链滴

RxJava2 中的 Backpressure

作者：flowaters

原文链接：https://ld246.com/article/1530090236346

来源网站：链滴

背景

反压(背压, Backpressure)是什么呢?

被观察者生产事件的速度太快，超过了观察者处理事件的速度时，这里系统就会变的不稳定。

如果不做处理，事件会堆积到内存中，最终导致OOM。

反压提供了一种方式，来制定这种情况发生时，可以采用的策略。

原理

响应式拉取(reactive pull)

在Observable中，是被观察者生产数据后，主动将数据推向观察者，观察者来消费。

在Flowable中，是观察者将数据消费掉后，主动从被观察者那边来拉取数据，被观察者等待通知再发数据。

具体在代码中，是通过Subscription.request(number)来拉取指定个数的数据的。

如果不想用反压策略，可以设置Subscription.request(LONG.MAX_VALUE)。

异步线程

观察者线程和被观察者线程是不同的线程。在同一样线程中，被观察者需要等待观察者将事件处理完后才会继续发送下面的事件。

示例

Observable无反压

```
import java.util.Date;

import io.reactivex.Observable;
import io.reactivex.ObservableEmitter;
import io.reactivex.ObservableOnSubscribe;
import io.reactivex.Observer;
import io.reactivex.disposables.Disposable;
import io.reactivex.schedulers.Schedulers;

public class HelloObservable {
    public static void main(String[] args) throws InterruptedException {
        Observable.create(new ObservableOnSubscribe<Long>() {

            @Override
            public void subscribe(ObservableEmitter<Long> e) throws Exception {
                for (long i = 0;; ++i) {
                    e.onNext(i);
                }
```

```
            }
        }).observeOn(Schedulers.newThread()).subscribe(new Observer<Long>() {

            @Override
            public void onSubscribe(Disposable d) {

            }

            @Override
            public void onNext(Long t) {
                System.out.println((new Date()) + ": " + t);
                try {
                    Thread.sleep(1000L);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            @Override
            public void onError(Throwable e) {
                e.printStackTrace();
            }

            @Override
            public void onComplete() {
                System.out.println("Complete");
            }
        });

        Thread.sleep(10000L);
    }
}
```

结果

```
Wed Jun 27 10:08:41 CST 2018: 0
Wed Jun 27 10:08:42 CST 2018: 1
Wed Jun 27 10:08:44 CST 2018: 2
Wed Jun 27 10:08:48 CST 2018: 3
Wed Jun 27 10:08:49 CST 2018: 4
Wed Jun 27 10:08:55 CST 2018: 5
Wed Jun 27 10:08:56 CST 2018: 6
Wed Jun 27 10:08:59 CST 2018: 7
```

如果正常的话，应该每1秒输出一个值。但是由于GC的影响，输出时间在逐渐增加。

Flowable有反压

Flowable支持反压，有五种策略

策略	原文描述	备注
BUFFER	Buffers all onNext values until the downstream cons mes it.	缓存所有的值

原文链接：RxJava2 中的 Backpressure

DROP Drops the most recent onNext value if the downstream
an't keep up 丢弃最新的值

ERROR Signals a MissingBackpressureException in case the d
wnstream can't keep up 触发异常

MISSING OnNext events are written without any buffering or
ropping. Downstream has to deal with any overflow. Useful when one applies one of the cust
m-parameter onBackpressureXXX operators. 不丢弃, 不缓存,
发异常

LATEST Keeps only the latest onNext value, overwriting any p
evious value if the downstream can't keep up.

Drop 和 Latest 都不会Buffer数据，详见参考中的[4]。

示例: Flowable Buffer策略

使用的通用的测试模板，可以测试多种策略，需要注意的点是：

1. 策略对应于模板中的 BackpressureStrategy.BUFFER，调整策略即调整这块儿的参数。

2. 通过 observeOn(Schedulers.newThread()) 来保证异步。如果没有这段的话，会在同一个线程运行。

下面的示例，先快速生产3秒的数据，然后停止生产数据；消费时头3秒先缓慢消费，然后快速消费完余的数据。

```
import java.util.Date;

import org.reactivestreams.Subscriber;
import org.reactivestreams.Subscription;

import io.reactivex.BackpressureStrategy;
import io.reactivex.Flowable;
import io.reactivex.FlowableEmitter;
import io.reactivex.FlowableOnSubscribe;
import io.reactivex.schedulers.Schedulers;

public class HelloFlowableBuffer {
    public static void main(String[] args) throws InterruptedException {

        Flowable.create(new FlowableOnSubscribe<Long>() {

            @Override
            public void subscribe(FlowableEmitter<Long> e) throws Exception {
                // 仅持续产生3秒数据
                long begin = System.currentTimeMillis();
                for (long i = 0;; ++i) {
                    e.onNext(i);
                    // 到3秒，则退出.
                    if (System.currentTimeMillis() - begin > 3000L) {
                        break;
                    }
                }
            }
```

```
        }, BackpressureStrategy.BUFFER).observeOn(Schedulers.newThread()).subscribe(new Sub
criber<Long>() {

            private Subscription s;

            private long begin = System.currentTimeMillis();

            @Override
            public void onComplete() {
                System.out.println("Complete");
            }

            @Override
            public void onError(Throwable t) {
                t.printStackTrace();
            }

            @Override
            public void onNext(Long l) {
                System.out.println((new Date()) + ": " + l);

                // 前3秒时有停留, 之后不停留.
                if (System.currentTimeMillis() - begin < 3000) {
                    try {
                        Thread.sleep(1000L);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                this.s.request(1);
            }

            @Override
            public void onSubscribe(Subscription s) {
                this.s = s;
                s.request(1);
            }
        });

        Thread.sleep(5000L);
    }
}
```

Buffer策略就是都保存在内存中，其实和上面的情况是一样的。

结果

```
Wed Jun 27 10:38:40 CST 2018: 0
Wed Jun 27 10:38:41 CST 2018: 1
Wed Jun 27 10:38:44 CST 2018: 2
.............
Wed Jun 27 10:38:49 CST 2018: 567521
Wed Jun 27 10:38:49 CST 2018: 567522
Wed Jun 27 10:38:49 CST 2018: 567523
```

示例: Flowable Drop策略

使用策略: BackpressureStrategy.DROP

结果

```
Wed Jun 27 10:50:04 CST 2018: 0
Wed Jun 27 10:50:06 CST 2018: 1
Wed Jun 27 10:50:07 CST 2018: 2
.............
Wed Jun 27 10:50:08 CST 2018: 125
Wed Jun 27 10:50:08 CST 2018: 126
Wed Jun 27 10:50:08 CST 2018: 127
```

示例: Flowable Latest策略

使用策略: BackpressureStrategy.LATEST

结果

```
Wed Jun 27 10:58:08 CST 2018: 0
Wed Jun 27 10:58:09 CST 2018: 1
Wed Jun 27 10:58:10 CST 2018: 2
.............
Wed Jun 27 10:58:11 CST 2018: 126
Wed Jun 27 10:58:11 CST 2018: 127
Wed Jun 27 10:58:11 CST 2018: 50155889
```

可见LATEST策略的处理方式为只保留最新的一条数据。

示例: Flowable Error策略

使用策略: BackpressureStrategy.ERROR

结果

```
Wed Jun 27 11:01:49 CST 2018: 0
io.reactivex.exceptions.MissingBackpressureException: create: could not emit value due to lack of requests
    at io.reactivex.internal.operators.flowable.FlowableCreate$ErrorAsyncEmitter.onOverflow(FlowableCreate.java:442)
    at io.reactivex.internal.operators.flowable.FlowableCreate$NoOverflowBaseAsyncEmitter.onNext(FlowableCreate.java:408)
    at backpressure.HelloFlowableError$1.subscribe(HelloFlowableError.java:24)
    at io.reactivex.internal.operators.flowable.FlowableCreate.subscribeActual(FlowableCreate.java:72)
    at io.reactivex.Flowable.subscribe(Flowable.java:14349)
    at io.reactivex.internal.operators.flowable.FlowableObserveOn.subscribeActual(FlowableObserveOn.java:56)
    at io.reactivex.Flowable.subscribe(Flowable.java:14349)
    at io.reactivex.Flowable.subscribe(Flowable.java:14298)
    at backpressure.HelloFlowableError.main(HelloFlowableError.java:31)
```

示例: Flowable Missing策略

使用策略: BackpressureStrategy.MISSING

结果

```
Wed Jun 27 11:02:53 CST 2018: 0
io.reactivex.exceptions.MissingBackpressureException: Queue is full?!
    at io.reactivex.internal.operators.flowable.FlowableObserveOn$BaseObserveOnSubscriber.
nNext(FlowableObserveOn.java:114)
    at io.reactivex.internal.operators.flowable.FlowableCreate$MissingEmitter.onNext(Flowable
reate.java:369)
    at backpressure.HelloFlowableMissing$1.subscribe(HelloFlowableMissing.java:24)
    at io.reactivex.internal.operators.flowable.FlowableCreate.subscribeActual(FlowableCreate.j
va:72)
    at io.reactivex.Flowable.subscribe(Flowable.java:14349)
    at io.reactivex.internal.operators.flowable.FlowableObserveOn.subscribeActual(FlowableOb
erveOn.java:56)
    at io.reactivex.Flowable.subscribe(Flowable.java:14349)
    at io.reactivex.Flowable.subscribe(Flowable.java:14298)
    at backpressure.HelloFlowableMissing.main(HelloFlowableMissing.java:31)
```

附录

调度器类型

调度器类型有

调度器类型	效果
computation	计算任务
io	IO任务
newThread	新启动一个线程
from(executor)	指定线程池
single	单线程串行执行
trampoline	放入当前线程队列

自由切换调度器示例

```
import io.reactivex.Flowable;
import io.reactivex.functions.Consumer;
import io.reactivex.functions.Function;
import io.reactivex.schedulers.Schedulers;

public class HelloThread {
    public static void main(String[] args) throws InterruptedException {

        Flowable.range(1, 3).observeOn(Schedulers.computation()) // 下面的computation线程
                .map(new Function<Integer, String>() {
```

```
                @Override
                public String apply(Integer t) throws Exception {
                    System.out.println("map 1: " + Thread.currentThread());
                    return String.valueOf(t);
                }
            }).observeOn(Schedulers.io()) // 下面的是io线程
            .map(new Function<String, Integer>() {

                @Override
                public Integer apply(String t) throws Exception {
                    System.out.println("map 2: " + Thread.currentThread());
                    return Integer.valueOf(t);
                }
            }).subscribeOn(Schedulers.single()) // 下面的在严格的单线程串行
            .subscribe(new Consumer<Integer>() {
                @Override
                public void accept(Integer t) throws Exception {
                    System.out.println("final: " + Thread.currentThread() + ", t");
                }
            });

        Thread.sleep(3000L);
    }
}
```

结果

```
map 1: Thread[RxComputationThreadPool-1,5,main]
map 1: Thread[RxComputationThreadPool-1,5,main]
map 1: Thread[RxComputationThreadPool-1,5,main]
map 2: Thread[RxCachedThreadScheduler-1,5,main]
final: Thread[RxCachedThreadScheduler-1,5,main], t
map 2: Thread[RxCachedThreadScheduler-1,5,main]
final: Thread[RxCachedThreadScheduler-1,5,main], t
map 2: Thread[RxCachedThreadScheduler-1,5,main]
final: Thread[RxCachedThreadScheduler-1,5,main], t
```

减少数据量的方法

采样: Sample算子

使用Sample算子，可以采样数据

```
import java.util.Date;
import java.util.concurrent.TimeUnit;

import org.reactivestreams.Subscriber;
import org.reactivestreams.Subscription;

import io.reactivex.BackpressureStrategy;
import io.reactivex.Flowable;
import io.reactivex.FlowableEmitter;
import io.reactivex.FlowableOnSubscribe;
```

```
import io.reactivex.schedulers.Schedulers;

public class HelloFlowableBufferSample {
    public static void main(String[] args) throws InterruptedException {

        Flowable.create(new FlowableOnSubscribe<Long>() {

            @Override
            public void subscribe(FlowableEmitter<Long> e) throws Exception {
                // 仅持续产生3秒数据
                long begin = System.currentTimeMillis();
                for (long i = 0;; ++i) {
                    e.onNext(i);
                    // 到3秒，则退出.
                    if (System.currentTimeMillis() - begin > 3000L) {
                        break;
                    }
                }
            }
        }, BackpressureStrategy.BUFFER).sample(1, TimeUnit.SECONDS) // 采样: 每秒一条数据
                .observeOn(Schedulers.newThread()).subscribe(new Subscriber<Long>() {

                    private Subscription s;

                    private long begin = System.currentTimeMillis();

                    @Override
                    public void onComplete() {
                        System.out.println("Complete");
                    }

                    @Override
                    public void onError(Throwable t) {
                        t.printStackTrace();
                    }

                    @Override
                    public void onNext(Long l) {
                        System.out.println((new Date()) + ": " + l);

                        // 前3秒时有停留, 之后不停留.
                        if (System.currentTimeMillis() - begin < 3000) {
                            try {
                                Thread.sleep(1000L);
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        }
                        this.s.request(1);
                    }

                    @Override
                    public void onSubscribe(Subscription s) {
                        this.s = s;
```

```
                    s.request(1);
                }
            });

        Thread.sleep(5000L);
    }
}
```

结果

```
Wed Jun 27 15:13:10 CST 2018: 14425216
Wed Jun 27 15:13:11 CST 2018: 29448431
Wed Jun 27 15:13:12 CST 2018: 44742890
```

可见每秒采样一条数据。

过滤：过滤事件

```
import java.util.Date;

import org.reactivestreams.Subscriber;
import org.reactivestreams.Subscription;

import io.reactivex.BackpressureStrategy;
import io.reactivex.Flowable;
import io.reactivex.FlowableEmitter;
import io.reactivex.FlowableOnSubscribe;
import io.reactivex.schedulers.Schedulers;

public class HelloFlowableBufferFilter {
    public static void main(String[] args) throws InterruptedException {

        Flowable.create(new FlowableOnSubscribe<Long>() {

            @Override
            public void subscribe(FlowableEmitter<Long> e) throws Exception {
                // 仅持续产生3秒数据
                long begin = System.currentTimeMillis();
                for (long i = 0;; ++i) {
                    e.onNext(i);
                    // 到3秒，则退出.
                    if (System.currentTimeMillis() - begin > 3000L) {
                        break;
                    }
                }
            }
        }, BackpressureStrategy.BUFFER).filter(t -> t % 5000000 == 0) // 过滤: 每500万条取1条
                .observeOn(Schedulers.newThread()).subscribe(new Subscriber<Long>() {

                    private Subscription s;

                    private long begin = System.currentTimeMillis();

                    @Override
```

```
                public void onComplete() {
                    System.out.println("Complete");
                }

                @Override
                public void onError(Throwable t) {
                    t.printStackTrace();
                }

                @Override
                public void onNext(Long l) {
                    System.out.println((new Date()) + ": " + l);

                    // 前3秒时有停留, 之后不停留.
                    if (System.currentTimeMillis() - begin < 3000) {
                        try {
                            Thread.sleep(1000L);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                    this.s.request(1);
                }

                @Override
                public void onSubscribe(Subscription s) {
                    this.s = s;
                    s.request(1);
                }
            });

        Thread.sleep(5000L);
    }
}
```

结果

```
Wed Jun 27 15:18:28 CST 2018: 0
Wed Jun 27 15:18:29 CST 2018: 5000000
Wed Jun 27 15:18:30 CST 2018: 10000000
Wed Jun 27 15:18:31 CST 2018: 15000000
Wed Jun 27 15:18:31 CST 2018: 20000000
Wed Jun 27 15:18:31 CST 2018: 25000000
```

ParallelFlowable

ParallelFlowable提供了一种并行执行的方式.

```
import io.reactivex.Flowable;
import io.reactivex.parallel.ParallelFlowable;
import io.reactivex.schedulers.Schedulers;

public class HelloParallelFlowableBufferSample {
    public static void main(String[] args) throws InterruptedException {
```

```
        ParallelFlowable.from(Flowable.range(1, 10), 4) // 并行度为4
                .runOn(Schedulers.computation()) // 运行在计算线程池上
                .map(i -> i * 2) // 简单map
                .sequential() // 将结果merge
                .subscribe(System.out::println); // 输出到控制台

        Thread.sleep(3000L);
    }
}
```

结果

```
2
4
6
8
12
14
16
20
10
18
```

参考

1. 关于RxJava最友好的文章——背压（Backpressure）
2. 关于RxJava最友好的文章（初级篇）
3. 关于RxJava背压
4. Reactive Streams and RxJava2
5. RxJava中backpressure这个概念的理解