RACSignal

信号是 ReactiveCocoa 中最基础概念之一就是。其中信号的实例为 RACRACStream .它两个子类 RACSignal 和 RACSequence .其中 RACSequence 用的较少。本次只分析 RACSignal 。

```
RACSignal *signal = [RACSignal createSignal:
                     ^RACDisposable *(id<RACSubscriber> subscriber)
{
    [subscriber sendNext:@1];
    [subscriber sendNext:@2];
    [subscriber sendNext:@3];
    [subscriber sendCompleted];
    return [RACDisposable disposableWithBlock:^{
        NSLog(@"signal dispose");
    }];
}];
RACDisposable *disposable = [signal subscribeNext:^(id x) {
    NSLog(@"subscribe value = %@", x);
} error:^(NSError *error) {
    NSLog(@"error: %@", error);
} completed:^{
    NSLog(@"completed");
}];

[disposable dispose];
```

试分析下。以上代码分别作了什么。首先这是一个 RACSignal 作用的过程。
当调用了 RACSignal 中 createSignal 函数的时候，会调用 RACSignal 的子类 RACDynamicSignal 的 createSignal 函数。

```
+ (RACSignal *)createSignal:(RACDisposable * (^)(id<RACSubscriber> subscriber))didSubscribe {
 return [RACDynamicSignal createSignal:didSubscribe];
}
```

而 RACDynamicSignal 中 createSignal 的函数做了什么呢?

```
@implementation RACDynamicSignal

+ (RACSignal *)createSignal:(RACDisposable * (^)(id<RACSubscriber> subscriber))didSubscribe {
        RACDynamicSignal *signal = [[self alloc] init];
        signal->_didSubscribe = [didSubscribe copy];
        return [signal setNameWithFormat:@"+createSignal:"];
}
```

可以通过源码看出。只是生成了一个实例。保存了传入的 `didSubscribe` 。此时需要注意 `didSubscribe` 的类型。首先它是一个 `block` 。他的参数是 `id<RACSubscriber>` 返回值是 `RACDisposable` 。那么现在我们已经保存了传入的 `didSubscribe` 。现在我们需要找到它执行的时机。

首先我们找到 `RACSignal (Subscription)` 这个 `RACSignal` 的分类。

```
@implementation RACSignal (Subscription)

- (RACDisposable *)subscribe:(id<RACSubscriber>)subscriber {
        NSCAssert(NO, @"This method must be overridden by subclasses");
        return nil;
}

- (RACDisposable *)subscribeNext:(void (^)(id x))nextBlock {
        NSCParameterAssert(nextBlock != NULL);

        RACSubscriber *o = [RACSubscriber subscriberWithNext:nextBlock error:NULL completed:NULL];
        return [self subscribe:o];
}

- (RACDisposable *)subscribeNext:(void (^)(id x))nextBlock completed:(void (^)(void))completedBlock {
        NSCParameterAssert(nextBlock != NULL);
        NSCParameterAssert(completedBlock != NULL);

        RACSubscriber *o = [RACSubscriber subscriberWithNext:nextBlock error:NULL completed:completedBl
        return [self subscribe:o];
}

- (RACDisposable *)subscribeNext:(void (^)(id x))nextBlock error:(void (^)(NSError *error))errorBlock c
        NSCParameterAssert(nextBlock != NULL);
        NSCParameterAssert(errorBlock != NULL);
        NSCParameterAssert(completedBlock != NULL);

        RACSubscriber *o = [RACSubscriber subscriberWithNext:nextBlock error:errorBlock completed:compl
        return [self subscribe:o];
}

- (RACDisposable *)subscribeError:(void (^)(NSError *error))errorBlock {
        NSCParameterAssert(errorBlock != NULL);

        RACSubscriber *o = [RACSubscriber subscriberWithNext:NULL error:errorBlock completed:NULL];
        return [self subscribe:o];
}

- (RACDisposable *)subscribeCompleted:(void (^)(void))completedBlock {
        NSCParameterAssert(completedBlock != NULL);

        RACSubscriber *o = [RACSubscriber subscriberWithNext:NULL error:NULL completed:completedBlock];
        return [self subscribe:o];
}

- (RACDisposable *)subscribeNext:(void (^)(id x))nextBlock error:(void (^)(NSError *error))errorBlock {
        NSCParameterAssert(nextBlock != NULL);
        NSCParameterAssert(errorBlock != NULL);

        RACSubscriber *o = [RACSubscriber subscriberWithNext:nextBlock error:errorBlock completed:NULL]
        return [self subscribe:o];
}

- (RACDisposable *)subscribeError:(void (^)(NSError *))errorBlock completed:(void (^)(void))completedBl
        NSCParameterAssert(completedBlock != NULL);
        NSCParameterAssert(errorBlock != NULL);

        RACSubscriber *o = [RACSubscriber subscriberWithNext:NULL error:errorBlock completed:completedB
        return [self subscribe:o];
}

@end
```

可以看到在信号被订阅的时候统一都生成了一个 `RACSubscriber` 。并通过 `[self subscribe:o]` 传递下去。接下来我们来看 `RACDynamicSignal` 中 `subscribe` 这个函数的实现。

```
@implementation RACDynamicSignal
- (RACDisposable *)subscribe:(id<RACSubscriber>)subscriber {
        NSCParameterAssert(subscriber != nil);

        RACCompoundDisposable *disposable = [RACCompoundDisposable compoundDisposable];
        subscriber = [[RACPassthroughSubscriber alloc] initWithSubscriber:subscriber signal:self dispos

        if (self.didSubscribe != NULL) {
                RACDisposable *schedulingDisposable = [RACScheduler.subscriptionScheduler schedule:^{
                        //重要细节
                        RACDisposable *innerDisposable = self.didSubscribe(subscriber);
                        [disposable addDisposable:innerDisposable];
                }];

                [disposable addDisposable:schedulingDisposable];
        }

        return disposable;
}
```

通过代码可以看到。调用了保存下来的 `didSubscribe` 这个 `block` ，同时将传入进来的 `subscriber` 当成参数传入进去。此时。我们便能确定。订阅式生成的 `subscriber` 和我们创建信号的传进来的 `subscriber` 。其实是同一个对象。那么接下来。我们来看下 `RACSubscriber` 到底作了什么。

```
@interface RACSubscriber ()

// These callbacks should only be accessed while synchronized on self.
@property (nonatomic, copy) void (^next)(id value);
@property (nonatomic, copy) void (^error)(NSError *error);
@property (nonatomic, copy) void (^completed)(void);

@property (nonatomic, strong, readonly) RACCompoundDisposable *disposable;

@end

@implementation RACSubscriber

#pragma mark Lifecycle

+ (instancetype)subscriberWithNext:(void (^)(id x))next error:(void (^)(NSError *error))error completed
        RACSubscriber *subscriber = [[self alloc] init];

        subscriber->_next = [next copy];
        subscriber->_error = [error copy];
        subscriber->_completed = [completed copy];

        return subscriber;
}
@end

- (void)sendNext:(id)value {
        @synchronized (self) {
                void (^nextBlock)(id) = [self.next copy];
                if (nextBlock == nil) return;

                nextBlock(value);
        }
}

- (void)sendError:(NSError *)e {
        @synchronized (self) {
                void (^errorBlock)(NSError *) = [self.error copy];
                [self.disposable dispose];

                if (errorBlock == nil) return;
                errorBlock(e);
        }
}

- (void)sendCompleted {
        @synchronized (self) {
                void (^completedBlock)(void) = [self.completed copy];
                [self.disposable dispose];

                if (completedBlock == nil) return;
                completedBlock();
        }
}
```

此时再看示例代码。就简单明了了。整体的流程也就是分成以下几个部分。

```
RACSignal *signal = [RACSignal createSignal:
                     ^RACDisposable *(id<RACSubscriber> subscriber)
{
    [subscriber sendNext:@1];
    [subscriber sendNext:@2];
    [subscriber sendNext:@3];
    [subscriber sendCompleted];
    return [RACDisposable disposableWithBlock:^{
        NSLog(@"signal dispose");
    }];
}];
RACDisposable *disposable = [signal subscribeNext:^(id x) {
    NSLog(@"subscribe value = %@", x);
} error:^(NSError *error) {
    NSLog(@"error: %@", error);
} completed:^{
    NSLog(@"completed");
}];

[disposable dispose];
```

- 创建新号。同时保存 `subscriber` 。等待信号被订阅是调用。
- 信号被订阅时，创建一个 `RACSubscriber` 实例.将订阅是传入的 `next` `error` `completed` 回调事件保存在创建的 `RACSubscriber` 实例中。
- 将创建的 `RACSubscriber` 实例传入创建信号是保存的 `subscriber` `block` 中
- 当 `RACSubscriber` 被调用对应的函数的时候。内部调用对应保存的 `block` 。

基础使用原理分析

concat

```
- (RACSignal *)concat:(RACSignal *)signal {
        return [[RACSignal createSignal:^(id<RACSubscriber> subscriber) {
                RACCompoundDisposable *compoundDisposable = [[RACCompoundDisposable alloc] init];

                RACDisposable *sourceDisposable = [self subscribeNext:^(id x) {
                        [subscriber sendNext:x];
                } error:^(NSError *error) {
                        [subscriber sendError:error];
                } completed:^{
                        RACDisposable *concattedDisposable = [signal subscribe:subscriber];
                        [compoundDisposable addDisposable:concattedDisposable];
                }];

                [compoundDisposable addDisposable:sourceDisposable];
                return compoundDisposable;
        }] setNameWithFormat:@"[%@] -concat: %@", self.name, signal];
}
```

调用 concat 时。首先创建了一个新的信号。订阅了第一个信号。当第一个信号结束的时候。开始订阅第二个信号。并将两个信号传递的消息通过新创建的信号的管道工人发送出去。

zipWith

```
- (RACSignal *)zipWith:(RACSignal *)signal {
        NSCParameterAssert(signal != nil);

        return [[RACSignal createSignal:^(id<RACSubscriber> subscriber) {
                __block BOOL selfCompleted = NO;
                NSMutableArray *selfValues = [NSMutableArray array];

                __block BOOL otherCompleted = NO;
                NSMutableArray *otherValues = [NSMutableArray array];

                void (^sendCompletedIfNecessary)(void) = ^{
                        @synchronized (selfValues) {
                                BOOL selfEmpty = (selfCompleted && selfValues.count == 0);
                                BOOL otherEmpty = (otherCompleted && otherValues.count == 0);
                                if (selfEmpty || otherEmpty) [subscriber sendCompleted];
                        }
                };

                void (^sendNext)(void) = ^{
                        @synchronized (selfValues) {
                                if (selfValues.count == 0) return;
                                if (otherValues.count == 0) return;

                                RACTuple *tuple = RACTuplePack(selfValues[0], otherValues[0]);
                                [selfValues removeObjectAtIndex:0];
                                [otherValues removeObjectAtIndex:0];

                                [subscriber sendNext:tuple];
                                sendCompletedIfNecessary();
                        }
                };

                RACDisposable *selfDisposable = [self subscribeNext:^(id x) {
                        @synchronized (selfValues) {
                                [selfValues addObject:x ?: RACTupleNil.tupleNil];
                                sendNext();
                        }
                } error:^(NSError *error) {
                        [subscriber sendError:error];
                } completed:^{
                        @synchronized (selfValues) {
                                selfCompleted = YES;
                                sendCompletedIfNecessary();
                        }
                }];

                RACDisposable *otherDisposable = [signal subscribeNext:^(id x) {
                        @synchronized (selfValues) {
                                [otherValues addObject:x ?: RACTupleNil.tupleNil];
                                sendNext();
                        }
                } error:^(NSError *error) {
                        [subscriber sendError:error];
                } completed:^{
                        @synchronized (selfValues) {
                                otherCompleted = YES;
                                sendCompletedIfNecessary();
                        }
                }];

                return [RACDisposable disposableWithBlock:^{
                        [selfDisposable dispose];
                        [otherDisposable dispose];
                }];
        }] setNameWithFormat:@"[%@] -zipWith: %@", self.name, signal];
}
```

原理和 concat 类似。创建新的信号。订阅传递进来的两个信号。匹配数据。通过新的信号的管道工人将组合好的数据发送出去。

bind

```
- (RACSignal *)bind:(RACSignalBindBlock (^)(void))block {
        NSCParameterAssert(block != NULL);

        /*
         * -bind: should:
         *
         * 1. Subscribe to the original signal of values.
         * 2. Any time the original signal sends a value, transform it using the binding block.
         * 3. If the binding block returns a signal, subscribe to it, and pass all of its values throug
         * 4. If the binding block asks the bind to terminate, complete the _original_ signal.
         * 5. When _all_ signals complete, send completed to the subscriber.
         *
         * If any signal sends an error at any point, send that to the subscriber.
         */

        return [[RACSignal createSignal:^(id<RACSubscriber> subscriber) {
                RACSignalBindBlock bindingBlock = block();

                __block volatile int32_t signalCount = 1;   // indicates self

                RACCompoundDisposable *compoundDisposable = [RACCompoundDisposable compoundDisposable];

                void (^completeSignal)(RACDisposable *) = ^(RACDisposable *finishedDisposable) {
                        if (OSAtomicDecrement32Barrier(&signalCount) == 0) {
                                [subscriber sendCompleted];
                                [compoundDisposable dispose];
                        } else {
                                [compoundDisposable removeDisposable:finishedDisposable];
                        }
                };

                void (^addSignal)(RACSignal *) = ^(RACSignal *signal) {
                        OSAtomicIncrement32Barrier(&signalCount);

                        RACSerialDisposable *selfDisposable = [[RACSerialDisposable alloc] init];
                        [compoundDisposable addDisposable:selfDisposable];

                        RACDisposable *disposable = [signal subscribeNext:^(id x) {
                                [subscriber sendNext:x];
                        } error:^(NSError *error) {
                                [compoundDisposable dispose];
                                [subscriber sendError:error];
                        } completed:^{
                                @autoreleasepool {
                                        completeSignal(selfDisposable);
                                }
                        }];

                        selfDisposable.disposable = disposable;
                };
                @autoreleasepool {
                        RACSerialDisposable *selfDisposable = [[RACSerialDisposable alloc] init];
                        [compoundDisposable addDisposable:selfDisposable];

                        RACDisposable *bindingDisposable = [self subscribeNext:^(id x) {
                                // Manually check disposal to handle synchronous errors.
                                if (compoundDisposable.disposed) return;

                                BOOL stop = NO;
                                id signal = bindingBlock(x, &stop);

                                @autoreleasepool {
                                        if (signal != nil) addSignal(signal);
                                        if (signal == nil || stop) {
                                                [selfDisposable dispose];
                                                completeSignal(selfDisposable);
                                        }
                                }
                        } error:^(NSError *error) {
                                [compoundDisposable dispose];
                                [subscriber sendError:error];
                        } completed:^{
                                @autoreleasepool {
                                        completeSignal(selfDisposable);
                                }
                        }];

                        selfDisposable.disposable = bindingDisposable;
                }

                return compoundDisposable;
        }] setNameWithFormat:@"[%@] -bind:", self.name];
}
```

- 1. 订阅当前的信号.
- 2. 任何时候当前信号发送了一个数据,使用绑定的 block 转换它。
- 3. 如果返回的是一个信号的话就订阅它。并在接收到数据时将其所有值传递给订阅者。
- 4. 如果返回值要求停止则完成当前信号。
- 5. 如果所有的信号都已经发送完毕。发送完成给当前订阅者。
- 6. 如果信号发送的 error,将其转发给当前订阅者.

merge flaten

```
+ (RACSignal *)merge:(id<NSFastEnumeration>)signals {
        NSMutableArray *copiedSignals = [[NSMutableArray alloc] init];
        for (RACSignal *signal in signals) {
                [copiedSignals addObject:signal];
        }

        return [[[RACSignal
                createSignal:^ RACDisposable * (id<RACSubscriber> subscriber) {
                        for (RACSignal *signal in copiedSignals) {
                                [subscriber sendNext:signal];
                        }

                        [subscriber sendCompleted];
                        return nil;
                }]
                flatten]
                setNameWithFormat:@"+merge: %@", copiedSignals];
}
- (__kindof RACStream *)flatten {
        return [[self flattenMap:^(id value) {
                return value;
        }] setNameWithFormat:@"[%@] -flatten", self.name];
}
- (__kindof RACStream *)flattenMap:(__kindof RACStream * (^)(id value))block {
        Class class = self.class;

        return [[self bind:^{
                return ^(id value, BOOL *stop) {
                        id stream = block(value) ?: [class empty];
                        NSCAssert([stream isKindOfClass:RACStream.class], @"Value returned from -flatte

                        return stream;
                };
        }] setNameWithFormat:@"[%@] -flattenMap:", self.name];
}
```

此处文字描述较为复杂。我会尽力说明。

- 1. 调用 merge 的时候创建了一个新的信号，我们将其命名为 new_merge_signal.
- 2. new_merge_signal 在被调用的时候。通过管道工人将需要 merge 的信号全部发送出去。
- 3. new_merge_signal 主动调用了 flatten， 而 flatten 调用了 falttenMap。
- 4. flattenMap 调用了 bind。
- 5. bind 创建了一个新的信号，我们将其命名为 new_bind_signal。
- 6. new_bind_signal 在被订阅的时候会主动订阅 new_merge_signal。new_merge_signal 在被订阅的时候会将所有需要 merge 的 signal 发送出去。 new_bind_signal 内部拿到需要 merge 的 signal 之后订阅所有需要合并的 signal。并将这些需要合并的 signal 发送的数据通过 new_bind_signal 的管道工人发送出去。