager downloadServiceRepairKJModel
sWithPramas:dic completion:^(NSArray *models, ZRError *error) {
                if (!error) {
                    NSLog(@"request repair zone finished");
                    [subscriber sendNext:models];
                    [subscriber sendCompleted];
```

```
                } else {
                    [subscriber sendError:error];
                }
            } failure:^(ZRError *error) {
                [self.view hideHUD];
                [subscriber sendError:[NSError errorWithDomain:kHR_Net
WorkFailureTip code:0 userInfo:nil]];
            }];
        });
        return nil;
    }];
}

//根据维修空间code获取维修类型
- (RACSignal *)requestRepairTypeSignalWithZoneCode:(NSString *)zoneCod
e{
    return [RACSignal createSignal:^RACDisposable *(id<RACSubscriber>
subscriber) {
        NSDictionary *dic = @{@"kjCode": zoneCode ?: @""};
        [ZRServiceRepairSubmitManager downloadServiceRepairZTModelsWit
hPramas:dic completion:^(NSArray *models, ZRError *error) {
            if (!error) {
                [subscriber sendNext:models];
                [subscriber sendCompleted];
            }else {
                [subscriber sendError:error];
            }
        } failure:^(ZRError *error) {
            [subscriber sendError:[NSError errorWithDomain:kHR_NetWork
FailureTip code:0 userInfo:nil]];
        }];
        return nil;
    }];
}
```

- 使用flattenMap解决接口串联问题

```
[self.view showHUD];
@weakify(self)
[[[[self requestRepairZoneSignal]
 map:^id(NSArray *value) {
     return [value.firstObject valueForKey:@"code"];
 }]
flattenMap:^RACStream *(NSString *code) {
    @strongify(self)
    return [self requestRepairTypeSignalWithZoneCode:code];
}]
subscribeNext:^(NSArray *value) {
    @strongify(self)
    [self.view hideHUD];
```

```
    NSLog(@"result:%@",value);
} error:^(NSError *error) {
    @strongify(self)
    [self.view hideHUD];
    NSLog(@"some error happend");
}];
```

- 使用zip操作解决多个接口都返回后再渲染的问题

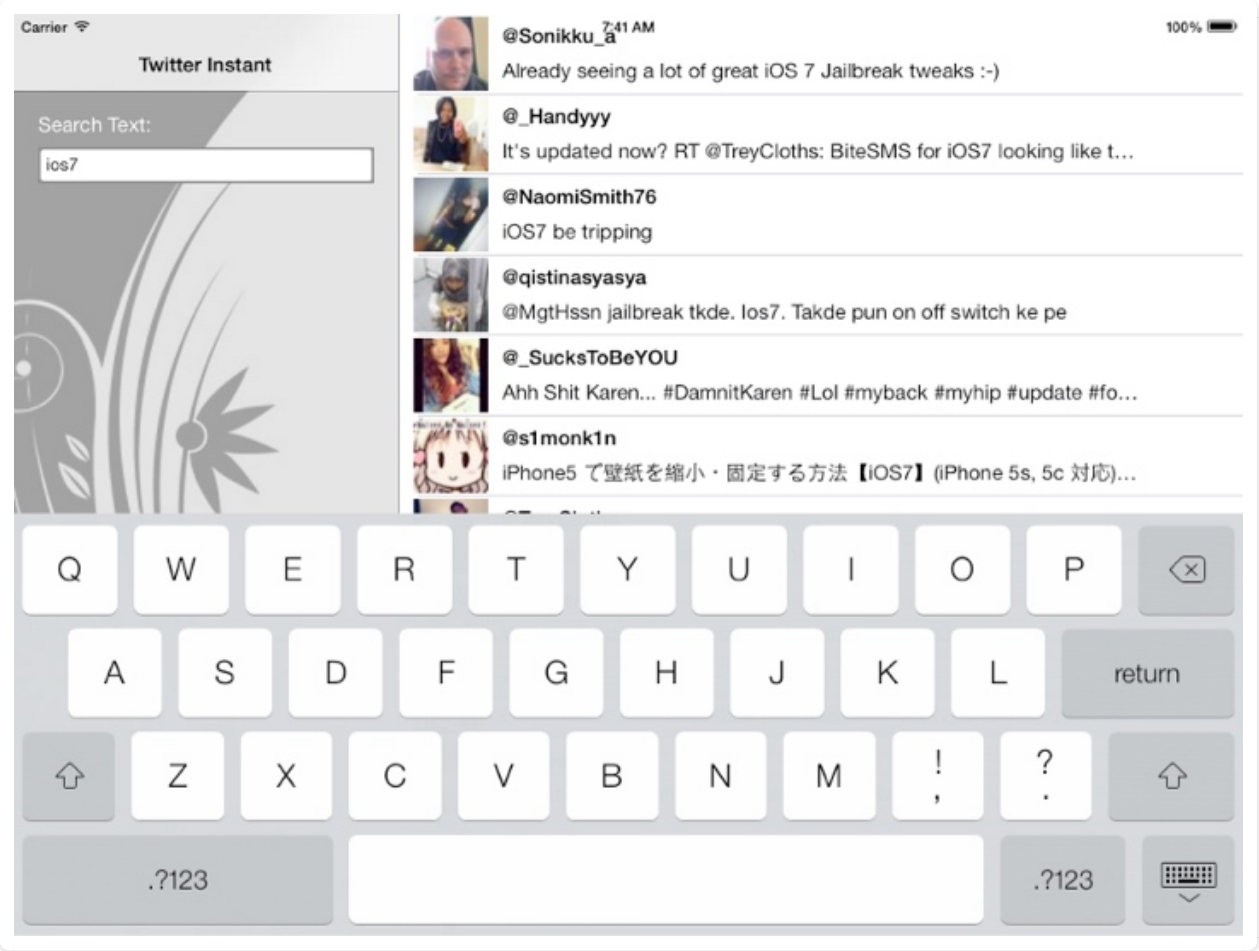
```
@weakify(self)
[self.view showHUD];
[[RACSignal zip:@[
                [self requestRepairZoneSignal],
                [self requestRepairTypeSignalWithZoneCode:zoneCode],
                ]]
subscribeNext:^(RACTuple *value) {
    @strongify(self)
    [self.view hideHUD];
    NSLog(@"all request finished:%@",value.first);
} error:^(NSError *error) {
    NSLog(@"some error happend");
}];
```

- 使用merge解决按返回顺序依次渲染的问题

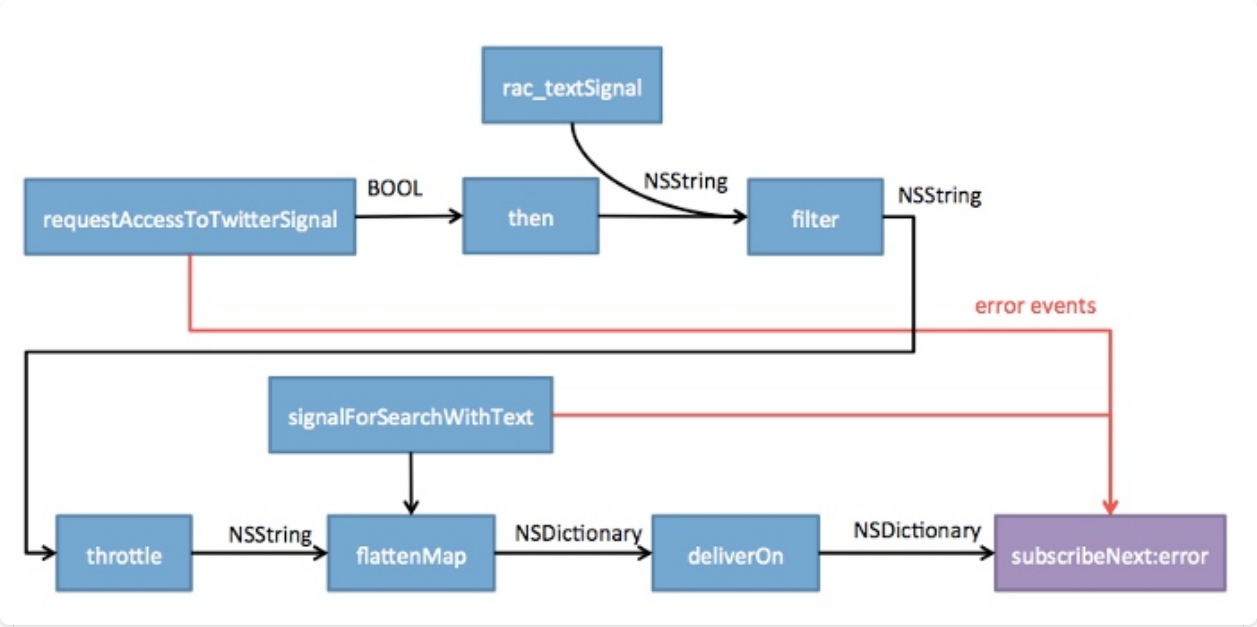
```
[[RACSignal merge:@[
                  [self requestRepairZoneSignal],
                  [self requestRepairTypeSignalWithZoneCode:zoneCode]
]
 ]
subscribeNext:^(id x) {
    NSLog(@"one of request finished:%@",x);
} error:^(NSError *error) {
    NSLog(@"some error happend");
}];
```

综合应用

样例摘自『ReactiveCocoa Tutorial – the Definitive Introduction: Part 2/2』教程（http://southpeak.github.io/2014/08/02/reactivecocoa-tutorial-the-definitive-introduction-2/）




```
[[[[[[[self requestAccessToTwitterSignal]
      then:^RACSignal *{
          @strongify(self)
          return self.searchText.rac_textSignal;
      }]
     filter:^BOOL(NSString *text) {
         @strongify(self)
         return [self isValidSearchText:text];
     }]
    throttle:0.5]
   flattenMap:^RACStream *(NSString *text) {
       @strongify(self)
       return [self signalForSearchWithText:text];
   }]
  deliverOn:[RACScheduler mainThreadScheduler]]
 subscribeNext:^(NSDictionary *jsonSearchResult) {
     NSArray *statuses = jsonSearchResult[@"statuses"];
     NSArray *tweets = [statuses linq_select:^id(id tweet) {
         return [RWTweet tweetWithStatus:tweet];
     }];
     [self.resultsViewController displayTweets:tweets];
 } error:^(NSError *error) {
     NSLog(@"An error occurred: %@", error);
 }];
```




```
- (RACSignal *)requestAccessToTwitterSignal{
    // 定义一个错误，如果用户拒绝访问则发送
    NSError *accessError = [NSError errorWithDomain:RWTwitterInstantDomain c
ode:RWTwitterInstantErrorAccessDenied userInfo:nil];
    // 创建并返回信号
    @weakify(self)
    return [RACSignal createSignal:^RACDisposable *(id<RACSubscriber> subscr
iber) {
        // 请求访问twitter
        @strongify(self)
        [self.accountStore requestAccessToAccountsWithType:self.twitterAccou
ntType
                                                   options:nil
                                                completion:^(BOOL granted, N
SError *error) {

                                                    // 处理响应
                                                    if (!granted){
                                                   [subscriber sendError:acc
essError];

                                                    } else {
                                                        [subscriber sendNext
:nil];

                                                        [subscriber sendComp
leted];

                                                    }
                                                }];

        return nil;
    }];
}
```

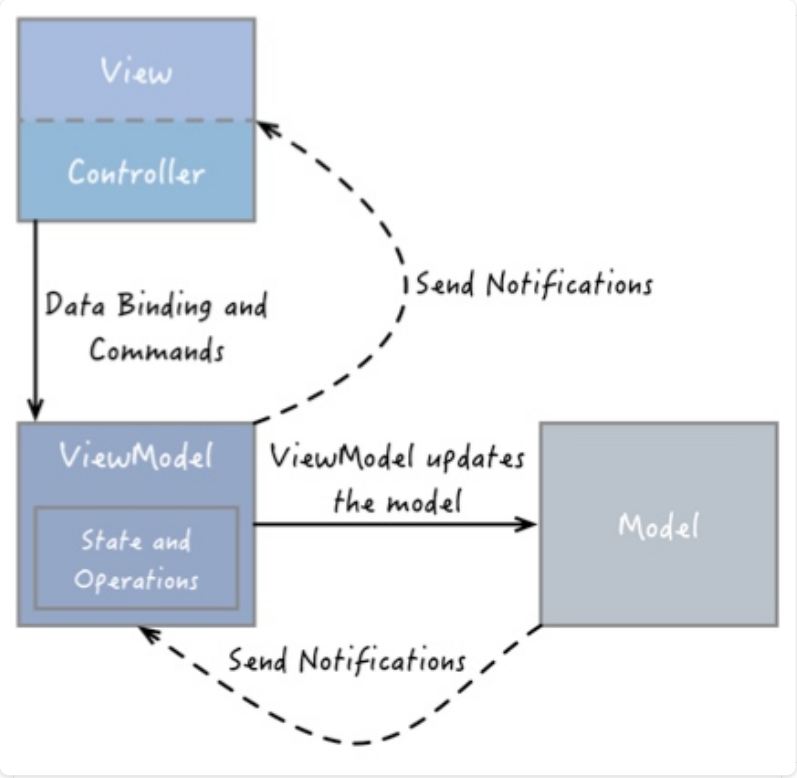
```
- (RACSignal *)signalForSearchWithText:(NSString *)text {
    // 定义错误
    NSError *noAccountError = [NSError errorWithDomain:RWTwitterInstantDomai
```

```
n code:RWTwitterInstantErrorNoTwitterAccounts userInfo:nil];
    NSError *invalidResponseError = [NSError errorWithDomain:RWTwitterInstan
tDomain code:RWTwitterInstantErrorInvalidResponse userInfo:nil];
    // 创建信号block
    @weakify(self)
    return [RACSignal createSignal:^RACDisposable *(id<RACSubscriber> subscr
iber) {
        @strongify(self)
        // 创建请求
        SLRequest *request = [self requestforTwitterSearchWithText:text];
        // 提供Twitter账户
        NSArray *twitterAccounts = [self.accountStore accountsWithAccountTyp
e:self.twitterAccountType];
        if (twitterAccounts.count == 0) {
            [subscriber sendError:noAccountError];
        } else {
            [request setAccount:[twitterAccounts lastObject]];
            // 执行请求
            [request performRequestWithHandler:^(NSData *responseData, NSHTT
PURLResponse *urlResponse, NSError *error) {
                if (urlResponse.statusCode == 200) {
                    // 成功，解析响应
                    NSDictionary *timelineData = [NSJSONSerialization JSONOb
jectWithData:responseData options:NSJSONReadingAllowFragments error:nil];
                    [subscriber sendNext:timelineData];
                    [subscriber sendCompleted];
                } else {
                    // 失败，发送一个错误
                    [subscriber sendError:invalidResponseError];
                }
            }];
        }
        return nil;
    }];
}
```

RAC与MVVM

MVVM简介

MVVM是一个UI设计模式。它是 MV* 模式集合中的一员。MV*模式还包含MVC(Model View Controller)、MVP(Model View Presenter)等。这些模式的目的在于将UI逻辑与业务逻辑分离，以让程序更容易开发和测试。



MVVM模式的核心是ViewModel，它是一种特殊的model类型，用于表示程序的UI状态。它包含描述每个UI控件的状态的属性。例如，文本输入域的当前文本，或者一个特定按钮是否可用。它同样暴露了视图可以执行哪些行为。

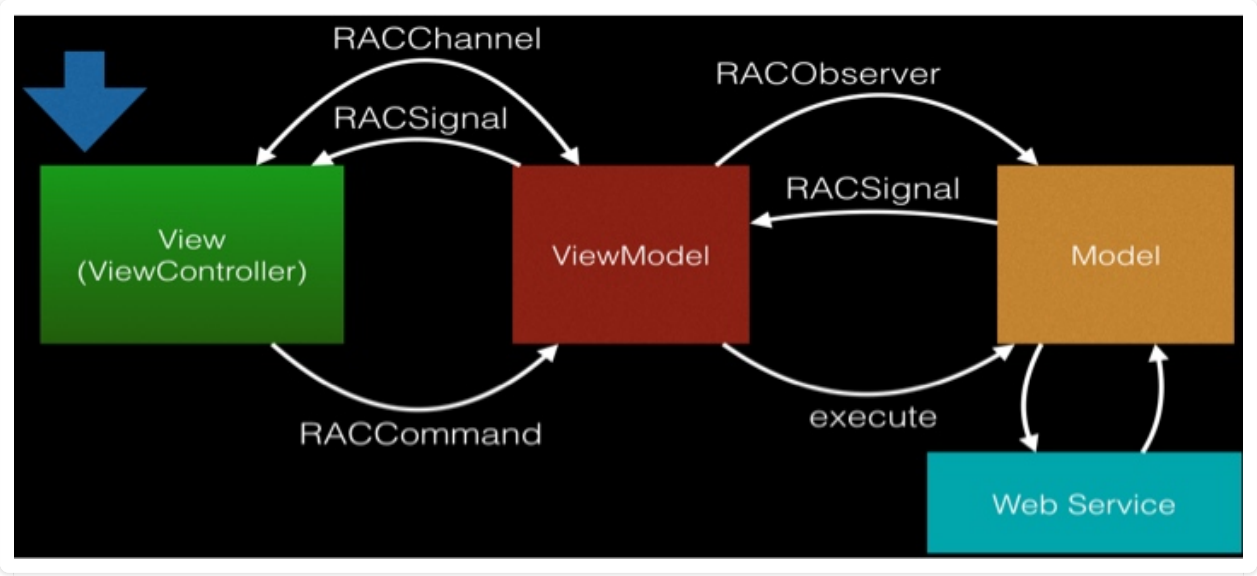
MVVM规则：

- View引用了ViewModel，但反过来不行。
- ViewModel引用了Model，但反过来不行。

MVVM优点：

- 轻量的视图：View层很薄，只提供ViewModel状态的显示及输出用户交互事件，所有的UI逻辑都在ViewModel中。
- 便于测试：我们可以在没有视图的情况下运行整个程序，这样大大地增加了它的可测试性。

RAC在MVVM中的应用



- View和ViewModel间通过RACSignal来进行单向绑定，通过RACChannel来进行双向绑定，通过RACCommand进行执行过程的绑定。
- ViewModel和Model间通过RACObserve进行监听，通过RACSignal进行回调处理，也可以直接调用方法。
- Model有自身的数据业务逻辑，包含请求Web Service和进行本地持久化。

总结

优点

- 减少编程中的状态变量存储，解决状态以及状态之间依赖过多的问题
- 提供统一的消息传递机制，使代码统一，逻辑清晰，易于维护
- 提供了强大的信号处理功能，易于实现传统编程中难处理的一些问题
- 结合MVVM设计模式，实现UI与业务逻辑分层，减少Controller代码，增强程序可测试性

缺点

- 框架复杂，与传统编程思想不同，概念抽象，学习成本较高。

学习资源

标题	网址
官方主页	https://github.com/ReactiveCocoa/ReactiveCocoa
RAC中文资源整合	https://github.com/ReactiveCocoaChina/
Learn ReactiveCocoa Source	https://github.com/KevinHM/FunctionalReactiveProgrammingOniOS
ReactiveCocoa	http://southpeak.github.io/2014/08/02/reactivecocoa-tutorial-

Tutorial 一 The Definitive Introduction	the-definitive-introduction-1/ http://southpeak.github.io/2014/08/02/reactivecocoa-tutorial-the-definitive-introduction-2/
iOS开发下的函数响应式编程（美团·大众点评框架演变）	http://williamzang.com/blog/2016/06/27/ios-kai-fa-xia-de-han-shu-xiang-ying-shi-bian-cheng/
ReactiveCocoa 中 RACSignal 是如何发送信号的	http://ios.jobbole.com/90676/
响应式编程框架ReactiveCocoa介绍与入门	http://blog.csdn.net/chenyufeng1991/article/details/51470316
这样好用的ReactiveCocoa，根本停不下来	http://ios.jobbole.com/82356/
MVVM Tutorial with ReactiveCocoa	http://southpeak.github.io/2014/08/08/mvvm-tutorial-with-reactivecocoa-1/ http://southpeak.github.io/2014/08/12/mvvm-tutorial-with-reactivecocoa-2/
...	...