**[从零开始的RxJava2.0教程(二)操作符](http://blog.csdn.net/qq_35064774/article/details/53057359)**

# 2. 准备工作

假设我的 Flowable 发射的是一个列表，接收者要把列表内容依次输出。根据上一篇blog的内容，你可以会写出这样的代码:

List<Integer> list = new ArrayList<>();

list.add(10);

list.add(1);

list.add(5);

Flowable.just(list)

.subscribe(new Consumer<List<Integer>>() {

@Override

public void accept(List<Integer> list) throws Exception {

for (Integer integer : list)

System.out.println(integer);

}

});

这样的代码当然是不能容忍的，因为上面的代码使我们丧失了变化数据流的能力。一旦我们想要更改列表中的每一个数据，只能在订阅者中做。

当然我们可以使用map来中间处理，但是这样做也需要遍历整个list。

万幸，RxJava 2.0 提供了fromIterable方法，可以接收一个 Iterable 容器作为输入，每次发射一个元素。

List<Integer> list = new ArrayList<>();

list.add(10);

list.add(1);

list.add(5);

Flowable.fromIterable(list)

.subscribe(num -> System.out.println(num));

我们把fromX用到这个例子中来。

List<Integer> list = new ArrayList<>();

list.add(10);

list.add(1);

list.add(5);

Flowable.just(list)

.subscribe(nums -> {

Observable.fromIterable(nums)

.subscribe(num -> System.out.println(num));

});

虽然去掉了 for 循环，但是代码依然看起来很乱。嵌套了两层，它会破坏某些我们现在还没有讲到的RxJava的特性。

# 3. 改进

救星来了,他就是 flatMap()。   
Flowable.flatMap 可以把一个 Flowable 转换成另一个 Flowable :

List<Integer> list = new ArrayList<>();

list.add(10);

list.add(1);

list.add(5);

Flowable.just(list)

.flatMap(new Function<List<Integer>, Publisher<Integer>>() {

@Override

public Publisher<Integer> apply(List<Integer> integers) throws Exception {

return Flowable.fromIterable(integers);

}

})

.subscribe(new Consumer<Integer>() {

@Override

public void accept(Integer integer) throws Exception {

System.out.println(integer);

}

});

和 map 不同之处在于 flatMap 返回的是一个 Flowable 对象。这正是我们想要的，我们可以把从List发射出来的一个一个的元素发射出去。

# 4. 更多操作符

目前为止，我们已经接触了两个操作符，RxJava中还有更多的操作符。

如果我们想要订阅者只能收到大于5的数据，那么你可以这样做:

Flowable.fromArray(1, 20, 5, 0, -1, 8)

.filter(new Predicate<Integer>() {

@Override

public boolean test(Integer integer) throws Exception {

return integer.intValue() > 5;

}

})

.subscribe(new Consumer<Integer>() {

@Override

public void accept(Integer integer) throws Exception {

System.out.println(integer);

}

});

filter 是用于过滤数据的，返回false表示拦截此数据。

如果我们只想要2个数据:

Flowable.fromArray(1, 2, 3, 4)

.take(2)

.subscribe(new Consumer<Integer>() {

@Override

public void accept(Integer integer) throws Exception {

System.out.println(integer);

}

});

take 用于指定订阅者最多收到多少数据。

如果我们想在订阅者接收到数据前干点事情，比如记录日志:

Flowable.just(1, 2, 3)

.doOnNext(new Consumer<Integer>() {

@Override

public void accept(Integer integer) throws Exception {

System.out.println("保存:" + integer);

}

})

.subscribe(new Consumer<Integer>() {

@Override

public void accept(Integer integer) throws Exception {

System.out.println(integer);

}

});

doOnNext 允许我们在每次输出一个元素之前做一些额外的事情。