理解 RxJava 的线程模型（上）

2016-08-02 安卓应用频道

(点击上方公众号，可快速关注)

> 来源：鸟窝（@colobu ）
> 链接：colobu.com/2016/07/25/understanding-rxjava-thread-model/

目录

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。

Netflix参考微软的Reactive Extensions创建了Java的实现RxJava，主要是为了简化服务器端的并发。2013年二月份,Ben Christensen 和 Jafar Husain发在Netflix技术博客的一篇文章第一次向世界展示了RxJava。

RxJava也在Android开发中得到广泛的应用。

> ReactiveX
> An API for asynchronous programming with observable streams.
> A combination of the best ideas from the Observer pattern, the Iterator pattern,

> and functional programming.

虽然RxJava是为异步编程实现的库，但是如果不清楚它的使用，或者错误地使用了它的线程调度，反而不能很好的利用它的异步编程提到系统的处理速度。本文通过实例演示错误的RxJava的使用，解释RxJava的线程调度模型，主要介绍Scheduler、observeOn和subscribeOn的使用。

本文中的例子以并发发送http request请求为基础，通过性能检验RxJava的线程调度。

一、第一个例子，性能超好？

我们首先看第一个例子:

```
public static void testRxJavaWithoutBlocking(int count) throws Exception {
    CountDownLatch finishedLatch = new CountDownLatch(1);
    long t = System.nanoTime();
    Observable.range(0, count).map(i -> {
        //System.out.println("A:" + Thread.currentThread().getName());
        return 200;
    }).subscribe(statusCode -> {
        //System.out.println("B:" + Thread.currentThread().getName());
    }, error -> {
    }, () -> {
        finishedLatch.countDown();
    });
    finishedLatch.await();
    t = (System.nanoTime() - t) / 1000000; //ms
    System.out.println("RxJavaWithoutBlocking TPS: " + count * 1000 / t);
}
```

这个例子是一个基本的RxJava的使用，利用Range创建一个Observable, subscriber处理接收的数据。因为整个逻辑没有阻塞，程序运行起来很快，
输出结果为：

RxJavaWithoutBlocking TPS: **7692307** 。

二 、加上业务的模拟，性能超差

上面的例子是一个理想化的程序，没雨任何阻塞。我们模拟一下实际的应用，加上业务处

理。

业务逻辑是发送一个http的请求，httpserver是一个模拟器，针对每个请求有30毫秒的延迟。subscriber统计请求结果：

```
public static void testRxJavaWithBlocking(int count) throws Exception {
    URL url = new URL("http://127.0.0.1:8999/");
    CountDownLatch finishedLatch = new CountDownLatch(1);
    long t = System.nanoTime();
    Observable.range(0, count).map(i -> {
      try {
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        int responseCode = conn.getResponseCode();
        BufferedReader in = new BufferedReader(new InputStreamReader(conn.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null) {
          //response.append(inputLine);
        }
        in.close();
        return responseCode;
      } catch (Exception ex) {
        return -1;
      }
    }).subscribe(statusCode -> {
    }, error -> {
    }, () -> {
      finishedLatch.countDown();
    });
    finishedLatch.await();
    t = (System.nanoTime() - t) / 1000000; //ms
    System.out.println("RxJavaWithBlocking TPS: " + count * 1000 / t);
  }
```

运行结果如下：

RxJavaWithBlocking TPS: **29**。

@#¥%%……&！

性能怎么突降呢，第一个例子看起来性能超好啊，http server只增加了一个30毫秒的延迟，导致这个方法每秒只能处理29个请求。

如果我们估算一下， 29*30= 870 毫秒，大约1秒，正好和单个线程发送处理所有的请求的TPS差不多。
后面我们也会看到，实际的确是一个线程处理的，你可以在代码中加入

三、加上调度器，不起作用？

如果你对subscribeOn和observeOn方法有些印象的话，可能会尝试使用调度器去解决：

```
public static void testRxJavaWithBlocking(int count) throws Exception {
    URL url = new URL("http://127.0.0.1:8999/");
    CountDownLatch finishedLatch = new CountDownLatch(1);
    long t = System.nanoTime();
    Observable.range(0, count).map(i -> {
      try {
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        int responseCode = conn.getResponseCode();
        BufferedReader in = new BufferedReader(new InputStreamReader(conn.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null) {
          //response.append(inputLine);
        }
        in.close();
        return responseCode;
      } catch (Exception ex) {
        return -1;
      }
    }).subscribeOn(Schedulers.io()).observeOn(Schedulers.computation()).subscribe(statusCode -> {
    }, error -> {
    }, () -> {
      finishedLatch.countDown();
    });
    finishedLatch.await();
    t = (System.nanoTime() - t) / 1000000; //ms
    System.out.println("RxJavaWithBlocking TPS: " + count * 1000 / t);
```

```
    }
```

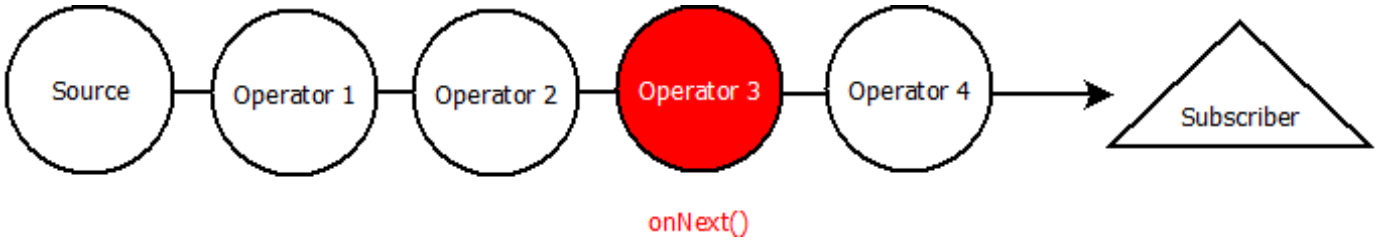
加上.subscribeOn(Schedulers.io()).observeOn(Schedulers.computation())看一下性能：

RxJavaWithBlocking TPS: **30**。

性能没有改观，是时候了解一下RxJava线程调度的问题了。

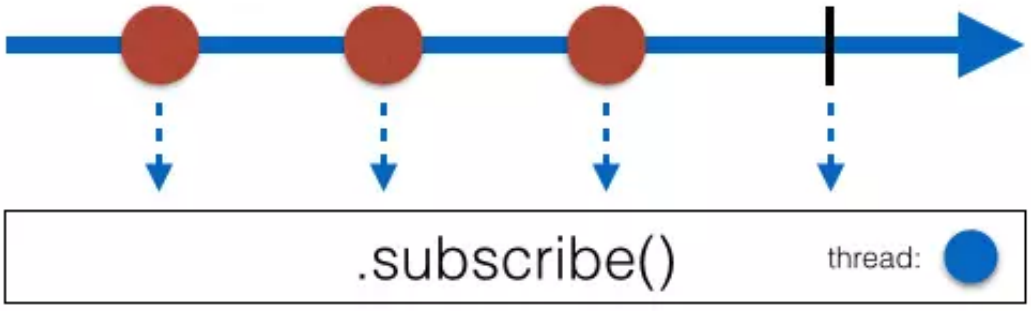
四、RxJava的线程模型

首先，依照Observable Contract, onNext是顺序执行的，不会同时由多个线程并发执行。



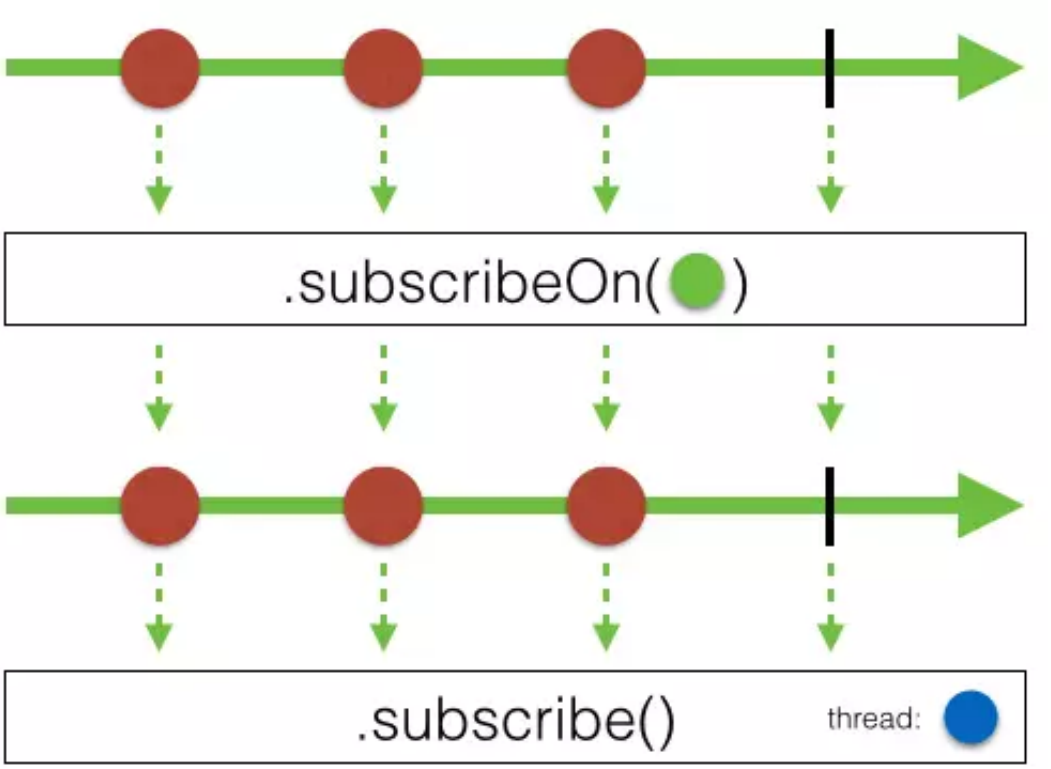
图片来自 http://tomstechnicalblog.blogspot.com/2016/02/rxjava-understanding-observeon-and.html

默认情况下，它是在调用subscribe方法的那个线程中执行的。如第一个例子和第二个例子，Rx的操作和消息接收处理都是在同一个线程中执行的。一旦由阻塞，比如第二个例子，久会导致这个线程被阻塞，吞吐量下降。



图片来自 https://medium.com/@diolor/observe-in-the-correct-thread-1939bb9bb9d2

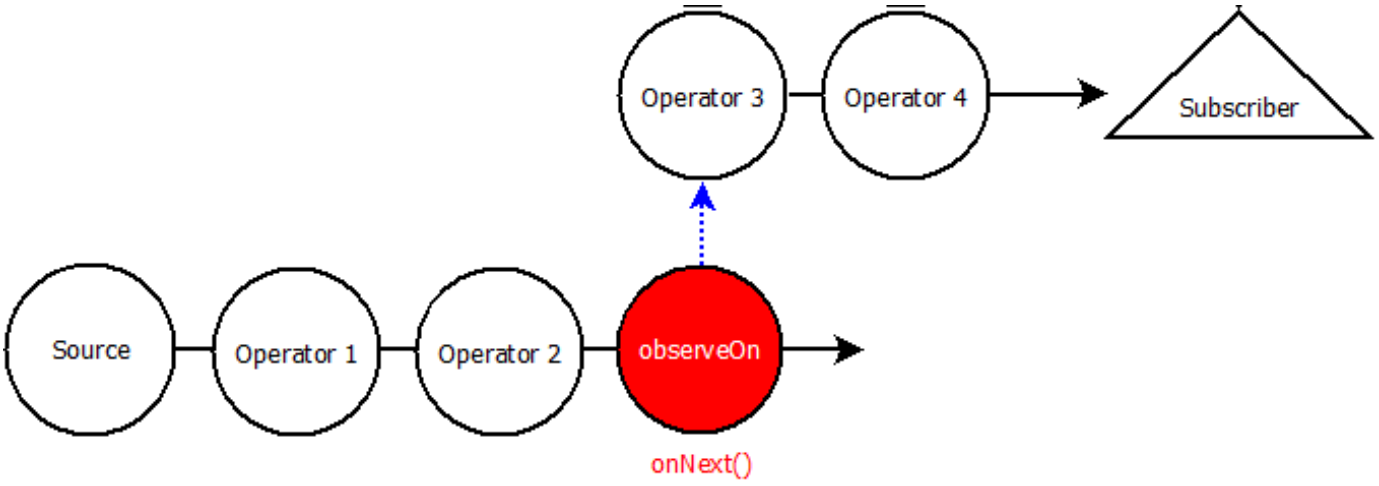
但是subscribeOn可以改变Observable的运行线程。



图片来自 https://medium.com/@diolor/observe-in-the-correct-thread-1939bb9bb9d2

上图中可以看到，如果你使用了subscribeOn方法，则Rx的运行将会切换到另外的线程上，而不是默认的调用线程。
需要注意的是，如果在Observable链中调用了多个subscribeOn方法，无论调用点在哪里，Observable链只会使用第一个subscribeOn指定的调度器，正所谓”一见倾情”。
但是onNext还是顺序执行的，所以第二个例子的性能依然低下。

observeOn可以中途改变Observable链的线程。前面说了，subscribeOn方法改变的源Observable的整个的运行线程，要想中途切换线程，就需要observeOn方法。

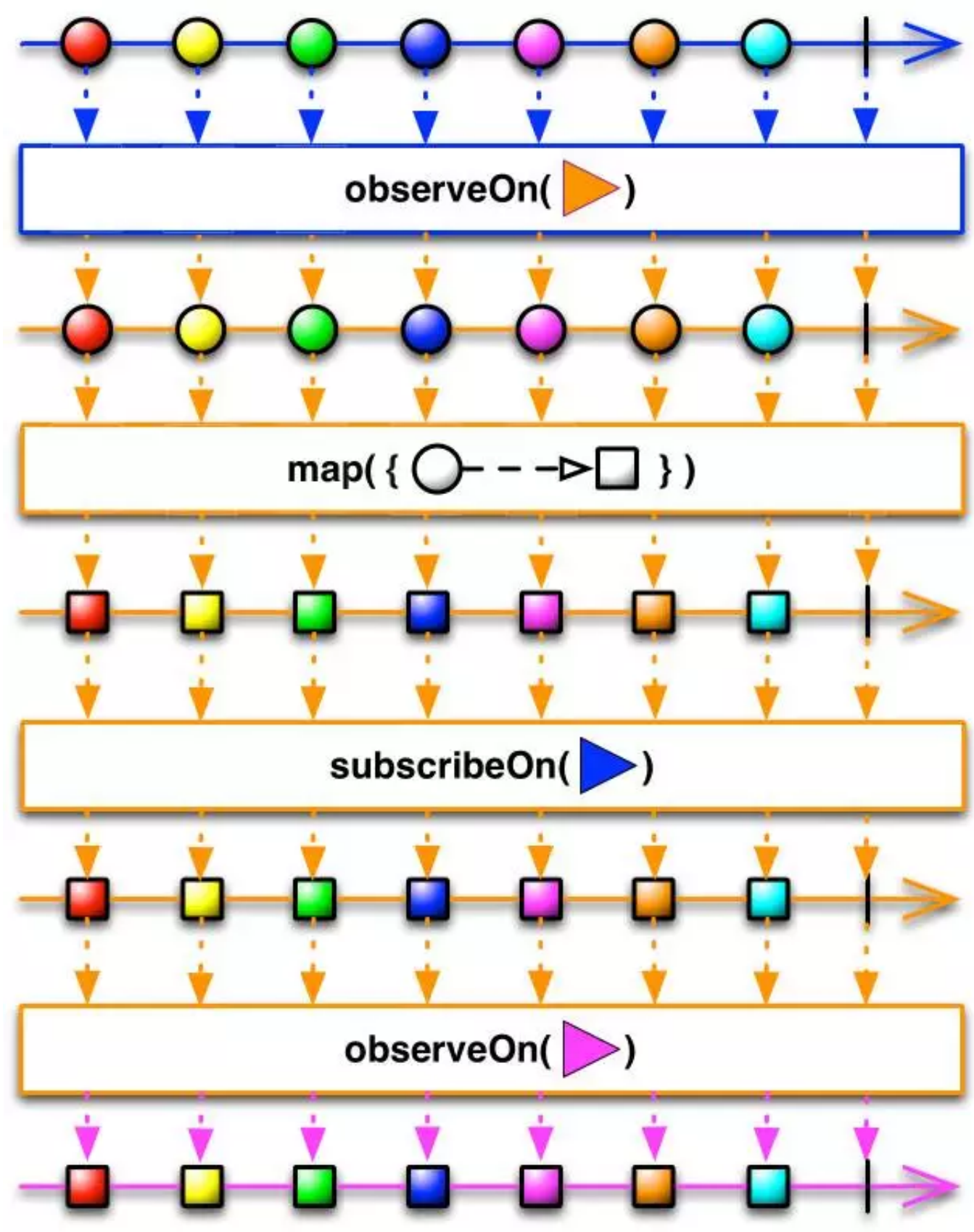


图片来自 http://tomstechnicalblog.blogspot.com/2016/02/rxjava-understanding-observeon-and.html

官方的一个简略晦涩的解释如下：

> The SubscribeOn operator changes this behavior by specifying a different Scheduler on which the Observable should operate. The ObserveOn operator specifies a different Scheduler that the Observable will use to send notifications to its observers.

一图胜千言：



图片来自 http://reactivex.io

注意箭头的颜色和横轴的颜色，不同的颜色代表不同的线程。

接下文