# ReactiveX

<http://reactivex.io/intro.html>

## Rx介绍

### ReactiveX的历史

ReactiveX是Reactive Extensions的缩写，一般简写为Rx，最初是LINQ的一个扩展，由微软的架构师Erik Meijer领导的团队开发，在2012年11月开源，Rx是一个编程模型，目标是提供一致的编程接口，帮助开发者更方便的处理异步数据流，Rx库支持.NET、JavaScript和C++，Rx近几年越来越流行了，现在已经支持几乎全部的流行编程语言了，Rx的大部分语言库由ReactiveX这个组织负责维护，比较流行的有RxJava/RxJS/Rx.NET，社区网站是 [reactivex.io](http://reactivex.io/)。

**什么是ReactiveX**

微软给的定义是，Rx是一个函数库，让开发者可以利用可观察序列和LINQ风格查询操作符来编写异步和基于事件的程序，使用Rx，开发者可以用Observables表示异步数据流，用LINQ操作符查询异步数据流， 用Schedulers参数化异步数据流的并发处理，Rx可以这样定义：Rx = Observables + LINQ + Schedulers。

ReactiveX.io给的定义是，Rx是一个使用可观察数据流进行异步编程的编程接口，ReactiveX结合了观察者模式、迭代器模式和函数式编程的精华。

**ReactiveX的应用**

很多公司都在使用ReactiveX，例如Microsoft、Netflix、Github、Trello、SoundCloud。

**ReactiveX宣言**

ReactiveX不仅仅是一个编程接口，它是一种编程思想的突破，它影响了许多其它的程序库和框架以及编程语言。

## Rx模式

### 使用观察者模式

* 创建：Rx可以方便的创建事件流和数据流
* 组合：Rx使用查询式的操作符组合和变换数据流
* 监听：Rx可以订阅任何可观察的数据流并执行操作

### 简化代码

* 函数式风格：对可观察数据流使用无副作用的输入输出函数，避免了程序里错综复杂的状态
* 简化代码：Rx的操作符通通常可以将复杂的难题简化为很少的几行代码
* 异步错误处理：传统的try/catch没办法处理异步计算，Rx提供了合适的错误处理机制
* 轻松使用并发：Rx的Observables和Schedulers让开发者可以摆脱底层的线程同步和各种并发问题
* **使用Observable的优势**
* Rx扩展了观察者模式用于支持数据和事件序列，添加了一些操作符，它让你可以声明式的组合这些序列，而无需关注底层的实现：如线程、同步、线程安全、并发数据结构和非阻塞IO。
* Observable通过使用最佳的方式访问异步数据序列填补了这个间隙

|  | **单个数据** | **多个数据** |
| --- | --- | --- |
| 同步 | T getData() | Iterable<T> getData() |
| 异步 | Future<T> getData() | Observable<T> getData() |

* Rx的Observable模型让你可以像使用集合数据一样操作异步事件流，对异步事件流使用各种简单、可组合的操作。

#### Observable可组合

对于单层的异步操作来说，Java中Future对象的处理方式是非常简单有效的，但是一旦涉及到嵌套，它们就开始变得异常繁琐和复杂。使用Future很难很好的组合带条件的异步执行流程（考虑到运行时各种潜在的问题，甚至可以说是不可能的），当然，要想实现还是可以做到的，但是非常困难，或许你可以用Future.get()，但这样做，异步执行的优势就完全没有了。从另一方面说，Rx的Observable一开始就是为组合异步数据流准备的。

#### Observable更灵活

Rx的Observable不仅支持处理单独的标量值（就像Future可以做的），也支持数据序列，甚至是无穷的数据流。Observable是一个抽象概念，适用于任何场景。Observable拥有它的近亲Iterable的全部优雅与灵活。

Observable是异步的双向push，Iterable是同步的单向pull，对比：

| **事件** | **Iterable(pull)** | **Observable(push)** |
| --- | --- | --- |
| 获取数据 | T next() | onNext(T) |
| 异常处理 | throws Exception | onError(Exception) |
| 任务完成 | !hasNext() | onCompleted() |

#### Observable无偏见

Rx对于对于并发性或异步性没有任何特殊的偏好，Observable可以用任何方式实现，线程池、事件循环、非阻塞IO、Actor模式，任何满足你的需求的，你擅长或偏好的方式都可以。无论你选择怎样实现它，无论底层实现是阻塞的还是非阻塞的，客户端代码将所有与Observable的交互都当做是异步的。

**Observable是如何实现的？**

public Observable<data> getData();

* 它能与调用者在同一线程同步执行吗？
* 它能异步地在单独的线程执行吗？
* 它会将工作分发到多个线程，返回数据的顺序是任意的吗？
* 它使用Actor模式而不是线程池吗？
* 它使用NIO和事件循环执行异步网络访问吗？
* 它使用事件循环将工作线程从回调线程分离出来吗？

从Observer的视角看，这些都无所谓，重要的是：使用Rx，你可以改变你的观念，你可以在完全不影响Observable程序库使用者的情况下，彻底的改变Observable的底层实现。

#### 使用回调存在很多问题

回调在不阻塞任何事情的情况下，解决了Future.get()过早阻塞的问题。由于响应结果一旦就绪Callback就会被调用，它们天生就是高效率的。不过，就像使用Future一样，对于单层的异步执行来说，回调很容易使用，对于嵌套的异步组合，它们显得非常笨拙。

#### Rx是一个多语言的实现

Rx在大量的编程语言中都有实现，并尊重实现语言的风格，而且更多的实现正在飞速增加。

#### 响应式编程

Rx提供了一系列的操作符，你可以使用它们来过滤(filter)、选择(select)、变换(transform)、结合(combine)和组合(compose)多个Observable，这些操作符让执行和复合变得非常高效。

你可以把Observable当做Iterable的推送方式的等价物，使用Iterable，消费者从生产者那拉取数据，线程阻塞直至数据准备好。使用Observable，在数据准备好时，生产者将数据推送给消费者。数据可以同步或异步的到达，这种方式更灵活。

下面的例子展示了相似的高阶函数在Iterable和Observable上的应用

// Iterable

getDataFromLocalMemory()

.skip(10)

.take(5)

.map({ s -> return s + " transformed" })

.forEach({ println "next => " + it })

// Observable

getDataFromNetwork()

.skip(10)

.take(5)

.map({ s -> return s + " transformed" })

.subscribe({ println "onNext => " + it })

Observable类型给GOF的观察者模式添加了两种缺少的语义，这样就和Iterable类型中可用的操作一致了：

1. 生产者可以发信号给消费者，通知它没有更多数据可用了（对于Iterable，一个for循环正常完成表示没有数据了；对于Observable，就是调用观察者的onCompleted方法）
2. 生产者可以发信号给消费者，通知它遇到了一个错误（对于Iterable，迭代过程中发生错误会抛出异常；对于Observable，就是调用观察者(Observer)的onError方法）

有了这两种功能，Rx就能使Observable与Iterable保持一致了，唯一的不同是数据流的方向。任何对Iterable的操作，你都可以对Observable使用。

## 名词定义

这里给出一些名词的翻译

* Reactive 直译为反应性的，有活性的，根据上下文一般翻译为反应式、响应式
* Iterable 可迭代对象，支持以迭代器的形式遍历，许多语言中都存在这个概念
* Observable 可观察对象，在Rx中定义为更强大的Iterable，在观察者模式中是被观察的对象，一旦数据产生或发生变化，会通过某种方式通知观察者或订阅者
* Observer 观察者对象，监听Observable发射的数据并做出响应，Subscriber是它的一个特殊实现
* emit 直译为发射，发布，发出，含义是Observable在数据产生或变化时发送通知给Observer，调用Observer对应的方法，文章里一律译为发射
* items 直译为项目，条目，在Rx里是指Observable发射的数据项，文章里一律译为数据，数据项