無料電子ブック

学習

rx-java

Free unaffiliated eBook created from
Stack Overflow contributors.

#rx-java

1
1: rx-java 2
2
2
Examples 2
2
3
RxJava 4
5
2: Android with RxJava 7
7
Examples 7
RxAndroid - AndroidSchedulers 7
RxLifecycle 7
Rxpermissions 9
3: RxJava2 FlowableSubscriber 10
10
10
Examples 10
10
4: 13
Examples 13
13
5: 15
15
Examples 15
TestSubscriber 15
15
16
16
Observable#error 16

TestScheduler 17

6: RxJava 19

Examples 19

RxJava 19

19

19

7: 21

21

Examples 21

21

flatMap 22

23

23

doOnNext 24

24

8: 27

27

27

27

Examples 27

27

PublishSubject 28

9: 33

Examples 33

33

onBackpressureXXX 35

35

/ 36

onBackpressureBuffer 37

onBackpressureBufferint capacity 37

onBackpressureBufferint capacityAction0 onOverflow 37

onBackpressureBufferint capacityAction0 onOverflowBackpressureOverflow.Strategy 37

onBackpressureDrop 38
onBackpressureLatest 38
..... 39
..... 39
fromCallable 40
..... 40
SyncOnSubscribe 41
..... 42
10: 45
Examples 45
Observable 45
..... 45
..... 45
..... 45
..... 45
..... 46
..... 46
..... 48

1: rx-javaをいめる

このセクションでは、rx-javaのなとについてします。

RxJavaは、 Reactive Extensionsの Java VMです。これは、なシーケンスをしておよびイベントベースのプログラムをするためのライブラリです。

RxJavaについてのはWiki Homeをごください。

バージョン

バージョン		の	
1.x		1.3.0	2017-05-05
2.x		2.1.1	2017-06-21

Examples

インストールまたはセットアップ

rx-javaのセットアップ

1. け

```
compile 'io.reactivex:rxjava2:rxjava:2.1.1'
```

2. Maven

```
<dependency>
    <groupId>io.reactivex.rxjava2</groupId>
    <artifactId>rxjava</artifactId>
    <version>2.1.1</version>
</dependency>
```

3. アイビー

```
<dependency org="io.reactivex.rxjava2" name="rxjava" rev="2.1.1" />
```

4. JFrogのスナップショット

```
repositories {
maven { url 'https://oss.jfrog.org/libs-snapshot' }
}
```

```
dependencies {
    compile 'io.reactivex:rxjava:2.0.0-SNAPSHOT'
}
```

5. ビルドシステムをせずにjarファイルをダウンロードするがあるは、するバージョンの Maven `pom`ファイルをのようにします。

```
<?xml version="1.0"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.netflix.rxjava.download</groupId>
    <artifactId>rxjava-download</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>Simple POM to download rxjava and dependencies</name>
    <url>http://github.com/ReactiveX/RxJava</url>
    <dependencies>
        <dependency>
            <groupId>io.reactivex</groupId>
            <artifactId>rxjava</artifactId>
            <version>2.0.0</version>
            <scope/>
        </dependency>
    </dependencies>
</project>
```

それから、

```
$ mvn -f download-rxjava-pom.xml dependency:copy-dependencies
```

このコマンドは、 `rxjava-*.jar`とそのを`./target/dependency/`.

Java 6がです。

こんにちは

`Hello, World!`というメッセージがされます`Hello, World!`コンソールに

```
public void hello() {
  Observable.just("Hello, World!") // create new observable
    .subscribe(new Action1<String>() { // subscribe and perform action

       @Override
       public void call(String st) {
         System.out.println(st);
       }

    });
}
```

またはJava 8ラムダをする

```
public void hello() {
      Observable.just("Hello, World!") // create new observable
        .subscribe(onNext -> { // subscribe and perform action
             System.out.println(onNext);
        });
}
```

RxJavaの

RxJavaのコアコンセプトは`Observables`と`Subscribers`です。 `Observable`は、 `Subscriber`それらをしているにオブジェクトをします。

な

`Observable`は、なデザインパターンをするクラスです。これらのObservableは、がイベントのをサブスクライブできるようにするメソッドをします。イベントのは、オブザーバブルによってトリガされます。 `Observable`がつことができるサブスクライバの、または`Observable`が`Observable`できるオブジェクトのにはありません。

えば

```
Observable<Integer> integerObservable = Observable.just(1, 2, 3); // Integer observable
Observable<String> stringObservable = Observable.just("Hello, ", "World", "!"); // String
observable
```

ここでは、とばれるなオブジェクト`integerObservable`と`stringObservable` 、ファクトリメソッドからされた`just`のRxライブラリでします。 `Observable`はであるため、のオブジェクトをすることができます。

`Subscriber`はです。 `Subscriber`は、 ***1***つのオブザーバブルのみにサブスクライブできます。 `Observable`びし`onNext() onCompleted()`と`onError()`のメソッド`Subscriber`。

```
Subscriber<Integer> mSubscriber = new Subscriber<Integer>() {
        // NOTE THAT ALL THESE ARE CALLED BY THE OBSERVABLE
        @Override
        public void onCompleted() {
            // called when all objects are emitted
            System.out.println("onCompleted called!");
        }

        @Override
        public void onError(Throwable throwable) {
            // called when an error occurs during emitting objects
            System.out.println("onError called!");
        }

        @Override
        public void onNext(Integer integer) {
            // called for each object that is emitted
            System.out.println("onNext called with: " + integer);
        }
    };
```

`Subscriber`もであり、のオブジェクトをサポートできることにしてください。 `Subscriber`は、オブザーバブルの`subscribe`メソッドをびすことによって、オブザーバブルにサブスクライブするがあります。

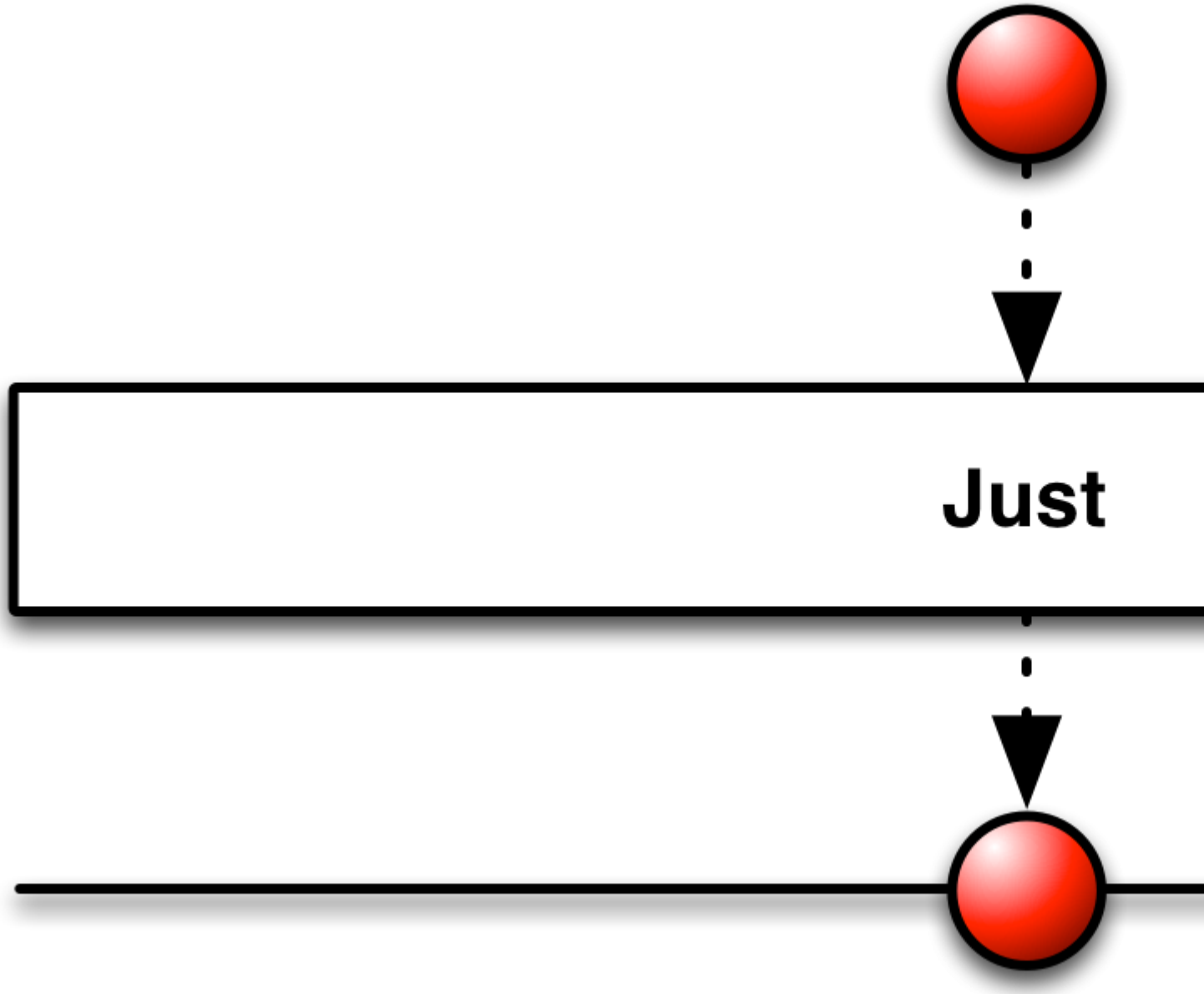
```
integerObservable.subscribe(mSubscriber);
```

をすると、のがされます。

```
onNext called with: 1
onNext called with: 2
onNext called with: 3
onCompleted called!
```

の

Observableはなるイベントのれとえることができます。 Observableをすると、onNext、onComplete、onErrorという3つのリスナーがあります。 Observableがしいをするたびにが`onNext`びされます。Observableがそれののをしたことをすると、onCompleteがびされます。Observableチェーンのにがスローされた、onErrorがびされます。 Rxにをするには、をして、のでがこるかをします。はなObservable "Just"のです。



マーブルには、のをすブロック、したイベントをすバー、エラーをすX、そののがをします。そのことをにいて、「ちょうど」がをち、onNextをい、にonCompleteでわることがわかります。ちょうど「ちょうど」というがたくさんあります。 ReactXプロジェクトのであるすべてのと、RexXサイトの RxJavaでのをることができます。 RxMarblesサイトをじたインタラクティブなもあります 。

オンラインでrx-javaをいめるをむ https://riptutorial.com/ja/rx-java/topic/974/rx-javaをいめる

2: Android with RxJava

RxAndroidはくのをえたライブラリでした。これは、バージョン0.25.0から1.xにするくのなるライブラリでされています。

1.0よりになをするライブラリのリストは、 ここでされています 。

Examples

RxAndroid - AndroidSchedulers

これはり、AndroidでRxJavaをいめるためになのものです。

あなたのgradleのにRxJavaとRxAndroidをめる

```
 // use the last version
compile 'io.reactivex.rxjava2:rxjava:2.1.1'
compile 'io.reactivex.rxjava2:rxandroid:2.0.1'
```

RxJavaへのRxAndroidなは、Androidのメインスレッドまたはuiスレッドのスケジューラです。

あなたのコードで

```
Observable.just("one", "two", "three", "four", "five")
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(
            data -> doStuffOnMainThread(),
            error -> handleErrorOnMainThread()
        )
```

または、カスタムLooperスケジューラをすることもできます。

```
Looper backgroundLooper = // ...
Observable.just("one", "two", "three", "four", "five")
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.from(backgroundLooper))
        .subscribe(
            data -> doStuffOnMainThread(),
            error -> handleErrorOnMainThread()
        )
```

ほとんどのは、のRxJavaのドキュメントをしてください。

RxLifecycleコンポーネント

RxLifecycleライブラリをすると、なサブスクリプションをAndroidアクティビティやフラグメントライフサイクルににバインドできます。

Observableのをれると、メモリリークがし、システムによってされたでアクティビティ/フラグメントアライブイベントがされるがあります。

にライブラリをする

```
// use the last version available
compile 'com.trello:rxlifecycle:0.6.1'
compile 'com.trello:rxlifecycle-components:0.6.1'
```

に、 `Rx*`クラスをします。

- `RxActivity` / `support.RxFragmentActivity` / `support.RxAppCompatActivity`
- `RxFragment` / `support.RxFragment`
- `RxDialogFragment` / `support.RxDialogFragment`
- `support.RxAppCompatDialogActivity`

あなたはすべてされていますが、Observableをするとのことがになります

```
someObservable
    .compose(bindToLifecycle())
    .subscribe();
```

アクティビティの`onCreate()`メソッドでこれをすると、 `onDestroy()`でにされます。

じことがこる

- `onStart()` -> `onStop()`
- `onResume()` -> `onPause()`
- `onAttach()` -> `onDetach()` フラグメントのみ
- `onViewCreated()` -> `onDestroyView()` フラグメントのみ

わりに、サブスクリプションをするときにイベントをすることもできます。

アクティビティから

```
someObservable
    .compose(bindUntilEvent(ActivityEvent.DESTROY))
    .subscribe();
```

から

```
someObservable
    .compose(bindUntilEvent(FragmentEvent.DESTROY_VIEW))
    .subscribe();
```

ライフサイクルイベントをするには、メソッド`lifecycle()`をしてなライフサイクルをすることもできます。

RxLifecycleはライフサイクルオブザーバブルにすこともできます

```
.compose(RxLifecycleAndroid.bindActivity(lifecycle))
```

`Single`または`Completable`をするがあるは、bindメソッドのに`forSingle()`または`forCompletable`をそれぞれするだけで`Completable`ます。

```
someSingle
    .compose(bindToLifecycle().forSingle())
    .subscribe();
```

Naviライブラリでもできます。

Rxpermissions

このライブラリは、しいAndroid Mモデルで RxJavaのをします。

にライブラリをする

Rxjava

```
dependencies {
    compile 'com.tbruyelle.rxpermissions:rxpermissions:0.8.0@aar'
}
```

Rxjava2

```
dependencies {
    compile 'com.tbruyelle.rxpermissions2:rxpermissions:0.8.1@aar'
}
```

にするためにRetrolambdaをしますが、ではありません

```
// Must be done during an initialization phase like onCreate
RxPermissions.getInstance(this)
    .request(Manifest.permission.CAMERA)
    .subscribe(granted -> {
        if (granted) { // Always true pre-M
           // I can control the camera now
        } else {
           // Oups permission denied
        }
    });
```

もっとむ https : //github.com/tbruyelle/RxPermissions 。

オンラインでAndroid with RxJavaをむ https://riptutorial.com/ja/rx-java/topic/7125/android-with-rxjava

3: RxJava2 FlowableおよびSubscriber

き

このトピックでは、rxjava version2でされたフロ―ティングとサブスクライバのにするとドキュメントをします。

このではとしてrxjava2がです。されるバ―ジョンのmavenはのとおりです。

```
    <dependency>
        <groupId>io.reactivex.rxjava2</groupId>
        <artifactId>rxjava</artifactId>
        <version>2.0.8</version>
    </dependency>
```

Examples

プロデュ―サ―のバックプレッシャ―をサポ―トするプロデュ―サ―のの

このの`TestProducer`は、されたの`Integer`オブジェクトをし、 `Subscriber`プッシュします。 `Flowable<Integer>`クラスをします。しいサブスクライバの、 `request(long)`メソッドをしてIntegerをおよびする`Subscription`オブジェクトをします。

`subscriber`される`Subscription`では、 `subscriber onNext()`メソッドをびす`request()`メソッドをこの`onNext()`コ―ルからにびすことが`onNext()`です。スタックのオ―バ―フロ―をぐために、では`outStandingRequests`カウンタと`isProducing`フラグをします。

```
class TestProducer extends Flowable<Integer> {
    static final Logger logger = LoggerFactory.getLogger(TestProducer.class);
    final int from, to;

    public TestProducer(int from, int to) {
        this.from = from;
        this.to = to;
    }

    @Override
    protected void subscribeActual(Subscriber<? super Integer> subscriber) {
        subscriber.onSubscribe(new Subscription() {

            /** the next value. */
            public int next = from;
            /** cancellation flag. */
            private volatile boolean cancelled = false;
            private volatile boolean isProducing = false;
            private AtomicLong outStandingRequests = new AtomicLong(0);

            @Override
            public void request(long n) {
```

```
                if (!cancelled) {

                    outStandingRequests.addAndGet(n);

                    // check if already fulfilling request to prevent call  between request()
an subscriber .onNext()
                    if (isProducing) {
                        return;
                    }

                    // start producing
                    isProducing = true;

                    while (outStandingRequests.get() > 0) {
                        if (next > to) {
                            logger.info("producer finished");
                            subscriber.onComplete();
                            break;
                        }
                        subscriber.onNext(next++);
                        outStandingRequests.decrementAndGet();
                    }
                    isProducing = false;
                }
            }

            @Override
            public void cancel() {
                cancelled = true;
            }
        });
    }
}
```

このConsumerは`DefaultSubscriber<Integer>`を`DefaultSubscriber<Integer>`、とにをするとのをします。 Integerをするとしれるので、はプロデューサのためにされます。

```
class TestConsumer extends DefaultSubscriber<Integer> {

    private static final Logger logger = LoggerFactory.getLogger(TestConsumer.class);

    @Override
    protected void onStart() {
        request(1);
    }

    @Override
    public void onNext(Integer i) {
        logger.info("consuming {}", i);
        if (0 == (i % 5)) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException ignored) {
                // can be ignored, just used for pausing
            }
        }
        request(1);
    }
```

```
    @Override
    public void onError(Throwable throwable) {
        logger.error("error received", throwable);
    }

    @Override
    public void onComplete() {
        logger.info("consumer finished");
    }
}
```

テストクラスののメインメソッドでは、プロデューサとコンシューマがされ、されます。

```
public static void main(String[] args) {
    try {
        final TestProducer testProducer = new TestProducer(1, 1_000);
        final TestConsumer testConsumer = new TestConsumer();

        testProducer
                .subscribeOn(Schedulers.computation())
                .observeOn(Schedulers.single())
                .blockingSubscribe(testConsumer);

    } catch (Throwable t) {
        t.printStackTrace();
    }
}
```

このをすると、ログファイルにはコンシューマがにされ、rxjava2のFlowableバッファをリフィルするがあるにのみプロデューサがアクティブになります。

オンラインでRxJava2 FlowableおよびSubscriberをむ https://riptutorial.com/ja/rx-java/topic/9810/rxjava2-flowableおよびsubscriber

4: スケジューラ

Examples

な

スケジューラは、にするRxJavaです。スケジューラーはExecutorサービスによってバックアップできますが、のスケジューラーをできます。

`Scheduler`はこのをたすがあります。

- のないタスクをシーケンシャルにするがありますFIFO
- タスクがれることがあります

`Scheduler`は、の `delay` のパラメータとしてすることも、 `subscribeOn` / `observeOn`メソッドとともにすることもできます。

のオペレータでは、 `Scheduler`がのオペレータのタスクをするためにされます。たとえば、 `delay` はのをするタスクをスケジュールします。これはでしてする`Scheduler`です。

`subscribeOn`は`Observable`ごとに1できます。これは、サブスクリプションのコードがになる`Scheduler`をします。

`observeOn`は`Observable`ごとにできます。これは、 `observeOn`メソッドのにされたすべてのタスクのにどの`Scheduler`がされるかをします。 `observeOn`はスレッドホッピングをするのにちます。

subscribeOnスケジューラ

```
// this lambda will be executed in the `Schedulers.io()`
Observable.fromCallable(() -> Thread.currentThread().getName())
          .subscribeOn(Schedulers.io())
          .subscribe(System.out::println);
```

のスケジューラで**observeOn**

```
Observable.fromCallable(() -> "Thread -> " + Thread.currentThread().getName())
         // next tasks will be executed in the io scheduler
         .observeOn(Schedulers.io())
         .map(str -> str + " -> " + Thread.currentThread().getName())
          // next tasks will be executed in the computation scheduler
         .observeOn(Schedulers.computation())
         .map(str -> str + " -> " + Thread.currentThread().getName())
         // next tasks will be executed in the io scheduler
         .observeOn(Schedulers.newThread())
         .subscribe(str -> System.out.println(str + " -> " +
Thread.currentThread().getName()));
```

をしてのスケジューラをする

のは、 `Scheduler`をパラメータとしてできます。

```
Observable.just(1)
          // the onNext method of the delay operator will be executed in a new thread
          .delay(1, TimeUnit.SECONDS, Schedulers.newThread())
          .subscribe(System.out::println);
```

にする

```
TestScheduler testScheduler = Schedulers.test();
EventBus sut = new DefaultEventBus(testScheduler);
TestSubscriber<Event> subscriber = new TestSubscriber<Event>();
sut.get().subscribe(subscriber);
sut.publish(event);
testScheduler.advanceTimeBy(1, TimeUnit.SECONDS);
```

スレッドプール

```
this.poolName = schedulerFig.getIoSchedulerName();
final int poolSize = schedulerFig.getMaxIoThreads();
final BlockingQueue<Runnable> queue = new ArrayBlockingQueue<Runnable>(poolSize);
final MaxSizeThreadPool threadPool = new MaxSizeThreadPool( queue, poolSize );
this.scheduler = Schedulers.from(threadPool);
```

な**Web**ソケット

```
final Subscription subscribe = socket.webSocketObservable()
        .subscribeOn(Schedulers.io())
        .doOnNext(new Action1<RxEvent>() {
            @Override
            public void call(RxEvent rxEvent) {
                System.out.println("Event: " + rxEvent);
            }
        })
        .subscribe();
```

オンラインでスケジューラをむ https://riptutorial.com/ja/rx-java/topic/2321/スケジューラ

5: ユニットテスト

Schedulerメソッドはすべてなので、RxJavaフックをしたテストをじJVMインスタンスでにすることはできません。それらがどこにあれば、ユニットテストので1つのTestSchedulerがされます。にはSchedulersクラスをすることのです。

Examples

TestSubscriber

TestSubscribersをすると、のサブスクライバをしたり、アクションをサブスクライブするをけることができます。のがされているかどうか、Observableがしたかどうか、がしたかどうか、

このでは、1,2,3および4がonNextでObservableにされたというアサーションをしています。

```
TestSubscriber<Integer> ts = TestSubscriber.create();
Observable.just(1,2,3,4).subscribe(ts);
ts.assertValues(1,2,3,4); // Success
```

`assertValues`は、カウントがしいことをアサートします。いくつかのをすだけなら、アサーションはします。

```
TestSubscriber<Integer> ts = TestSubscriber.create();
Observable.just(1,2,3,4).subscribe(ts);
ts.assertValues(1,2,3); // Fail
```

`assertValues`は、アサートに`equals`メソッドをします。これにより、データとしてわれるクラスをにテストできます。

```
TestSubscriber<Object> ts = TestSubscriber.create();
Observable.just(new Object(), new Object()).subscribe(ts);
ts.assertValues(new Object(), new Object()); // Fail
```

このでは、equalsがされ、Observableのをアサートするクラスをします。

```
public class Room {

    public String floor;
    public String number;

    @Override
    public boolean equals(Object o) {
        if (o == this) {
            return true;
        }
        if (o instanceof Room) {
            Room that = (Room) o;
```

```
            return (this.floor.equals(that.floor))
                    && (this.number.equals(that.number));
        }
        return false;
    }
}

TestSubscriber<Room> ts = TestSubscriber.create();
Observable.just(new Room("1", "10")).subscribe(ts);
ts.assertValue(new Room("1", "10"); // Success
```

また、1つのをチェックするだけでよいので、よりい`assertValue`をすることにしてください。

すべてのイベントをする

があれば、すべてのイベントをリストとしてねることもできます。

```
TestSubscriber<Integer> ts = TestSubscriber.create();
Observable.just(1,2,3,4).subscribe(ts);
List<Integer> onNextEvents = ts.getOnNextEvents();
List<Throwable> onErrorEvents = ts.getOnErrorEvents();
List<Notification<Integer>> onCompletedEvents = ts.getOnCompletedEvents();
```

イベントのアサート

あなたのイベントについてよりなテストをしたいは、あなたのきなアサーションライブラリと`getOnNextEvents`または`getOn*Events`をみわせることができます

```
Observable<Integer> obs = Observable.just(1,2,3,4)
    .filter( x -> x % 2 == 0);

// note that we instanciate TestSubscriber via the constructor here
TestSubscriber<Integer> ts = new TestSubscriber();
obs.subscribe(ts);

// Note that we are not using Observable#forEach here
// but java.lang.Iterable#forEach.
// You should never use Observable#forEach unless you know
// exactly what you're doing
ts.getOnNextEvents()
    .forEach( integer -> assertTrue(integer % 2 == 0));
```

`Observable#error`テストする

しいクラスがされていることをすることができます

```
Observable<Integer> obs = Observable.error(new Exception("I am a Teapot"));

TestSubscriber<Integer> ts = new TestSubscriber<>();
```

```
obs.subscribe(ts);

ts.assertError(Exception.class);
```

また、ながスロ―されたことをすることもできます。

```
Exception e = new Exception("I am a Teapot");
Observable<Integer> obs = Observable.error(e);

TestSubscriber<Integer> ts = new TestSubscriber<>();
obs.subscribe(ts);

ts.assertError(e);
```

TestScheduler

TestSchedulersをすると、ビジ―なちやスレッドなど、システムをするがなく、オブザ―バブルのとをできます。これは、でのあるなテストをしたいにはにです。あなたがをしているので、スレッドがしたり、テストがいマシンでしたり、をっているをしたりするはなくなりました。

TestSchedulersは、RxJavaのためにスケジュ―ラをとするオ―バ―ロ―ドをしてすることができます。

```
TestScheduler testScheduler = new TestScheduler();
TestSubscriber<Integer> subscriber = TestSubscriber.create();
Observable.just(1,2,3)
          .delay(10, TimeUnit.SECONDS, testScheduler)
          .subscribe(subscriber);

try {
    Thread.sleep(TimeUnit.SECONDS.toMillis(11));
} catch (InterruptedException ignored) { }
subscriber.assertValues(1,2,3); // fails

testScheduler.advanceTimeBy(10, TimeUnit.SECONDS);
subscriber.assertValues(1,2,3); // success
```

TestSchedulerはかなりです。これは3つのでされています。

```
testScheduler.advanceTimeBy(amount, timeUnit);
testScheduler.advanceTimeTo(when, timeUnit);
testScheduler.triggerActions();
```

これにより、TestSchedulerがのあるのすべてのアクションをするがあるときにできます。

スケジュ―ラ―をする、これはどのようにでないかのためにTestSchedulerがにされるではありません。スケジュ―ラをクラスにすことで、しでもくのなコ―ドをすることになります。わりに、RxJavaのSchedulers.io/ computed/ etcにフックできます。これはRxJavaのフックでわれます。これにより、Schedulerメソッドの1つからコ―ルからされるをできます。

```
public final class TestSchedulers {
```

```
    public static TestScheduler test() {
        final TestScheduler testScheduler = new TestScheduler();
        RxJavaHooks.reset();
        RxJavaHooks.setOnComputationScheduler((scheduler) -> {
            return testScheduler;
        });
        RxJavaHooks.setOnIOScheduler((scheduler) -> {
            return testScheduler;
        });
        RxJavaHooks.setOnNewThreadScheduler((scheduler) -> {
            return testScheduler;
        });
        return testScheduler;
    }
}
```

このクラスをすると、スケジューラへのすべてのコールにされるテストスケジューラをできます。テストでは、このスケジューラをセットアップするだけでよいでしょう。 TestSchedulerはあなたがをめるときにのテストからtriggerActionsをみるかもしれないので、セットアップでそれをすることをくおめします。のは

```
TestScheduler testScheduler = new TestScheduler();
TestSubscriber<Integer> subscriber = TestSubscriber.create();
Observable.just(1,2,3)
          .delay(10, TimeUnit.SECONDS, testScheduler)
          .subscribe(subscriber);
testScheduler.advanceTimeBy(9, TimeUnit.SECONDS);
subscriber.assertValues(); // success (delay hasn't finished)
testScheduler.advanceTimeBy(10, TimeUnit.SECONDS);
subscriber.assertValues(1,2,3); // success (delay has finished)
```

これはあなたがユニットテストからシステムクロックをにりくですなくともRxJavaにすること

オンラインでユニットテストをむ https://riptutorial.com/ja/rx-java/topic/5207/ユニットテスト

6: レトロフィットとRxJava

Examples

レトロフィットとRxJavaをセットアップする

Retrofit2はのプラガブルメカニズムをサポートしています。その1つはRxJavaです。

RxJavaでレトロフィットをするには、まずレトロフィットRxJavaアダプタをプロジェクトにするがあります。

```
compile 'com.squareup.retrofit2:adapter-rxjava:2.1.0'
```

レトロフィットインスタンスをするときにアダプタをするがあります。

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.example.com")
    .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
    .build();
```

APIをするのインタフエースでは、りのは`Observable`なければなりません。

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Observable<List<Repo>> listRepos(@Path("user") String user);
}
```

`Observable`ではなく`Single`をすることもできます。

シリアルリクエストの

RxJavaはシリアルリクエストをうときにです。リクエストのをしてのリクエストをするは、`flatMap`をできます。

```
api.getRepo(repoId).flatMap(repo -> api.getUser(repo.getOwnerId())
    .subscribe(/*do something with the result*/);
```

パラレルリクエストの

`zip`をしてしてリクエストし、をすることができます。

```
Observable.zip(api.getRepo(repoId1), api.getRepo(repoId2), (repo1, repo2) ->
    {
        //here you can combine the results
    }).subscribe(/*do something with the result*/);
```

オンラインでレトロフィットとRxJavaをむ https://riptutorial.com/ja/rx-java/topic/2950/レトロフィットとrxjava

7:

このドキュメントでは、オペレータのなについてします。

Examples

オペレーター、

`Observable`から`Subscriber`へのオブジェクトのれをするために、をすることができます。

```
Observable<Integer> integerObservable = Observable.just(1, 2, 3); // creating a simple Integer
observable
Subscriber<String> mSubscriber = new Subscriber<String>() {
    @Override
    public void onCompleted() {
        System.out.println("onCompleted called!");
    }

    @Override
    public void onError(Throwable throwable) {
        System.out.println("onError called");
    }

    @Override
    public void onNext(String string) {
        System.out.println("onNext called with: " + string);
    }
}; // a simple String subscriber

integerObservable
    .map(new Func1<Integer, String>() {
        @Override
        public String call(Integer integer) {
            switch (integer) {
                case 1:
                    return "one";
                case 2:
                    return "two";
                case 3:
                    return "three";
                default:
                    return "zero";
            }
        }
}).subscribe(mSubscriber);
```

はのようになります。

```
onNext called with: one
onNext called with: two
onNext called with: three
onCompleted called!
```

`map`は、 `Integer` observableを`String` observableにし、オブジェクトのフローをしました。

オペレーターチェーン

よりなとのために、のを`chained`させることができます。

```
integerObservable // emits 1, 2, 3
            .map(i -> i + 10) // adds 10 to each item; emits 11, 12, 13
            .filter(i -> i > 11) // emits items that satisfy condition; 12, 13
            .last() // emits last item in observable; 13
            // unlimited operators can be added ...
            .subscribe(System.out::println); // prints 13
```

`Observable`と`Subscriber`には、ののをできます。

flatMap

`flatMap`は、あるイベントをの`Observable`にするのにちますまたは、イベントを1つ、またはのイベントにします。

これは`Observable`をすのメソッドをびすときにはなです

```
 public Observable<String> perform(int i) {
      // ...
 }

 Observable.just(1, 2, 3)
           .flatMap(i -> perform(i))
           .subscribe(result -> System.out.println("result ->" + result);
```

`flatMap`シリアライズされます`perform`ことにより、emitedサブスクリプションが、イベントの`perform`じたことがないかもしれません。したがって、の`perform`コールからのイベントのににされたイベントによってイベントがされるがありますわりに`concatMap`をするがあります。

あなたのサブスクライバでの`Observable`をするは、わりに`flatMap` をするがあります。なアイデアは、Observable

えば

```
 Observable.just(1, 2, 3)
           .subscribe(i -> perform(i));
```

のようににきえることができます

```
 Observable.just(1, 2, 3)
           .flatMap(i -> perform(i))
           .subscribe();
```

Reactivex.ioのドキュメント http : //reactivex.io/documentation/operators/flatmap.html

フィルタ

filterをして、メソッドのについてストリームからをフィルタリングできます。

つまり、オブザーバからサブスクライバにされるアイテムは、 filterをすについてされます。がのにしてfalseをす、そのはされます。

```
List<Integer> integers = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);

Observable.from(integers)
    .filter(number -> {
        return (number  % 2 == 0);
        // odd numbers will return false, that will cause them to be filtered
    })
    .map(i -> {
        return Math.pow(i, 2); // take each number and multiply by power of 2
    })
    .subscribe(onNext -> {
         System.out.println(onNext); // print out the remaining numbers
    });
```

このコードはされます

```
0.0
4.0
16.0
36.0
64.0
```

mapをすると、 mapされたののについて、ストリームのをなるにmap。ストリームはしいコピーであり、されたのストリームをしません。ストリームのさはストリームのさとじですが、なるタイプであるがあります。

.map()されたは、をすがあります。

```
List<Integer> numbers = Arrays.asList(1, 2, 3);
Observable.from(numbers)
    .map(number -> {
        return number.toString(); // convert each integer into a string and return it
    })
    .subscribe(onNext -> {
         System.out.println(onNext); // print out the strings
    });
```

このコードはされます

```
"1"
"2"
"3"
```

このでは、Observableが<Integer>けれると、 <Integer>はパイプラインの<String>にされ、 .subscribeは<String> .subscribeます

doOnNext

ソース`Observable`がアイテムを`doOnNext`たびにびされる`doOnNext`。それは、デバッグのためにされ、されたアイテムにいくつかのアクションをし、ロギングするなど...

```
Observable.range(1, 3)
    .doOnNext(value -> System.out.println("before transform: " + value))
    .map(value -> value * 2)
    .doOnNext(value -> System.out.println("after transform: " + value))
    .subscribe();
```

のでは`doOnNext`ソースためとばれることはありません`Observable`あるため、もしない`Observable.empty()`のびし`onCompleted`した。

```
Observable.empty()
    .doOnNext(item -> System.out.println("item: " + item))
    .subscribe();
```

りし

`repeat`オペレータは、ソースからのシーケンスをりすことができ`Observable`。

```
Observable.just(1, 2, 3)
    .repeat()
    .subscribe(
        next -> System.out.println("next: " + next),
        error -> System.out.println("error: " + error),
        () -> System.out.println("complete")
    );
```

のの

```
next: 1
next: 2
next: 3
next: 1
next: 2
next: 3
```

このシーケンスはりされ、してしません。

シーケンスを`repeat`、をとしてしてオペレータを`repeat`ます。

```
Observable.just(1, 2, 3)
    // Repeat three times and complete
    .repeat(3)
    .subscribe(
        next -> System.out.println("next: " + next),
        error -> System.out.println("error: " + error),
        () -> System.out.println("complete")
    );
```

このでは、

```
next: 1
next: 2
next: 3
next: 1
next: 2
next: 3
next: 1
next: 2
next: 3
complete
```

ソースの`Observable`シーケンスがしたときに、 `repeat`がソース`Observable`にすることをすることはにです。のを`Observable.create`をってきしてみましょう。

```
Observable.<Integer>create(subscriber -> {

    //Same as Observable.just(1, 2, 3) but with output message
    System.out.println("Subscribed");
    subscriber.onNext(1);
    subscriber.onNext(2);
    subscriber.onNext(3);
    subscriber.onCompleted();
})
        .repeat(3)
        .subscribe(
                next -> System.out.println("next: " + next),
                error -> System.out.println("error: " + error),
                () -> System.out.println("complete")
        );
```

このでは、

```
Subscribed
next: 1
next: 2
next: 3
Subscribed
next: 1
next: 2
next: 3
Subscribed
next: 1
next: 2
next: 3
complete
```

をする、 `repeat`がのではなくシーケンスを`repeat`ことがです。

```
Observable.<Integer>create(subscriber -> {
    System.out.println("Subscribed");
    subscriber.onNext(1);
    subscriber.onNext(2);
    subscriber.onNext(3);
    subscriber.onCompleted();
```

```
})
        .map(value -> value * 2) //First chain operator
        .map(value -> "modified " + value) //Second chain operator
        .repeat(3)
        .subscribe(
                next -> System.out.println("next: " + next),
                error -> System.out.println("error: " + error),
                () -> System.out.println("complete")
        );
```

このでは、

```
Subscribed
next: modified 2
next: modified 4
next: modified 6
Subscribed
next: modified 2
next: modified 4
next: modified 6
Subscribed
next: modified 2
next: modified 4
next: modified 6
complete
```

このでは、ことをして`repeat`オペレータは、シーケンスにresubscribingりして`Observable`ではなく、のりし`map`を、シーケンスのどのでもかまいません`repeat`をします。

このシーケンス

```
Observable.<Integer>create(subscriber -> {
        //...
    })
    .map(value -> value * 2) //First chain operator
    .map(value -> "modified " + value) //Second chain operator
    .repeat(3)
    .subscribe(
        /*....*/
    );
```

このシーケンスにしい

```
Observable.<Integer>create(subscriber -> {
        //...
    })
    .repeat(3)
    .map(value -> value * 2) //First chain operator
    .map(value -> "modified " + value) //Second chain operator
    .subscribe(
        /*....*/
    );
```

オンラインでをむ https://riptutorial.com/ja/rx-java/topic/2316/

8:

- サブジェクト<T、R> subject = AsyncSubject.create; //デフォルトのAsyncSubject
- Subject <T、R> subject = BehaviorSubject.create; //デフォルトのBehaviorSubject
- サブジェクト<T、R> subject = PublishSubject.create; //デフォルトのPublishSubject
- Subject <T、R> subject = ReplaySubject.create; //デフォルトのReplaySubject
- mySafeSubject =しいSerializedSubjectunSafeSubject; // unsafeSubjectをsafeSubjectにする - にはマルチスレッド

パラ メーター

パラ メーター	
T	
R	タイプ

このドキュメントは、 `Subject`にするとをします。およびは、 をごください。

Examples

RxJavaの`Subject`は`Observable`と`Observer`のクラスです。 これはに`Observable`としてし、をにし、`Observer`としてのObservableからをることをします。

```
Subject<String, String> subject = PublishSubject.create();
subject.subscribe(System.out::print);
subject.onNext("Hello, World!");
```

の "Hello、World" `Subjects`をってコンソールに

1. コードののは、 `PublishSubject`のしい`Subject`をします

   ```
   Subject<String, String> subject = PublishSubject.create();
       |      |        |        |                  |
    subject<input, output>  name    = default publish subject
   ```

2. 2はサブジェクトにサブスクライブし、 `Observer`をします。

   ```
   subject.subscribe(System.out::print);
   ```

   これにより、 `Subject`はのサブスクライバのようにをけります

3. 3では、 `onNext`メソッドをびし、 `Observable`ビヘイビアをします。

```
subject.onNext("Hello, World!");
```

これにより、 `Subject`は`Subject`にサブスクライブするすべてのにをえることができます。

タイプ

`Subject` RxJavaは、の4つのタイプのいずれかになります。

- AsyncSubject
-
- PublishSubject
- ReplaySubject

また、 `Subject`は`SerializedSubject`でも`SerializedSubject`ません。このタイプは、 `Subject`が *Observable Contract* すべてのびしをシリアライズするがあることをするにしないことをします。

- Dave Sextonのブログから`Subject`をするかしないか

PublishSubject

`PublishSubject`は、サブスクリプションのにいて`Observable`ソースによって`PublishSubject`されたアイテムのみを`Observer`します。

な`PublishSubject`

```
Observable<Long> clock = Observable.interval(500, TimeUnit.MILLISECONDS);
Subject<Long, Long> subjectLong = PublishSubject.create();

clock.subscribe(subjectLong);

System.out.println("sub1 subscribing...");
subjectLong.subscribe(l -> System.out.println("sub1 -> " + l));
Thread.sleep(3000);
System.out.println("sub2 subscribing...");
subjectLong.subscribe(l -> System.out.println("sub2 -> " + l));
Thread.sleep(5000);
```

```
sub1 subscribing...
sub1 -> 0
sub1 -> 1
sub2 subscribing...
sub1 -> 2
sub2 -> 2
sub1 -> 3
sub2 -> 3
```

のでは、 `PublishSubject`はのようにする`Observable`をサブスクライブし、500ミリごとにアイテムLongを`PublishSubject`ます。にられるように、 `PublishSubject` 、ソースからバレスに`clock`そのへ`sub1`および`sub2` 。

`PublishSubject`は、オブザーバーがなくても、オブザーバーがするまで1つまたはのアイテムがわ

れるがある、されるとすぐにアイテムのをすることができます。

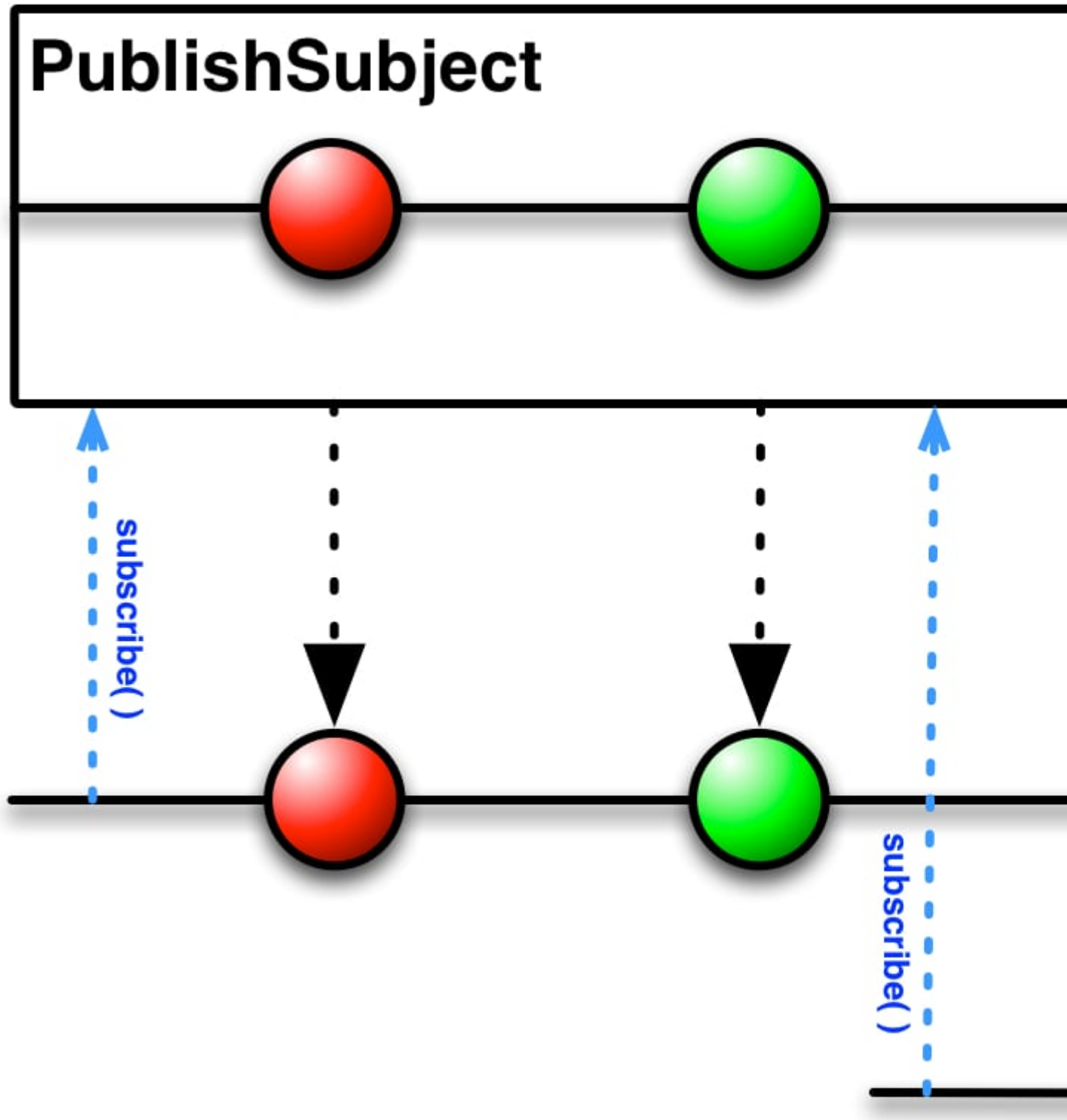
```
createClock(); // 3 lines moved for brevity. same as above example

Thread.sleep(5000); // introduces a delay before first subscribe

sub1andsub2(); // 6 lines moved for brevity. same as above example
```

```
sub1 subscribing...
sub1 -> 10
sub1 -> 11
sub2 subscribing...
sub1 -> 12
sub2 -> 12
sub1 -> 13
sub2 -> 13
```

`sub1`は`10`からまるをする`sub1`してください。された5のれは、アイテムのをきこした。これらはすることはできません。これはに`PublishSubject`を`Hot Observable`ます。

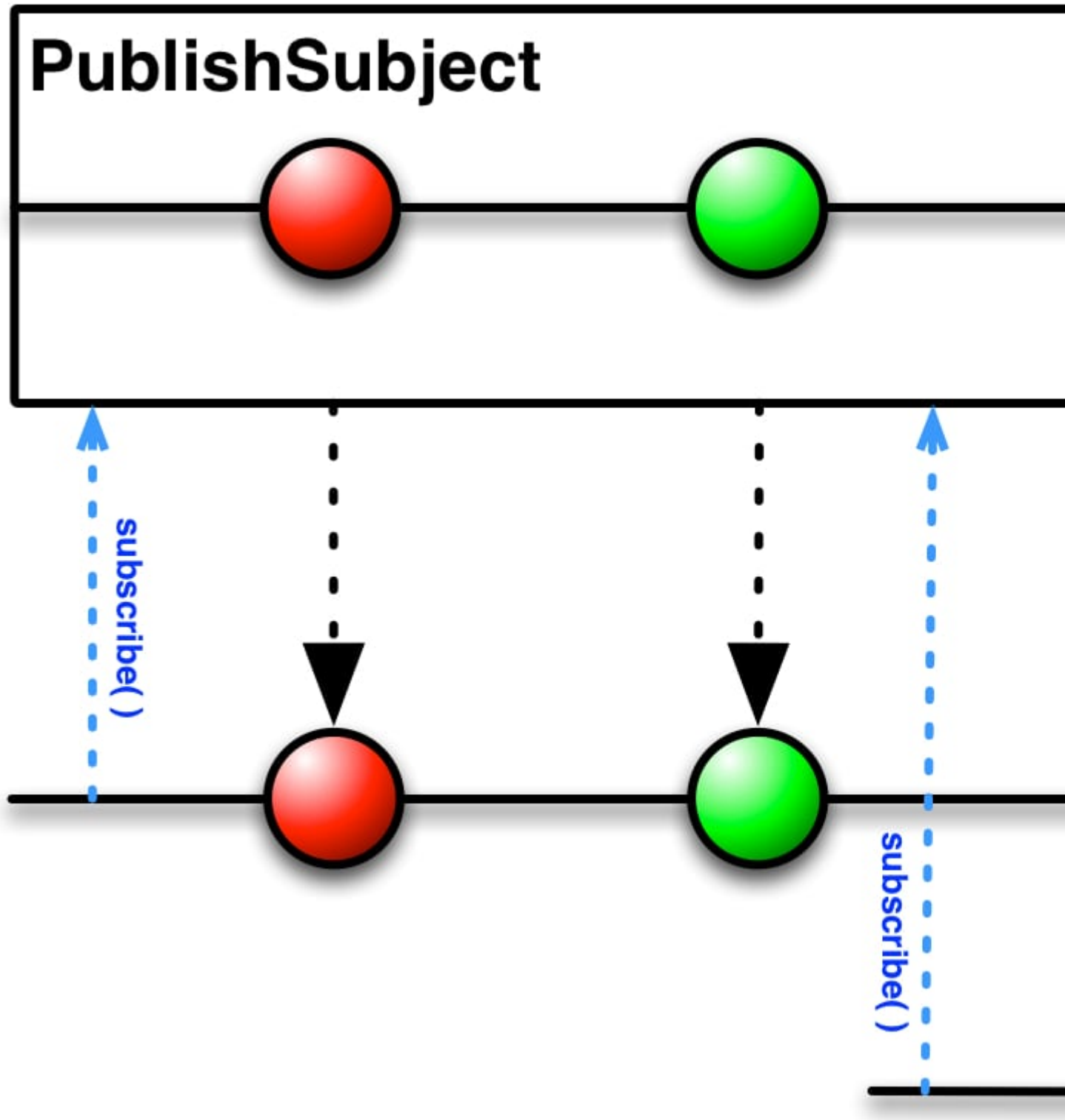
また、オブザ―バ―が***n***のアイテムを`PublishSubject`にオブザ―バ―が`PublishSubject`にサブスクライブすると、これらの***n***のアイテムはこのオブザ―バ―のためにできません 。

は、 `PublishSubject`のです



`onCompleted`は、 `Observable`ソースの`onCompleted`がびされるのので、したすべての`PublishSubject`アィテムをします。

`Observable`ソースがエラーでした、 `PublishSubject`はのオブザーバにアイテムを`PublishSubject`ませんが、にObservableソースからのエラーをします。



ののをし、それをしたすべてのクライアントにするアプリケーションをするとします。

```
/* Dummy stock prices */
Observable<Integer> prices = Observable.just(11, 12, 14, 11, 10, 12, 15, 11, 10);

/* Your server */
PublishSubject<Integer> watcher = PublishSubject.create();
/* subscribe to listen to stock price changes and push to observers/clients */
prices.subscribe(watcher);
```

```
/* Client application */
stockWatcher = getWatcherInstance(); // gets subject
Subscription steve = stockWatcher.subscribe(i -> System.out.println("steve watching " + i));
Thread.sleep(1000);
System.out.println("steve stops watching");
steve.unsubscribe();
```

のでは、 `PublishSubject`は、サーバーから`watcher`しているすべてのクライアントにをすためのブリッジとしてします。

- PublishSubject javadocs
- Thomas Nieldのブログ

オンラインでをむ https://riptutorial.com/ja/rx-java/topic/3287/

9:

Examples

き

バックプレッシャーは、 `Observable`パイプラインでは、のステージではをくできず、のプロデューサにくするようするがです。

バックプレッシャーののなケースは、があるときです

```
PublishSubject<Integer> source = PublishSubject.create();

source
.observeOn(Schedulers.computation())
.subscribe(v -> compute(v), Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}

Thread.sleep(10_000);
```

このでは、メインスレッドは、バックエンドスレッドでそれをしているエンドコンシューマに100アイテムをします。 `compute(int)`メソッドにはがかかるがありますが、 `Observable`チェーンのオーバーヘッドがアイテムのにかかるをやすこともあります。しかし、forループのスレッドはこれをることができず、 `onNext`をします。

には、には、できるようになるまでそのようなをするバッファがあります。なRx.NETとのRxJavaでは、これらのバッファにはがありませんでした。つまり、このからほぼ100のすべてをするがあります。このは、えば10のがする、または100のシーケンスがプログラムに1000し、`OutOfMemoryError`につながり、なGCオーバーヘッドのためににする`OutOfMemoryError`します。

エラーは、オブジェクトとなり、をして、それにするためのをするとに`onErrorXXX`オペレーター、はプログラマがえるとするためにっているデータフローののである`onBackpressureXXX`オペレーターが。

の`PublishSubject`にも、をサポートしていないのがあります。これはになによるものです。たとえば、オペレータの`interval`はにをし、バックプレッシャはとのなのシフトにつながります。

のRxJavaでは、ほとんどのがの`observeOn`ようなバッファを`observeOn`ようになりました。このバッファをオーバーフローさせると、シーケンスが`MissingBackpressureException`します。オペレータのドキュメントには、のについてのがあります。

しかし、はのコールドシーケンス `MissingBackpressureException`ていないし、そうでなければならないにもっとにします。のがきされた

```
Observable.range(1, 1_000_000)
.observeOn(Schedulers.computation())
.subscribe(v -> compute(v), Throwable::printStackTrace);

Thread.sleep(10_000);
```

エラ―はなく、さなメモリですべてがスム―ズにされます。このは、くのソ―スオペレ―タ―はオンデマンドを「」することができますので、オペレ―タということです`observeOn`えることができる`range`、せいぜいにくのが`observeOn`バッファがオ―バ―フロ―することなく、にすることができます。

このはコンピュ―タ･サイエンスのコル―チンのにづいていますはあなたをにびます。オペレ―タ`range`のでは、コ―ルバックをする`Producer`に、インタ―フェ―ス`observeOn`そのびすことによって、`Subscriber`の`setProducer`。りに、`observeOn`びし`Producer.request(n)`すると`range`すなわち、するようにされているが`onNext`くのことを。なに`request`メソッドをびし、デ―タがれてもオ―バ―フロ―しないようにしいをするのは`observeOn`のです。

エンド･コンシュ―マのバックプレッシャのはほとんどありませんコ―ルスタック･ブロッキングのためにとバックプレッシャにしてしているのでにしますが、そのをすることはです。

```
Observable.range(1, 1_000_000)
.subscribe(new Subscriber<Integer>() {
    @Override
    public void onStart() {
        request(1);
    }

    public void onNext(Integer v) {
        compute(v);

        request(1);
    }

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

    @Override
    public void onCompleted() {
        System.out.println("Done!");
    }
});
```

ここで、`onStart`は、のをする`range`をし、その、`onNext`けられ`onNext`。`compute(int)`すると、`range`からのがされ`range`。`range`なで`range`、そのようなびしはに`onNext`びす`onNext`、もちろんましくない`StackOverflowError`します。

これをぐために、オペレ―タは、このようなリエントラントコ―ルをする、いわゆるトランポリングロジックをします。`range`のでは、`onNext()`をびしていて`onNext()`がったときに`request(1)`コ―ルがあったことをえていて、ので`onNext()`をびしてびします。したがって、2つがスワップされても、はじようにします。

```
@Override
public void onNext(Integer v) {
    request(1);

    compute(v);
}
```

しかし、これは`onStart`はてはまりません。 `Observable`インフラストラクチャでは、`Subscriber`でも1びされることがされていますが、 `request(1)`びしはすぐにのをトリガ―するがあります。`onNext`がとする`request(1)`びしのにロジックがある、がするがあります。

```
Observable.range(1, 1_000_000)
.subscribe(new Subscriber<Integer>() {

    String name;

    @Override
    public void onStart() {
        request(1);

        name = "RangeExample";
    }

    @Override
    public void onNext(Integer v) {
        compute(name.length + v);

        request(1);
    }

    // ... rest is the same
});
```

このの、 `onStart`にすぐに`NullPointerException`がスロ―されます。 `request(1)`びしによって、のスレッドで`onNext`へのびしが`onNext`れ、 `onNext`レ―スの`name`が`onStart`ポスト`request`きまれる、よりなバグがします。

したがって、 `onStart`フィ―ルドをうか、それよりであっても`request()`にびすがあります。オペレ―タの`request()`は、に`request()`なまたはメモリまたはフェンスをします。

onBackpressureXXX

ほとんどのは、アプリケ―ションが`MissingBackpressureException`でしたときににし、は`observeOn`をしています。のは、、 `PublishSubject`、 `timer()`または`interval()`や`create()`されたカスタムの`PublishSubject`です。

このようなにするにはいくつかのがあります。

バッファサイズの

にはそのようなオ―バ―フロ―はバ―ストなソ―スのためにこります。、ユ―ザ―はをあまりに

もくタップし、Androidのデフォルトの16のバッファーがオーバーフローすることを`observeOn`ます。

のバージョンのRxJavaのになのほとんどは、プログラマがバッファのサイズをできるようになりました。するパラメータは、 `bufferSize` 、 `prefetch`または`capacityHint`とばれます。のがあふれているので、 `observeOn`のバッファサイズを`observeOn`て、すべてのになをたせることができます。

```
PublishSubject<Integer> source = PublishSubject.create();

source.observeOn(Schedulers.computation(), 1024 * 1024)
      .subscribe(e -> { }, Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

ただし、ソースがされたバッファサイズをすると、オーバーフローがするがあるため、これはになにぎません。この、ののいずれかをできます。

でをバッチ/スキップする

ソースデータをよりにバッチできるは、のバッチの1つサイズおよび/またはによるをして`MissingBackpressureException`のをらすことができます。

```
PublishSubject<Integer> source = PublishSubject.create();

source
      .buffer(1024)
      .observeOn(Schedulers.computation(), 1024)
      .subscribe(list -> {
          list.parallelStream().map(e -> e * e).first();
      }, Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

のをにできるは、サンプリングまたはのObservableとりみ `throttleFirst` 、 `throttleLast` 、 `throttleWithTimeout` をできます。

```
PublishSubject<Integer> source = PublishSubject.create();

source
      .sample(1, TimeUnit.MILLISECONDS)
      .observeOn(Schedulers.computation(), 1024)
      .subscribe(v -> compute(v), Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

これらのは、ののをさせるだけなので、 `MissingBackpressureException`がするがあります。

onBackpressureBuffer

パラメ―タなしののこのは、のとののにのバッファをします。とは、JVMのメモリがしているり、バ―ストなソ―スからのほとんどすべてのをできます。

```
Observable.range(1, 1_000_000)
          .onBackpressureBuffer()
          .observeOn(Schedulers.computation(), 8)
          .subscribe(e -> { }, Throwable::printStackTrace);
```

このでは、 `observeOn`にいバッファサイズとくまだありません`MissingBackpressureException`と`onBackpressureBuffer`へのさなバッチをえるすべての100のとをす`observeOn`。

ただし、 `onBackpressureBuffer`はソ―スをにします。つまり、 `onBackpressureBuffer`をしません。これは、 `range`などのがにされるというをもたらす。

`onBackpressureBuffer`さらに4つのオ―バ―ロ―ドがあり`onBackpressureBuffer`

onBackpressureBufferint capacity

これは、そのバッファがのにしたに、 `BufferOverflowError`するバ―ジョンです。

```
Observable.range(1, 1_000_000)
          .onBackpressureBuffer(16)
          .observeOn(Schedulers.computation())
          .subscribe(e -> { }, Throwable::printStackTrace);
```

ますますくのがバッファサイズをできるようになれば、こののはしています。りのでは、これは`onBackpressureBuffer`を`onBackpressureBuffer`よりきなをデフォルトよりきくすることによって、"バッファをする"をえます。

onBackpressureBufferint capacity、Action0 onOverflow

このオ―バ―ロ―ドは、オ―バ―フロ―がしたにえてアクションをびします。そのは、のびしスタックよりもオ―バ―フロ―にしてされるのがないので、かなりられています。

onBackpressureBufferint capacity、Action0 onOverflow、BackpressureOverflow.Strategy

このは、にはにしたにをすべきかをするので、にはよりです。 `BackpressureOverflow.Strategy`はにはインタ―フェイスですが、 `BackpressureOverflow`クラスにはなアクションをす4つのフィ―ルドがされています。

- `ON_OVERFLOW_ERROR` これはの2つのオ―バ―ロ―ドのデフォルトので、`BufferOverflowException`します
- `ON_OVERFLOW_DEFAULT` は`ON_OVERFLOW_ERROR`とじ`ON_OVERFLOW_ERROR`

- `ON_OVERFLOW_DROP_LATEST` オーバーフローがした、のはにされ、いだけがダウンストリームにされます。
- `ON_OVERFLOW_DROP_OLDEST` バッファのもいをし、のをします。

```
Observable.range(1, 1_000_000)
          .onBackpressureBuffer(16, () -> { },
              BufferOverflowStrategy.ON_OVERFLOW_DROP_OLDEST)
          .observeOn(Schedulers.computation())
          .subscribe(e -> { }, Throwable::printStackTrace);
```

の2つのストラテジは、がされるときにストリームにをきこすことにしてください。さらに、らは`BufferOverflowException`しません。

onBackpressureDrop

ダウンストリームがをけるができていないときはいつでも、このはシーケンスからそのエレメンツをします。 1つのストラテジを`ON_OVERFLOW_DROP_LATEST` `onBackpressureBuffer`して、0のの`onBackpressureBuffer`とえることができます。

このは、でさらにのがあるため、ソースからのマウスやのGPSなどをにできるにです。

```
component.mouseMoves()
.onBackpressureDrop()
.observeOn(Schedulers.computation(), 1)
.subscribe(event -> compute(event.x, event.y));
```

ソースの`interval()`とみわせてするとです。たとえば、なバックグラウンド･タスクをしたいが、がよりもくくは、あとでうようになをすることがです。

```
Observable.interval(1, TimeUnit.MINUTES)
.onBackpressureDrop()
.observeOn(Schedulers.io())
.doOnNext(e -> networkCall.doStuff())
.subscribe(v -> { }, Throwable::printStackTrace);
```

`onBackpressureDrop(Action1<? super T> onDrop)`ここで、sharedアクションがびされ、がされます。このでは、そのものをクリーンアップすることができますえば、リソースの。

onBackpressureLatest

のはののみをし、にはい、のをきします。これは、1のと`ON_OVERFLOW_DROP_OLDEST`をつ`onBackpressureBuffer`とえることができます。

`onBackpressureDrop`とはなり、ダウンストリームがれてしまった、にながあります。これは、データがバーストなパターンでれることがありますが、にのものだけがにがあるテレメトリのようなではにちます。

たとえば、ユーザーがでたくさんのボタンをクリックしたでも、のにしたいとえています。

```
component.mouseClicks()
.onBackpressureLatest()
.observeOn(Schedulers.computation())
.subscribe(event -> compute(event.x, event.y), Throwable::printStackTrace);
```

この、 `onBackpressureDrop`をすると、のクリックが`onBackpressureDrop` 、ビジネスロジックがされなかったがユーザーに`onBackpressureDrop`ます。

データソースの

ライブラリがにのバック`Observable`をうメソッドを`Observable`しているので、のデータソースをすることは、バックプレッシャをうのなです。のにづいてをしてするたい「ジェネレータ」と、、のデータソースおよび/またはバックラブルのデータソースをしするホット「プッシャ」と、それら。

ちょうど

もなをソースがでされた`just`。

```
Observable.just(1).subscribe(new Subscriber<Integer>() {
    @Override
    public void onStart() {
        request(0);
    }

    @Override
    public void onNext(Integer v) {
        System.out.println(v);
    }

    // the rest is omitted for brevity
}
```

たちはに`onStart`でリクエストしないので、もされません。たちがシーケンスをジャンプ·スタートしたいというのがあるときには、 `just`です。

ながら、 `just`くの、でされるためにかをするをえている`Subscriber`

```
int counter;

int computeValue() {
   return ++counter;
}

Observable<Integer> o = Observable.just(computeValue());

o.subscribe(System.out:println);
o.subscribe(System.out:println);
```

くべきことに、これは1と2のわりに1を2します。コールがきえられた、なぜそれがするのかがになります。

```
int temp = computeValue();

Observable<Integer> o = Observable.just(temp);
```

computeValueは、メインルチンのとしてびされ、サブスクライバのサブスクリプcomputeValueへのではありません。

fromCallable

々がにとするのは、メソッドfromCallableです。

```
Observable<Integer> o = Observable.fromCallable(() -> computeValue());
```

ここでは、 computeValueはサブスクライバがサブスクライブするときにのみされ、それぞれにしてされる1と2をfromCallableます。もちろん、 fromCallableもfromCallableをにサポトし、されないりをしません。しかし、はとにかくこることにしてください。にがにするまでをらせるがあるは、 map justをうことができます

```
Observable.just("This doesn't matter").map(ignored -> computeValue())...
```

just 、それがにマッピングされるときにされるまでそのをしないであろうcomputeValue 、として々のをめました。

から

データがオブジェクトの、オブジェクトのリスト、またはのIterableソースとしてにな、それぞれのfrom Iterableおよびそのようなソースのがされます。

```
 Observable.from(Arrays.asList(1, 2, 3, 4, 5)).subscribe(System.out::println);
```

、ジェネリックのにするをけるために、そこfromにされるjustに210のオバロードがあります。

from(Iterable)もいをえます。くのは、のでることができます。されたは、とりのをトリガします。

IterableのようなステトマシンをIterableはややですが Observableをするために Observableをくよりもです、Cとはなり、Javaはコンパイラからクラシックなコードをくだけでコンパイラからのサポートはありません yield returnとyield break。のライブラリは、Google GuavaのAbstractIterableやIxJavaのIx.generate()やIx.forloop()などのヘルプをしています。これらはそれでなシリズにふさわしいので、あるのをにりすになIterableソースをてみましょう

```
Iterable<Integer> iterable = () -> new Iterator<Integer>() {
    @Override
    public boolean hasNext() {
        return true;
    }

    @Override
    public Integer next() {
        return 1;
    }
};

Observable.from(iterable).take(5).subscribe(System.out::println);
```

なfor-loopをしてiteratorをすると、ループになります。 Observableをするので、の5つだけをし、もしないをすることができます。これはObservableのでなとをうのです。

SyncOnSubscribe

には、なにされるデータソースは、ブロッキングとプルです。つまり、のデータをするために、いくつかのgetメソッドまたはreadメソッドをびすgetあります。もちろん、それをIterableえることはできますが、そのようなソースがリソースにけられている、ダウンストリームがするにシーケンスをアンIterableすると、それらのリソースがリークするがあります。

このようなケースをするために、RxJavaにはSyncOnSubscribeクラスがあります。それをしてメソッドをするか、ラムダベースのファクトリメソッドの1つをしてインスタンスをすることができます。

```
SyncOnSubscribe<Integer, InputStream> binaryReader = SyncOnSubscribe.createStateful(
     () -> new FileInputStream("data.bin"),
     (inputstream, output) -> {
         try {
             int byte = inputstream.read();
             if (byte < 0) {
                 output.onCompleted();
             } else {
                 output.onNext(byte);
             }
         } catch (IOException ex) {
             output.onError(ex);
         }
         return inputstream;
     },
     inputstream -> {
         try {
             inputstream.close();
         } catch (IOException ex) {
             RxJavaHooks.onError(ex);
         }
     }
 );

 Observable<Integer> o = Observable.create(binaryReader);
```

に、 `SyncOnSubscribe`は3つのコールバックをします。

のコールバックでは、このの`FileInputStream`など、サブスクライバごとのをできます。ファイルは々のごとにしてかれます。

2のコールバックはこのオブジェクトをとり、 `onXXX`メソッドがびされてをできる`Observer`をします。このコールバックは、ダウンストリームでされただけされます。びしごとに`onError`または`onCompleted`いずれかがにくは、くても`onNext`をびすがあります。このでは、みみバイトがのは`onCompleted()`をびし、ファイルのとをし、みみが`IOException`スローするは`onError`をびします。

ダウンストリームのサブスクライブストリームをじるまたはのコールバックがターミナルメソッドをびしたときに、のコールバックがびされます。リソースをすることができます。すべてのソースがこれらのをすべてとするわけではないので、 `SyncOnSubscribe`のメソッドをすると、インスタンスなしでインスタンスがされます。

なことに、JVMやのライブラリのメソッドびしのくは、チェックをスローし、このクラスでされるインタフェースではチェックをげられないため、 `try-catch`ラップするがあります。

もちろん、のなど、のなソースをすることもできます。

```
SyncOnSubscribe.createStateful(
     () -> 0,
     (current, output) -> {
         output.onNext(current);
         return current + 1;
     },
     e -> { }
);
```

このでは、 `current`は`0`まり、ラムダがびされたはパラメータ`current`が`1`ます。

ミドルコールバックにはダウンストリームからのリクエストをすいがで、コールバックはまったくじさの`Observable`をするがあるというとによくた`SyncOnSubscribe`という`AsyncOnSubscribe`があります。このソースは、これらの`Observable`をすべて1つのシーケンスにします。

```
 AsyncOnSubscribe.createStateful(
     () -> 0,
     (state, requested, output) -> {
         output.onNext(Observable.range(state, (int)requested));
         return state + 1;
     },
     e -> { }
 );
```

このクラスのについてのながあり、にされません。なぜなら、それらのされたがどのようににどのようにされ、どのようにするか、よりなシナリオ

エミッタ

`Observable`ラップされるソースは、マウスのきなどでにいですが、のネットワークコールバックなど、APIではバックプレッシャーではありません。

このようなケースをするために、RxJavaののバージョンでは`create(emitter)`ファクトリメソッドがされました。 2つのパラメータがです。

- ごとに`Emitter<T>`インタフェースのインスタンスでびされるコールバック、
- がのをするようにする`Emitter.BackpressureMode`。 `onBackpressureXXX`にたのモードがあり、`MissingBackpressureException`をするだけでなく、でそのようなオーバーフローをにすることもできます。

のところ、これらのバックプレッシャモードへのパラメータはサポートしていないことにしてください。これらのカスタマイズがなは、 `onBackpressureXXX`プレッシャーモードとして`NONE`をし、する`onBackpressureXXX`をとしての`Observable`にするがあります。

GUIイベントなど、プッシュベースのソースとしたいののなケースです。これらのAPIには、できる`addListener` / `removeListener`びしのいくつかのがあります。

```
Observable.create(emitter -> {
    ActionListener al = e -> {
        emitter.onNext(e);
    };

    button.addActionListener(al);

    emitter.setCancellation(() ->
        button.removeListener(al));

}, BackpressureMode.BUFFER);
```

`Emitter`はにできます。 `onNext` 、 `onError`および`onCompleted`をびすことができ、オペレータは`onCompleted`および`onCompleted`をでします。さらに、ラップされたAPIがりしこのではリスナーのなどをサポートしている、ダウンストリームのサブスク`setSubscription`や`onError` / `setSubscription`がびされたときにびされるりしコールバックをするために`setCancellation` または`Subscription`ようなリソースにして`setSubscription` `onCompleted`は、された`Emitter`インスタンスでびされます。

これらのメソッドでは、に1つのリソースだけをエミッタにけることができ、しいリソースをすると、いものをにりします。のリソースをするがあるは、 `CompositeSubscription`し、それをエミッタにけてから、 `CompositeSubscription`にさらにリソースをします。

```
Observable.create(emitter -> {
    CompositeSubscription cs = new CompositeSubscription();

    Worker worker = Schedulers.computation().createWorker();

    ActionListener al = e -> {
        emitter.onNext(e);
    };

    button.addActionListener(al);
```

```
    cs.add(worker);
    cs.add(Subscriptions.create(() ->
        button.removeActionListener(al));

    emitter.setSubscription(cs);

}, BackpressureMode.BUFFER);
```

2のシナリオでは、 `Observable`するのあるのコールバックベースのAPIがまれています。

```
Observable.create(emitter -> {

    someAPI.remoteCall(new Callback<Data>() {
        @Override
        public void onSuccess(Data data) {
            emitter.onNext(data);
            emitter.onCompleted();
        }

        @Override
        public void onFailure(Exception error) {
            emitter.onError(error);
        }
    });

}, BackpressureMode.LATEST);
```

この、はじようにします。なことに、、これらのなコールバックスタイルのAPIはキャンセルをサポートしていませんが、そうした、previoiusののようにおそらくよりなでキャンセルをセットアップできます。 `LATEST`プレッシャモードのにしてください。のしかしないことがかっているは、してにされないデフォルト128のロングバッファにじてするをりてるため、 `BUFFER`はありません。

オンラインでをむ https://riptutorial.com/ja/rx-java/topic/2341/

10: な

Examples

Observableをする

ObservableをRxJavaでするはいくつかあります。もなは、 `Observable.create`メソッドをすることです。しかし、それはまた、もなです。だからなりしないでください 。

のをす

すでにがあるは、 `Observable.just`をしてをできます。

```
Observable.just("Hello World").subscribe(System.out::println);
```

すべきをする

まだされていない、またはにがかかるをするは、 `Observable.fromCallable`をしてのをできます。

```
Observable.fromCallable(() -> longComputation()).subscribe(System.out::println);
```

`longComputation()`は`Observable`をするときにのみびされます。このでは、がになります。

すべきをするの

`Observable.defer`ビルドを`Observable`だけのよう`Observable.fromCallable`いますが、するがあるにはされている`Observable`のわりに。のエラーをするにです。

```
Observable.defer(() -> {
          try {
                return Observable.just(longComputation());
          } catch(SpecificException e) {
                return Observable.error(e);
          }).subscribe(System.out::println);
```

ホットコールドオブザーバブル

は、そのにじて、 `Hot`か`Cold`かにしてされる。
`Cold Observable`は、リクエストサブスクリプションにじてするのにし、 `Hot Observable`はサブスクリプションになくします。

たいな

```
/* Demonstration of a Cold Observable */
Observable<Long> cold = Observable.interval(500, TimeUnit.MILLISECONDS); // emits a long every
500 milli seconds
cold.subscribe(l -> System.out.println("sub1, " + l)); // subscriber1
Thread.sleep(1000); // interval between the two subscribes
cold.subscribe(l -> System.out.println("sub2, " + l)); // subscriber2
```

のコーードのはのようになりますするがあります。

```
sub1, 0    -> subscriber1 starts
sub1, 1
sub1, 2
sub2, 0    -> subscriber2 starts
sub1, 3
sub2, 1
sub1, 4
sub2, 2
```

sub2がくまっても、からをけることにしてください。として、 Cold Observableは、されたときにのみをす。のがのパイプラインをします。

ホット

ホットオブジェクトは、々のサブスクリプションとはしたをします。らはのタイムラインをち、かがいているかどうかにかかわらずイベントがします。

Cold Observaleはなpublish Hot Observableできます。

```
Observable.interval(500, TimeUnit.MILLISECONDS)
    .publish(); // publish converts cold to hot
```

publishは、Observableへのとの ためのをするConnectableObservableをします。

```
ConnectableObservable<Long> hot = Observable
                                    .interval(500, TimeUnit.MILLISECONDS)
                                    .publish(); // returns ConnectableObservable
hot.connect(); // connect to subscribe

hot.subscribe(l -> System.out.println("sub1, " + l));
Thread.sleep(1000);
hot.subscribe(l -> System.out.println("sub2, " + l));
```

のはのとおりです。

```
sub1, 0  -> subscriber1 starts
sub1, 1
sub1, 2
sub2, 2  -> subscriber2 starts
```

```
sub1, 3
sub2, 3
```

にもかかわらずことにしてください`sub2`くをし、それはとしている`sub1`。
はもうしです Disconnectは`Observable`ではなく`Subscription`でわれます。

```
ConnectableObservable<Long> hot = Observable
                                    .interval(500, TimeUnit.MILLISECONDS)
                                    .publish(); // same as above
Subscription subscription = hot.connect(); // connect returns a subscription object, which we
store for further use

hot.subscribe(l -> System.out.println("sub1, " + l));
Thread.sleep(1000);
hot.subscribe(l -> System.out.println("sub2, " + l));
Thread.sleep(1000);
subscription.unsubscribe(); // disconnect, or unsubscribe from subscription

System.out.println("reconnecting");
/* reconnect and redo */
subscription = hot.connect();
hot.subscribe(l -> System.out.println("sub1, " + l));
Thread.sleep(1000);
hot.subscribe(l -> System.out.println("sub2, " + l));
Thread.sleep(1000);
subscription.unsubscribe();
```

はをする

```
sub1, 0   -> subscriber1 starts
sub1, 1
sub1, 2
sub2, 2   -> subscriber2 starts
sub1, 3
sub2, 3
reconnecting  -> reconnect after unsubscribe
sub1, 0
...
```

されると`Observable`に「」し、しいサブスクリプションがされるとします。

`Hot Observable`は`EventBus`にできます。このようなEventBusはにくてです。 RxBusののは、すべてのイベントをでしてバスにすがあることです。

オンラインでなをむ https://riptutorial.com/ja/rx-java/topic/1418/な

クレジット

S. No		Contributors
1	rx-javaをいめる	Buttink, Community, dimsuz, Dmitry Avtonomov, Hans Wurst, hello_world, Omar Al Halabi, Saulius Next, Sneh Pandya, svarog, Tom
2	Android with RxJava	akarnokd, Athafoud, Daniele Segato, Eugen Martynov, Geng Jiawen, Sneh Pandya
3	RxJava2 FlowableおよびSubscriber	P.J.Meisch
4	スケジューラ	dwursteisen, Gal Dreiman
5	ユニットテスト	Buttink, Sir Celsius
6	レトロフィットとRxJava	LordRaydenMK
7		dwursteisen, hello_world, svarog, Vadeg
8		hello_world, mavHarsha
9		akarnokd, Bartek Lipinski, Chris A, Cristian, dwursteisen, Niklas, Sebas LG
10	な	Aki K, dwursteisen, hello_world, JonesV