



链滴

给初学者的 RxJava2.0 教程

作者: [cyd](#)

原文链接: <https://ld246.com/article/1482074104435>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

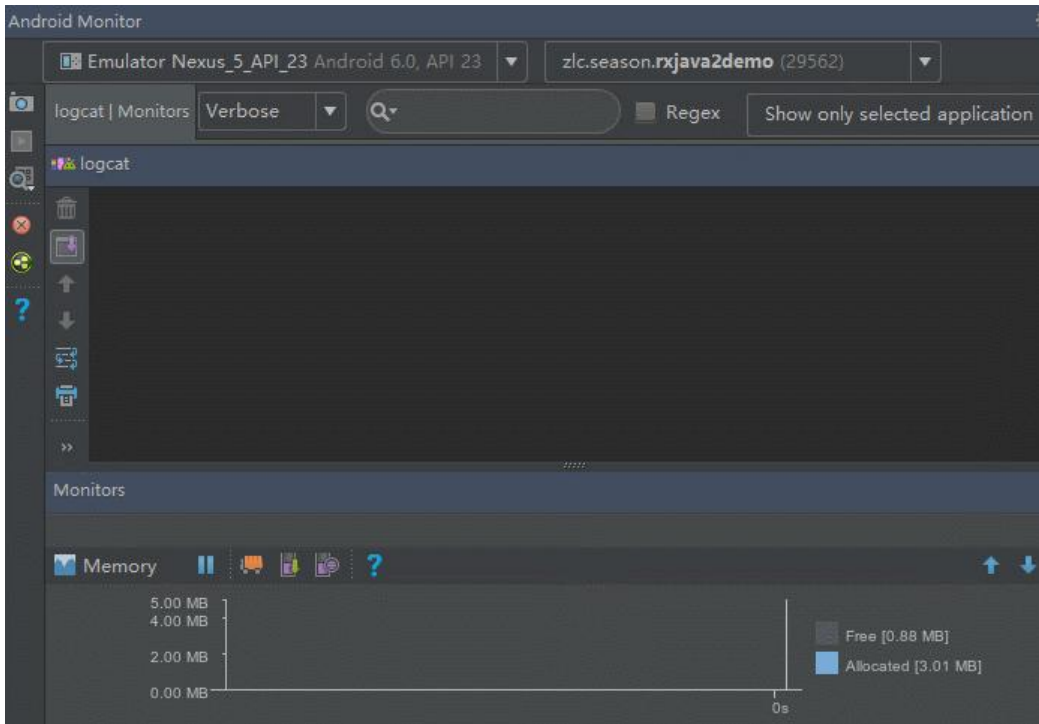
可能很多看过其他人写的文章的朋友都会觉得只有Flowable才能解决Backpressure, 所以大家对这个Flowable都抱有很大的期许, 其实呐, 你们毕竟图样图森破, 今天我们先抛开Flowable, 仅仅依靠我们自己的双手和智慧, 来看看我们如何去治理BackPressure, 通过本节的学习之后我们再来看Flowable, 会发现它其实并没有想象中那么牛叉, 它只是被其他人过度神化了.

看代码

```
Observable.create(new ObservableOnSubscribe() {
    @Override
    public void subscribe(ObservableEmitter emitter) throws Exception {
        for (int i = 0; ; i++) { //无限循环发送事件
            emitter.onNext(i);
        }
    }
}).subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Consumer() {
        @Override
        public void accept(Integer integer) throws Exception {
            Log.d(TAG, "" + integer);
        }
    });
```

```
Observable.create(new ObservableOnSubscribe() {
    @Override
    public void subscribe(ObservableEmitter emitter) throws Exception {
        for (int i = 0; ; i++) {
            emitter.onNext(i);
        }
    }
}).subscribeOn(Schedulers.io())
    .filter(new Predicate() {
        @Override
        public boolean test(Integer integer) throws Exception {
            return integer % 10 == 0;
        }
    })
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Consumer() {
        @Override
        public void accept(Integer integer) throws Exception {
            Log.d(TAG, "" + integer);
        }
    });
```

在这段代码中我们增加了一个filter, 只允许能被10整除的事件通过, 再来看看运行结果:



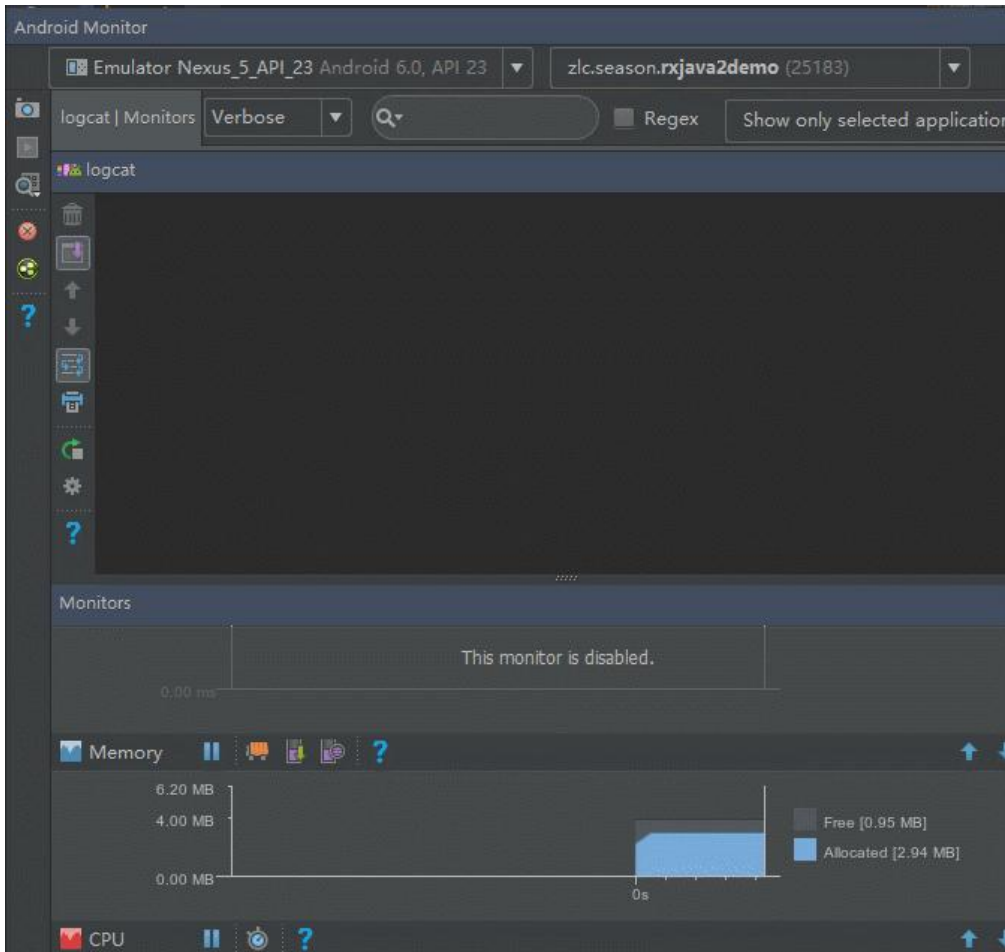
filter.gif

可以看到, 虽然内存依然在增长, 但是增长速度相比之前, 已经减少了太多了, 至少在我录完GIF之前还有爆掉内存, 大家可以试着改成能被100整除试试.

可以看到, 通过减少进入水缸的事件数量的确可以缓解BackPressure, 但是力度还不够, 我们再来看一段代码:

```
Observable.create(new ObservableOnSubscribe() {
    @Override
    public void subscribe(ObservableEmitter emitter) throws Exception {
        for (int i = 0; ; i++) {
            emitter.onNext(i);
        }
    }
}).subscribeOn(Schedulers.io())
    .sample(2, TimeUnit.SECONDS) //sample取样
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Consumer() {
        @Override
        public void accept(Integer integer) throws Exception {
            Log.d(TAG, "" + integer);
        }
    });
```

这里用了一个sample操作符, 简单做个介绍, 这个操作符每隔指定的时间就从上游中取出一个事件发给下游. 这里我们让它每隔2秒取一个事件给下游, 来看看这次的运行结果吧:



sample.gif

这次我们可以看到, 虽然上游仍然一直在不停的发事件, 但是我们只是每隔一定时间取一个放进水缸里, 并没有全部放进水缸里, 因此这次内存仅仅只占用了5M.

大家以后可以出去吹牛逼了: 我曾经通过技术手段去优化一个程序, 最终使得内存占用从300多M变成到5M. $\sim(\geq \nabla \leq) / \sim$ 啦啦啦

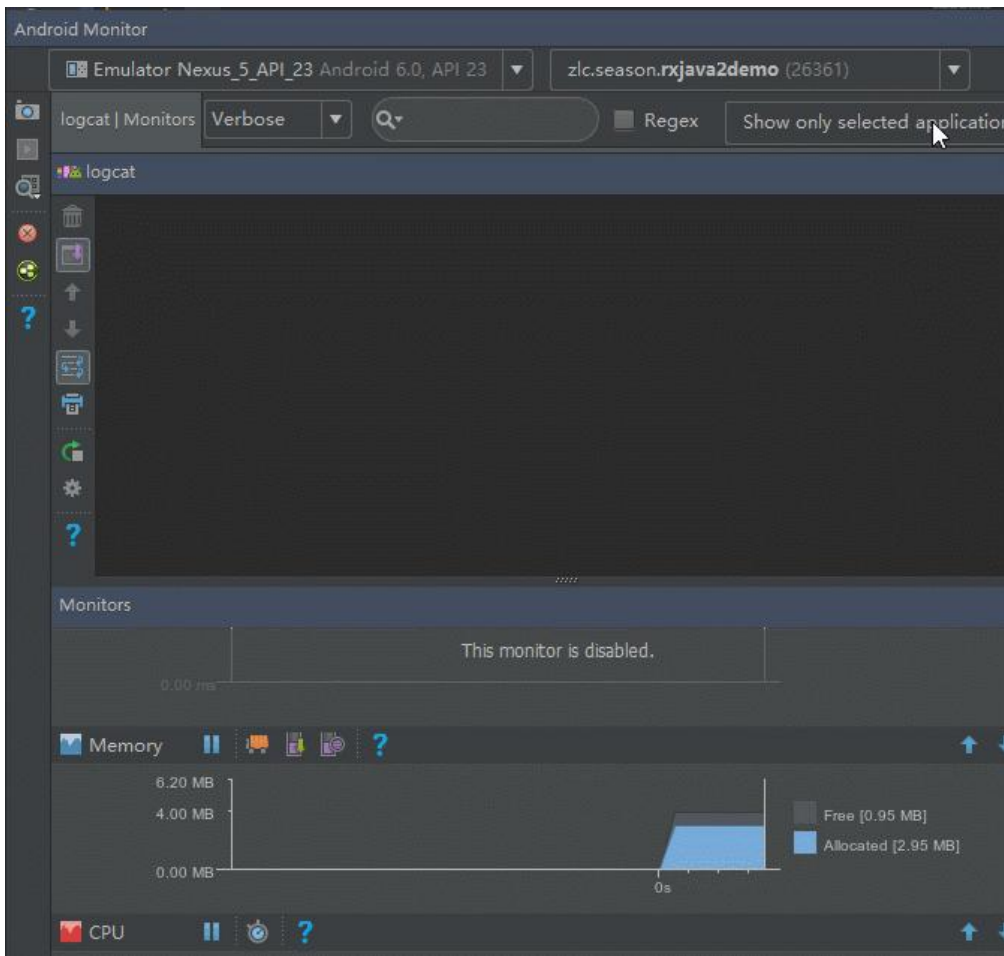
前面这两种方法归根到底其实就是减少放进水缸的事件的数量, 是以数量取胜, 但是这个方法有个缺点, 就是丢失了大部分的事件.

那么我们换一个角度来思考, 既然上游发送事件的速度太快, 那我们就适当减慢发送事件的速度, 从速度取胜, 听上去不错, 我们来试试:

```
Observable.create(new ObservableOnSubscribe() {
    @Override
    public void subscribe(ObservableEmitter emitter) throws Exception {
        for (int i = 0; ; i++) {
            emitter.onNext(i);
            Thread.sleep(2000); //每次发送完事件延时2秒
        }
    }
}).subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Consumer() {
        @Override
        public void accept(Integer integer) throws Exception {
```

```
        Log.d(TAG, "" + integer);
    }
});
```

这次我们让上游每次发送完事件后都延时了2秒, 来看看运行结果:



sleep.gif

完美！一切都是那么完美！

可以看到, 我们给上游加上延时了之后, 瞬间一头发情的公牛就变得跟只小绵羊一样, 如此温顺, 如此平, 如此平稳的内存线, 美妙极了. 而且事件也没有丢失, 上游通过适当的延时, 不但减缓了事件进入水缸的度, 也可以让下游有充足的时间从水缸里取出事件来处理, 这样一来, 就不至于导致大量的事件涌进水缸也就不会OOM啦.

到目前为止, 我们没有依靠任何其他的工具, 就轻易解决了Backpressure的问题.

因此我们总结一下, 治理Backpressure的办法就两种:

- 一是从数量上进行治理, 减少发送进水缸里的事件
- 二是从速度上进行治理, 减缓事件发送进水缸的速度

大家一定没忘记, 在上一节还有个Zip的例子, 这个例子也爆了我们的内存, 现学现用, 我们用刚学到的法来试试能不能惩奸除恶, 先来看看第一种办法.

先来减少进入水缸的事件的数量:

```

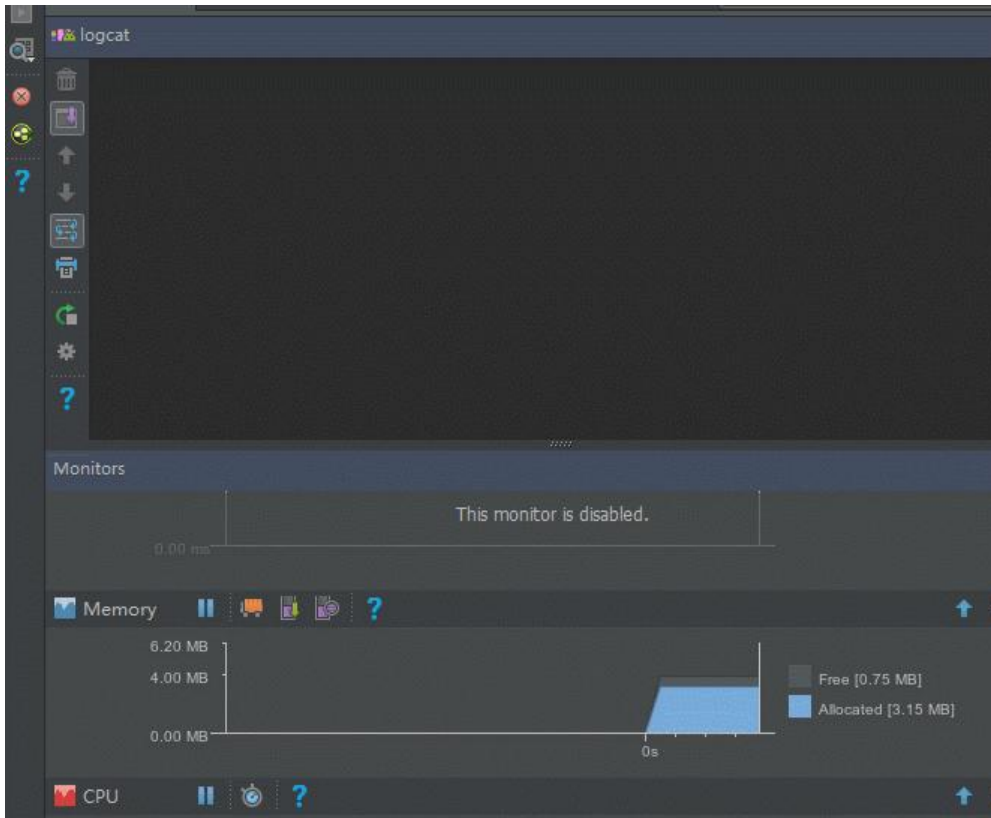
Observable observable1 = Observable.create(new ObservableOnSubscribe() {
    @Override
    public void subscribe(ObservableEmitter emitter) throws Exception {
        for (int i = 0; ; i++) {
            emitter.onNext(i);
        }
    }
}).subscribeOn(Schedulers.io()).sample(2, TimeUnit.SECONDS); //进行sample采样

Observable observable2 = Observable.create(new ObservableOnSubscribe() {
    @Override
    public void subscribe(ObservableEmitter emitter) throws Exception {
        emitter.onNext("A");
    }
}).subscribeOn(Schedulers.io());

Observable.zip(observable1, observable2, new BiFunction() {
    @Override
    public String apply(Integer integer, String s) throws Exception {
        return integer + s;
    }
}).observeOn(AndroidSchedulers.mainThread()).subscribe(new Consumer() {
    @Override
    public void accept(String s) throws Exception {
        Log.d(TAG, s);
    }
}, new Consumer() {
    @Override
    public void accept(Throwable throwable) throws Exception {
        Log.w(TAG, throwable);
    }
});

```

来试试运行结果吧:



zip_sample.gif

哈哈, 成功了吧, 再来用第二种办法试试.

这次我们来减缓速度:

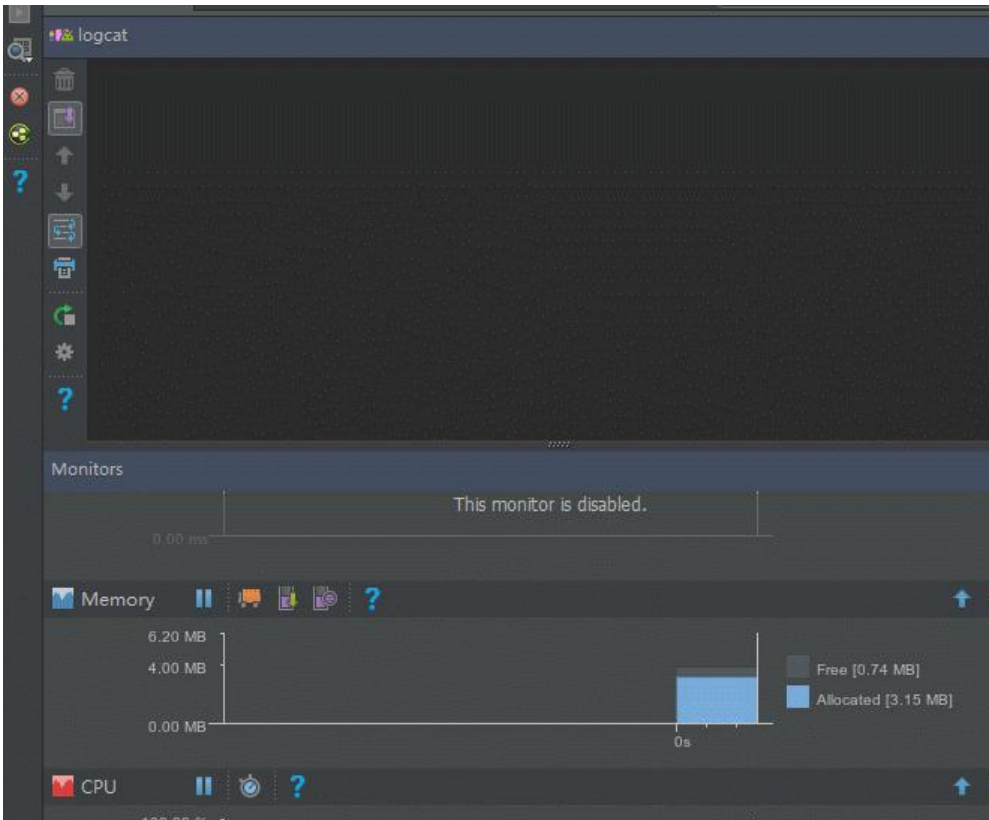
```
Observable observable1 = Observable.create(new ObservableOnSubscribe() {
    @Override
    public void subscribe(ObservableEmitter emitter) throws Exception {
        for (int i = 0; i << 10; i++) {
            emitter.onNext(i);
            Thread.sleep(2000); //发送事件之后延时2秒
        }
    }
}).subscribeOn(Schedulers.io());

Observable observable2 = Observable.create(new ObservableOnSubscribe() {
    @Override
    public void subscribe(ObservableEmitter emitter) throws Exception {
        emitter.onNext("A");
    }
}).subscribeOn(Schedulers.io());

Observable.zip(observable1, observable2, new BiFunction() {
    @Override
    public String apply(Integer integer, String s) throws Exception {
        return integer + s;
    }
}).observeOn(AndroidSchedulers.mainThread()).subscribe(new Consumer() {
    @Override
```

```
public void accept(String s) throws Exception {
    Log.d(TAG, s);
}
}, new Consumer() {
    @Override
    public void accept(Throwable throwable) throws Exception {
        Log.w(TAG, throwable);
    }
});
```

来看看运行结果吧:



zip_sleep.gif

果然也成功了, 这里只打印出了下游收到的事件, 所以只有一个. 如果你对这个结果看不懂, 请自觉掉头前面几篇文章.

通过本节的学习, 大家应该对如何处理 **Backpressure** 已经有了基本的认识了, 大家也可以看到, 我们并没有使用 **Flowable**, 就轻易的解决了背压的问题, 所以很多时候仔细去分析问题, 找到问题的原因, 从源去解决才是最根本的办法. 后面我们讲到 **Flowable** 的时候, 大家就会发现它其实没什么神秘的, 它用到办法和我们本节所讲的基本上是一样的, 只是它稍微做了点封装.