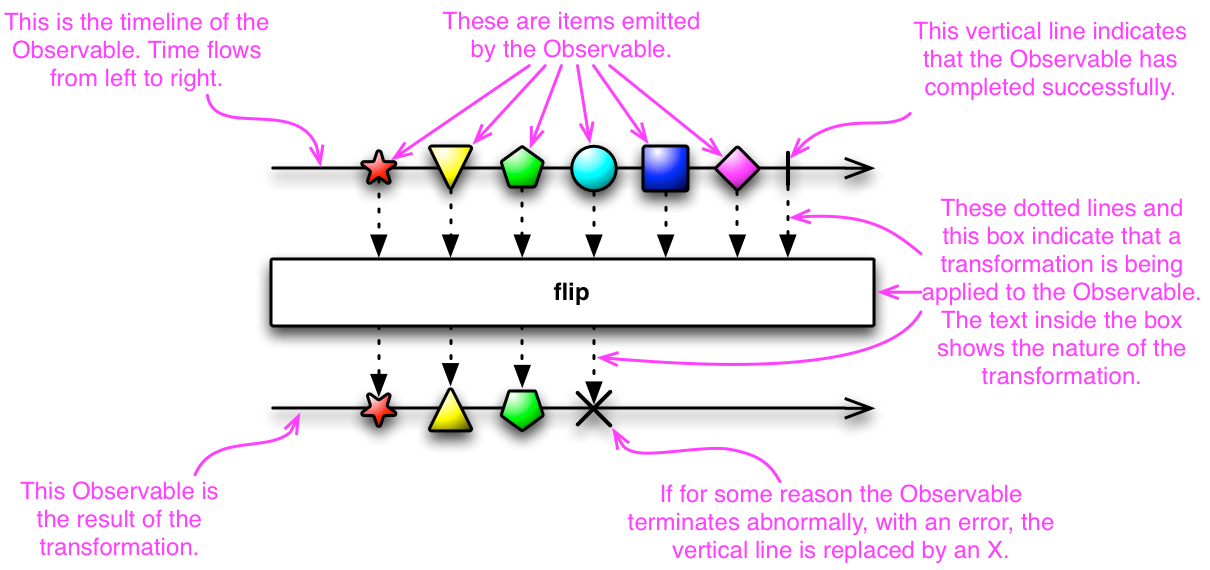
**Observable**

**概述**

在ReactiveX中，一个观察者(Observer)订阅一个可观察对象(Observable)。观察者对Observable发射的数据或数据序列作出响应。这种模式可以极大地简化并发操作，因为它创建了一个处于待命状态的观察者哨兵，在未来某个时刻响应Observable的通知，不需要阻塞等待Observable发射数据。

这篇文章会解释什么是响应式编程模式(reactive pattern)，以及什么是可观察对象(Observables)和观察者(observers)，其它几篇文章会展示如何用操作符组合和改变Observable的行为。



#### 相关参考：

* [Single](https://github.com/mcxiaoke/RxDocs/blob/master/Single.md) - 一个特殊的Observable，只发射单个数据。

## 背景知识

在很多软件编程任务中，或多或少你都会期望你写的代码能按照编写的顺序，一次一个的顺序执行和完成。但是在ReactiveX中，很多指令可能是并行执行的，之后他们的执行结果才会被观察者捕获，顺序是不确定的。为达到这个目的，你定义一种获取和变换数据的机制，而不是调用一个方法。在这种机制下，存在一个可观察对象(Observable)，观察者(Observer)订阅(Subscribe)它，当数据就绪时，之前定义的机制就会分发数据给一直处于等待状态的观察者哨兵。

这种方法的优点是，如果你有大量的任务要处理，它们互相之间没有依赖关系。你可以同时开始执行它们，不用等待一个完成再开始下一个（用这种方式，你的整个任务队列能耗费的最长时间，不会超过任务里最耗时的那个）。

有很多术语可用于描述这种异步编程和设计模式，在在本文里我们使用这些术语：**一个观察者订阅一个可观察对象** (An observer subscribes to an Observable)。通过调用观察者的方法，Observable发射数据或通知给它的观察者。

在其它的文档和场景里，有时我们也将**Observer**叫做Subscriber、Watcher、Reactor。这个模型通常被称作Reactor模式。

## 创建观察者

本文使用类似于Groovy的伪代码举例，但是ReactiveX有多种语言的实现。

普通的方法调用（不是某种异步方法，也不是Rx中的并行调用），流程通常是这样的：

1. 调用某一个方法
2. 用一个变量保存方法返回的结果
3. 使用这个变量和它的新值做些有用的事

用代码描述就是：

// make the call, assign its return value to `returnVal`

returnVal = someMethod(itsParameters);

// do something useful with returnVal

在异步模型中流程更像这样的：

1. 定义一个方法，这个方法拿着某个异步调用的返回值做一些有用的事情。这个方法是观察者的一部分。
2. 将这个异步调用本身定义为一个Observable
3. 观察者通过订阅(Subscribe)操作关联到那个Observable
4. 继续你的业务逻辑，等方法返回时，Observable会发射结果，观察者的方法会开始处理结果或结果集

用代码描述就是：

// defines, but does not invoke, the Subscriber's onNext handler

// (in this example, the observer is very simple and has only an onNext handler)

def myOnNext = { it -> do something useful with it };

// defines, but does not invoke, the Observable

def myObservable = someObservable(itsParameters);

// subscribes the Subscriber to the Observable, and invokes the Observable

myObservable.subscribe(myOnNext);

// go on about my business

### 回调方法 (onNext, onCompleted, onError)

Subscribe方法用于将观察者连接到Observable，你的观察者需要实现以下方法的一个子集：

* **onNext(T item)**

Observable调用这个方法发射数据，方法的参数就是Observable发射的数据，这个方法可能会被调用多次，取决于你的实现。

* **onError(Exception ex)**

当Observable遇到错误或者无法返回期望的数据时会调用这个方法，这个调用会终止Observable，后续不会再调用onNext和onCompleted，onError方法的参数是抛出的异常。

* **onComplete**

正常终止，如果没有遇到错误，Observable在最后一次调用onNext之后调用此方法。

根据Observable协议的定义，onNext可能会被调用零次或者很多次，最后会有一次onCompleted或onError调用（不会同时），传递数据给onNext通常被称作发射，onCompleted和onError被称作通知。

下面是一个更完整的例子：

def myOnNext = { item -> /\* do something useful with item \*/ };

def myError = { throwable -> /\* react sensibly to a failed call \*/ };

def myComplete = { /\* clean up after the final response \*/ };

def myObservable = someMethod(itsParameters);

myObservable.subscribe(myOnNext, myError, myComplete);

// go on about my business

### 取消订阅 (Unsubscribing)

在一些ReactiveX实现中，有一个特殊的观察者接口Subscriber，它有一个unsubscribe方法。调用这个方法表示你不关心当前订阅的Observable了，因此Observable可以选择停止发射新的数据项（如果没有其它观察者订阅）。

取消订阅的结果会传递给这个Observable的操作符链，而且会导致这个链条上的每个环节都停止发射数据项。这些并不保证会立即发生，然而，对一个Observable来说，即使没有观察者了，它也可以在一个while循环中继续生成并尝试发射数据项。

### 关于命名约定

ReactiveX的每种特定语言的实现都有自己的命名偏好，虽然不同的实现之间有很多共同点，但并不存在一个统一的命名标准。

而且，在某些场景中，一些名字有不同的隐含意义，或者在某些语言看来比较怪异。

例如，有一个onEvent命名模式(onNext, onCompleted, onError)，在一些场景中，这些名字可能意味着事件处理器已经注册。然而在ReactiveX里，他们是事件处理器的名字。

## Observables的"热"和"冷"

Observable什么时候开始发射数据序列？这取决于Observable的实现，一个"热"的Observable可能一创建完就开始发射数据，因此所有后续订阅它的观察者可能从序列中间的某个位置开始接受数据（有一些数据错过了）。一个"冷"的Observable会一直等待，直到有观察者订阅它才开始发射数据，因此这个观察者可以确保会收到整个数据序列。

在一些ReactiveX实现里，还存在一种被称作Connectable的Observable，不管有没有观察者订阅它，这种Observable都不会开始发射数据，除非Connect方法被调用。

## 用操作符组合Observable

对于ReactiveX来说，Observable和Observer仅仅是个开始，它们本身不过是标准观察者模式的一些轻量级扩展，目的是为了更好的处理事件序列。

ReactiveX真正强大的地方在于它的操作符，操作符让你可以变换、组合、操纵和处理Observable发射的数据。

Rx的操作符让你可以用声明式的风格组合异步操作序列，它拥有回调的所有效率优势，同时又避免了典型的异步系统中嵌套回调的缺点。

下面是常用的操作符列表：

1. [创建操作](https://github.com/mcxiaoke/RxDocs/blob/master/operators/Creating-Observables.md) Create, Defer, Empty/Never/Throw, From, Interval, Just, Range, Repeat, Start, Timer
2. [变换操作](https://github.com/mcxiaoke/RxDocs/blob/master/operators/Transforming-Observables.md) Buffer, FlatMap, GroupBy, Map, Scan和Window
3. [过滤操作](https://github.com/mcxiaoke/RxDocs/blob/master/operators/Filtering-Observables.md) Debounce, Distinct, ElementAt, Filter, First, IgnoreElements, Last, Sample, Skip, SkipLast, Take, TakeLast
4. [组合操作](https://github.com/mcxiaoke/RxDocs/blob/master/operators/Combining-Observables.md) And/Then/When, CombineLatest, Join, Merge, StartWith, Switch, Zip
5. [错误处理](https://github.com/mcxiaoke/RxDocs/blob/master/operators/Error-Handling-Operators.md) Catch和Retry
6. [辅助操作](https://github.com/mcxiaoke/RxDocs/blob/master/operators/Observable-Utility-Operators.md) Delay, Do, Materialize/Dematerialize, ObserveOn, Serialize, Subscribe, SubscribeOn, TimeInterval, Timeout, Timestamp, Using
7. [条件和布尔操作](https://github.com/mcxiaoke/RxDocs/blob/master/operators/Conditional-and-Boolean-Operators.md) All, Amb, Contains, DefaultIfEmpty, SequenceEqual, SkipUntil, SkipWhile, TakeUntil, TakeWhile
8. [算术和集合操作](https://github.com/mcxiaoke/RxDocs/blob/master/operators/Mathematical-and-Aggregate-Operators.md) Average, Concat, Count, Max, Min, Reduce, Sum
9. [转换操作](https://github.com/mcxiaoke/RxDocs/blob/master/operators/To.md) To
10. [连接操作](https://github.com/mcxiaoke/RxDocs/blob/master/operators/Connectable-Observable-Operators.md) Connect, Publish, RefCount, Replay
11. [反压操作](https://github.com/mcxiaoke/RxDocs/blob/master/topics/Backpressure.md)，用于增加特殊的流程控制策略的操作符

这些操作符并不全都是ReactiveX的核心组成部分，有一些是语言特定的实现或可选的模块。

## RxJava

在RxJava中，一个实现了\_Observer\_接口的对象可以订阅(subscribe)一个\_Observable\_ 类的实例。订阅者(subscriber)对Observable发射(emit)的任何数据或数据序列作出响应。这种模式简化了并发操作，因为它不需要阻塞等待Observable发射数据，而是创建了一个处于待命状态的观察者哨兵，哨兵在未来某个时刻响应Observable的通知。