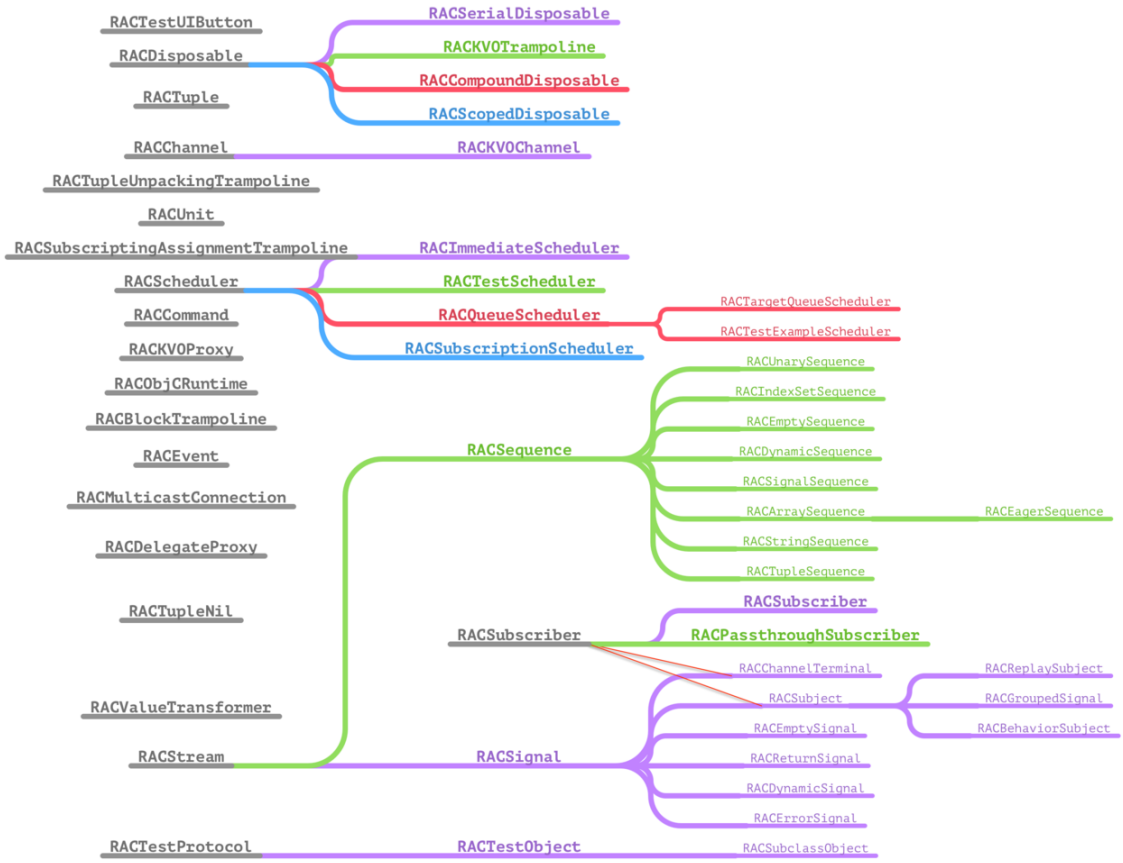
概述：

ReactiveCocoa是github开源的一个函数式响应式编程框架，是在iOS平台上对FRP的实现。FRP的核心是信号，信号在ReactiveCocoa(以下简称RAC)中是通过 `RACSignal` 来表示的，信号是数据流，可以被绑定和传递。

ReactiveCocoa比较复杂，在正式开始介绍它的核心组件前，我们先来看看它的类图，以便从宏观上了解它的层次结构：



ReactiveCocoa主要包含四个组件：

- 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。

而信号源是最核心的部分，其它所有组件都是围绕它运作的。 ReactiveCocoa最简单的工作过程就是 创建信号——订阅信号——发送信号。 所以首先我们就来介绍一下信号。

一、信号源

冷信号与热信号：

信号分为冷信号与热信号，理解冷信号与热信号的区别，对于RAC的理解有非常大的帮助，所以我们这篇文章也重点讲解这里。

- Hot Observable是主动的，尽管你并没有订阅事件，但是它会时刻推送，就像鼠标移动；而Cold Observable是被动的，只有当你订阅的时候，它才会发布消息。
- Hot Observable可以有多个订阅者，是一对多，集合可以与订阅者共享信息；而Cold Observable只能一对一，当有不同的订阅者，消息是重新完整发送。

而在RAC中除了RACSubject及其子类是热信号外，其它都是冷信号。subject类似“直播”，错过了就不再处理。而signal类似“点播”，每次订阅都会从头开始。所以我们有理由认定subject天然就是热信号。

Subject具备如下三个特点：

- Subject是“可变”的。
- Subject是非RAC到RAC的一个桥梁。
- Subject可以附加行为，例如RACReplaySubject具备为未来订阅者缓冲事件的能力。

我们平常使用RACSignal最简单的步骤如下：

```
    //创建信号
    RACSignal *signal = [RACSignal createSignal:^RACDisposable *
(id<RACSubscriber> subscriber) {
        //发送信号
        [subscriber sendNext:@"发送的数据"];
        [subscriber sendCompleted];
        return nil;
    }];

    //接收信号
```

```
    [signal subscribeNext:^(id x) {
        NSLog(@"这里是接收到的数据：%@",x);
    }];
```

为了了解热信号与冷信号的区别，我们用两段代码来展示一下：

```
    //创建热信号
    RACSubject *subject = [RACSubject subject];
    [subject sendNext:@1];    //立即发送1
    [[RACScheduler mainThreadScheduler] afterDelay:0.5 schedule:^{
        [subject sendNext:@2];      //0.5秒后发送2
    }];

    [[RACScheduler mainThreadScheduler] afterDelay:2 schedule:^{
        [subject sendNext:@3];     //2秒后发送3
    }];
    [[RACScheduler mainThreadScheduler] afterDelay:0.1 schedule:^{
        [subject subscribeNext:^(id x) {
            NSLog(@"subject1接收到了%@",x);    //0.1秒后subject1订阅了
        }];
    }];
    [[RACScheduler mainThreadScheduler] afterDelay:1 schedule:^{
        [subject subscribeNext:^(id x) {
            NSLog(@"subject2接收到了%@",x);        //1秒后subject2订阅
了
        }];
    }];
```

```
    //创建冷信号
    RACSignal *signal = [RACSignal createSignal:^RACDisposable *
(id<RACSubscriber> subscriber) {
        [subscriber sendNext:@1];
        [[RACScheduler mainThreadScheduler] afterDelay:0.5
schedule:^{
            [subscriber sendNext:@2];
        }];
        [[RACScheduler mainThreadScheduler] afterDelay:2
schedule:^{
            [subscriber sendNext:@3];
        }];
        return nil;
    }];
```

```
    [[RACScheduler mainThreadScheduler] afterDelay:0.1 schedule:^{
        [signal subscribeNext:^(id x) {
            NSLog(@"signal1接收到了%@", x);
        }];
    }];
    [[RACScheduler mainThreadScheduler] afterDelay:1 schedule:^{
        [signal subscribeNext:^(id x) {
            NSLog(@"signal2接收到了%@", x);
        }];
    }];
```

猜想一下上面两段代码的输出会是什么

。。。。。。。。。。。。。。。

```
2018-02-20 11:02:24.980462+0800 RACTest[14912:16295891] subject1接收到了2
2018-02-20 11:02:26.480232+0800 RACTest[14912:16295891] subject1接收到了3
2018-02-20 11:02:26.480408+0800 RACTest[14912:16295891] subject2接收到了3
```

```
2018-02-20 11:20:53.952995+0800 RACTest[15075:16311621] signal1接收到了1
2018-02-20 11:20:54.456881+0800 RACTest[15075:16311621] signal1接收到了2
2018-02-20 11:20:54.457046+0800 RACTest[15075:16311621] signal1接收到了3
2018-02-20 11:20:54.853391+0800 RACTest[15075:16311621] signal2接收到了1
2018-02-20 11:20:55.356641+0800 RACTest[15075:16311621] signal2接收到了2
2018-02-20 11:20:55.356851+0800 RACTest[15075:16311621] signal2接收到了3
```

两段代码很简单，我也做了注释，就不再多做解释，从输出中我们可以发现：

- 0.1秒后订阅的subject1接收到了0.5秒后2秒后发送的信号，没有接收到之前发

送的新号。

- 1秒后订阅的subject2接收到了2秒后发送的信号，也没有接收到之前发送的新号。
- signal1和signal2都接收到了所有信号。

从中我们可以得出结论：

1. 热信号是主动的，即使你没有订阅事件，它仍然会时刻推送。如上面没有接收到的信号都是因为在没有订阅者的时候，它也会推送出去。而冷信号是被动的，只有当你订阅的时候，它才会发送消息。如第二段代码，订阅后才把信号推送出去。
2. 热信号可以有多个订阅者，是一对多，信号可以与订阅者共享信息。如第一段代码，订阅者1和订阅者2是共享的，他们都能在同一时间接收到3这个值。而冷信号只能一对一，当有不同的订阅者，消息会从新完整发送。如第一个例子，我们可以观察到两个订阅者没有联系，都是基于各自的订阅时间开始接收消息的。

将冷信号转变为热信号

RAC库中对于冷信号转化成热信号有如下标准的封装：

```
- (RACMulticastConnection *)publish;
- (RACMulticastConnection *)multicast:(RACSubject *)subject;
- (RACSignal *)replay;
- (RACSignal *)replayLast;
- (RACSignal *)replayLazily;
```

如上面的第一段代码，我们可以用如下来达到同样的效果：

```
    RACMulticastConnection *connection = [[RACSignal
createSignal:^RACDisposable *(id<RACSubscriber> subscriber) {
            [subscriber sendNext:@1];

        [[RACScheduler mainThreadScheduler] afterDelay:0.5
schedule:^{
            [subscriber sendNext:@2];
        }];

        [[RACScheduler mainThreadScheduler] afterDelay:2
schedule:^{
```

```
            [subscriber sendNext:@3];
        }];
        return nil;
    }] publish];
    [connection connect];
    RACSignal *signal = connection.signal;

    [[RACScheduler mainThreadScheduler] afterDelay:0.1 schedule:^{
        [signal subscribeNext:^(id x) {
            NSLog(@"这里是热信号1，接收到了%@", x);
        }];
    }];

    [[RACScheduler mainThreadScheduler] afterDelay:1 schedule:^{
        [signal subscribeNext:^(id x) {
            NSLog(@"这里是热信号2，接收到了%@", x);
        }];
    }];
```

输出如下：

```
2018-02-20 11:45:38.331464+0800 RACTest[15171:16331870] 这里是热信号
1，接收到了2
2018-02-20 11:45:39.830300+0800 RACTest[15171:16331870] 这里是热信号
1，接收到了3
2018-02-20 11:45:39.830870+0800 RACTest[15171:16331870] 这里是热信号
2，接收到了3
```

可以看到，现在已经是热信号了，和前面的RACSubject相同。

感兴趣的同学可以去看看 `- (RACMulticastConnection *)multicast:(RACSubject *)subject` 这个方法的实现，它是将冷信号转换为热信号的核心。其实它的本质就是使用一个Subject来订阅原始信号，并让其他订阅者订阅这个Subject，这个Subject就是热信号。

使用信号中常见的问题：

1、多次订阅

对RAC的信号进行转换的时候，其实就是对原有的信号进行订阅从而产生新的信

号。如下代码所示：

```
    RACSignal *signal = [RACSignal createSignal:^RACDisposable *
(id<RACSubscriber> subscriber) {
        NSLog(@"来了");
        //网络请求，产生model
        [subscriber sendNext:model];
        return nil;
    }];

    RACSignal *name = [signal flattenMap:^RACStream *(Person
*model) {
        return [RACSignal return:model.name];
    }];
    RACSignal *age = [signal flattenMap:^RACStream *(Person *model)
{
        return [RACSignal return:model.age];
    }];

    RAC(self.userNameTextFiled,text) = [[name catchTo:[RACSignal
return:@"error"]] startWith:@"name:"];
    RAC(self.passwordTextField,text) = [[age catchTo:[RACSignal
return:@"error"]] startWith:@"age:"];
```

上面分别对model进行了map，也就是产生了两个新的信号，然后再对两个信号进行订阅，对这两个信号订阅的时候，也会对间接对原信号进行订阅，从而造成对原信号的多次订阅，如上所示来了就输出了三次，如果是网络请求的话，也会输出三次，所以一定在信号转换的时候一定要注意这些情况。

要解决也很简单,把signal转换成热信号就行了

```
    RACSignal *signal = [[RACSignal createSignal:^RACDisposable *
(id<RACSubscriber> subscriber) {
        NSLog(@"来了");
        [subscriber sendNext:model];
        return nil;
    }] replayLazily];     //转换为热信号

    RACSignal *name = [signal flattenMap:^RACStream *(Person
*model) {
        return [RACSignal return:model.name];
    }];
    RACSignal *age = [signal flattenMap:^RACStream *(Person *model)
```

```
{
        return [RACSignal return:model.age];
    }];

    RAC(self.userNameTextFiled,text) = [[name catchTo:[RACSignal
return:@"error"]] startWith:@"name:"];
    RAC(self.passwordTextField,text) = [[age catchTo:[RACSignal
return:@"error"]] startWith:@"age:"];
```

2、内存泄漏

```
    RACSignal *signal = [RACSignal createSignal:^RACDisposable *
(id<RACSubscriber> subscriber) { //1
        Person *model = [[Person alloc] init];
        [subscriber sendNext:model];
        [subscriber sendCompleted];
        return nil;
    }];
    self.flattenMapSignal = [signal flattenMap:^RACStream *(Person
*model) { //2
        return RACObserve(model, name);
    }];
    [self.flattenMapSignal subscribeNext:^(id x) { //3
        NSLog(@"recieve - %@", x);
    }];
```

如上代码，看起来工作正常，但你使用内存检测工具会发现，这里会造成内存泄漏，原因就是

```
#define RACObserve(TARGET, KEYPATH) \
    ({ \
        _Pragma("clang diagnostic push") \
        _Pragma("clang diagnostic ignored \"-Wreceiver-is-weak\"")
\
        __weak id target_ = (TARGET); \
        [target_ rac_valuesForKeyPath:@keypath(TARGET, KEYPATH)
observer:self]; \
        _Pragma("clang diagnostic pop") \
    })
```

这段代码，所以这里的Block引用了self，就造成了循环引用。 解决办法也很简单，使用@weakify和@strongify即可：

```
RACSignal *signal = [RACSignal createSignal:^RACDisposable *
(id<RACSubscriber> subscriber) {
        Person *model = [[Person alloc] init];
        [subscriber sendNext:model];
        [subscriber sendCompleted];
        return nil;
    }];
    @weakify(self);
    self.flattenMapSignal = [signal flattenMap:^RACStream *(Person
*model) {
        @strongify(self);
        return RACObserve(model, name);
    }];
    [self.flattenMapSignal subscribeNext:^(id x) {
        NSLog(@"recieve - %@", x);
    }];
```

本来还想介绍一下另外三个组件的，但是由于时间匆忙，马上要去赶火车了，暂时就写到这里。剩下三个我就简单介绍一下吧

订阅者：在 ReactiveCocoa 中，订阅者是一个抽象的概念，所有实现了 RACSubscriber 协议的类都可以作为信号源的订阅者。

```
@protocol RACSubscriber <NSObject>

@required

/// Sends the next value to subscribers.
///
/// value - The value to send. This can be `nil`.
- (void)sendNext:(id)value;

/// Sends the error to subscribers.
///
/// error - The error to send. This can be `nil`.
///
/// This terminates the subscription, and invalidates the
subscriber (such that
/// it cannot subscribe to anything else in the future).
- (void)sendError:(NSError *)error;
```

```
/// Sends completed to subscribers.
///
/// This terminates the subscription, and invalidates the
subscriber (such that
/// it cannot subscribe to anything else in the future).
- (void)sendCompleted;

/// Sends the subscriber a disposable that represents one of its
subscriptions.
///
/// A subscriber may receive multiple disposables if it gets
subscribed to
/// multiple signals; however, any error or completed events must
terminate _all_
/// subscriptions.
- (void)didSubscribeWithDisposable:(RACCompoundDisposable
*)disposable;

@end
```

即实现了这四个方法的类。

调度器：RACScheduler 在 ReactiveCocoa 中就是扮演着调度器的角色，本质上，它就是用 GCD 的串行队列来实现的，并且支持取消操作。是的，在 ReactiveCocoa 中，并没有使用到 NSOperationQueue 和 NSRunloop 等技术，RACScheduler 也只是对 GCD 的简单封装而已。

清洁工：RACDisposable 在 ReactiveCocoa 中就充当着清洁工的角色，它封装了取消和清理一次订阅所必需的工作。它有一个核心的方法 -dispose ，调用这个方法就会执行相应的清理工作，这有点类似于 NSObject 的 -dealloc 方法。