



免费电子书

学习

rx-java

Free unaffiliated eBook created from
Stack Overflow contributors.

#rx-java

	1
1: rx-java.....	2
.....	2
.....	2
Examples.....	2
.....	2
.....	3
RxJava.....	3
.....	4
2: AndroidRxJava.....	6
.....	6
Examples.....	6
RxAndroid - AndroidSchedulers.....	6
RxLifecycle.....	6
RxPermissions.....	7
3: RxJava2 FlowableSubscriber.....	9
.....	9
.....	9
Examples.....	9
.....	9
4:.....	12
.....	12
.....	12
.....	12
Examples.....	12
.....	12
PublishSubject.....	13
5:.....	17
.....	17
Examples.....	17
TestSubscriber.....	17

29	29
fromCallable.....	30
.....	30
SyncOnSubscribe.....	31
.....	32
9:	34
Examples.....	34
.....	34
10:	36
.....	36
Examples.....	36
.....	36
flatMap.....	37
.....	37
.....	38
doOnNext.....	38
.....	38
.....	42

You can share this PDF with anyone you feel could benefit from it, download the latest version from: [rx-java](#)

It is an unofficial and free rx-java ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official rx-java.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

1: rx-java

rx-java。

RxJava Reactive Extensions Java VM。

Wiki Home RxJava。

1.x	1.3.0	201755
2.X	2.1.1	2017621

Examples

rx-java

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

4. JFrog

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

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

5. jar Maven pom

```
<?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>

```

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

```
rxjava-*.jar./target/dependency/.
```

Java 6。

Hello, World!

```

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);
            }
        });
}

```

Java 8 lambda

```

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

```

RxJava

RxJava Observables Subscribers。 Observable Subscriber。

Observable。 Observables。 observable。 Observable Observable。

```

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

```

integerObservable stringObservable just RX。 Observable。

Subscriber。 Subscriber **observable**。 ObservableSubscriberonNext() onCompleted() onError()。

```
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!");
    }

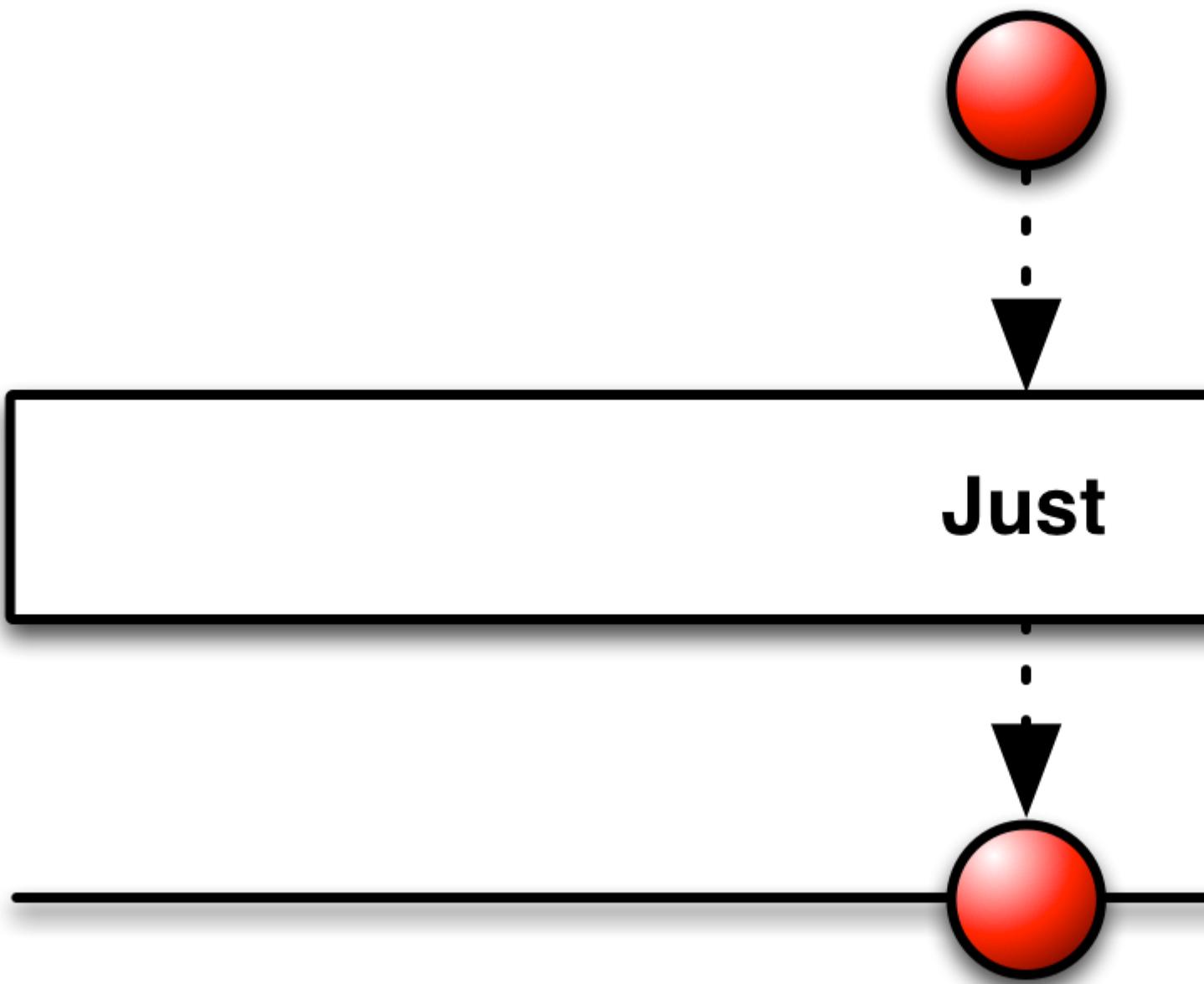
    @Override
    public void onNext(Integer integer) {
        // called for each object that is emitted
        System.out.println("onNext called with: " + integer);
    }
};
```

Subscriber。 Subscriber**observable**subscribesubscribe **observable**。

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

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

Observable。 ObservableonNextonCompleteonError。 observableonNext。 Observable
onComplete。 ObservableonError。 Rx。 Observable“Just”。



X。 “Just”onNextonComplete。 “Just”。 [ReativeX](#)[ReactiveX](#)[RxJava](#)。 [RxMarbles](#)。

[rx-java](#) <https://riptutorial.com/zh-CN/rx-java/topic/974/rx-java>

2: AndroidRxJava

RxAndroid。0.25.01.x。

1.0。

Examples

RxAndroid - AndroidSchedulers

AndroidRxJava。

gradleRxJavaRxAndroid

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

RxAndroidRxJavaAndroidUI。

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

Looper

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

RxJava。

RxLifecycle

RxLifecycleAndroid。

Observable/。

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

```
compile 'com.trello:rxlifecycle-components:0.6.1'
```

Rx*

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

Observable

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

onCreate() onDestroy() ◦

- onStart() -> onStop()
- onResume() -> onPause()
- onAttach() -> onDetach()
- onViewCreated() -> onDestroyView()

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

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

lifecycle() lifecycle()◦

RxLifecycle

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

SingleCompletable bind forSingle() forCompletable

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

Navi◦

Rxpermissions

RxJavaAndroid M◦

Rxjava

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

Rxjava2

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

Retrolambda

```
// 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
        }
    });
});
```

<https://github.com/tbruyelle/RxPermissions>。

AndroidRxJava <https://riptutorial.com/zh-CN/rx-java/topic/7125/androidrxjava>

3: RxJava2 FlowableSubscriber

rxjava2FlowableSubscriber

rxjava2maven

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

Examples

```
TestProducerIntegerSubscriber o Flowable<Integer>o Subscriptionrequest(long) Integero
```

```
subscriberSubscription onNext () onNext () request ()o outStandingRequestsisProducingo
```

```
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;
}
);
}
}

```

ConsumerDefaultSubscriber<Integer> Integer。。

```

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");
    }
}

```

```

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();
    }
}
```

rxjava2Flowable。

RxJava2 FlowableSubscriber <https://riptutorial.com/zh-CN/rx-java/topic/9810/rxjava2-flowable-subscriber>

4:

- Subject <TR> subject = AsyncSubject.create; //AsyncSubject
 - Subject <TR> subject = BehaviorSubject.create; //BehaviorSubject
 - Subject <TR> subject = PublishSubject.create; //PublishSubject
 - Subject <TR> subject = ReplaySubject.create; //ReplaySubject
 - mySafeSubject = new SerializedSubject; //unsafeSubjectsafeSubject -



Subject. .

Examples

RxJava Subject Observable Observer 。 Observable Observer Observable。

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

“HelloWorld” Subjects.

1. PublishSubject Subject

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

2 Observers

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

Subject:

3 onNext Observables

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

Subject.

Subject RxJava

- AsyncSubject

- BehaviorSubject
- PublishSubject
- ReplaySubject

SubjectSerializedSubject。 Subject ***Observable Contract***

- Dave Sexton

PublishSubject

PublishSubjectObserverObservable。

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
```

PublishSubjectObservable **500Long**。 PublishSubject clock sub1sub2。

PublishSubject**unsubscribe**。

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

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

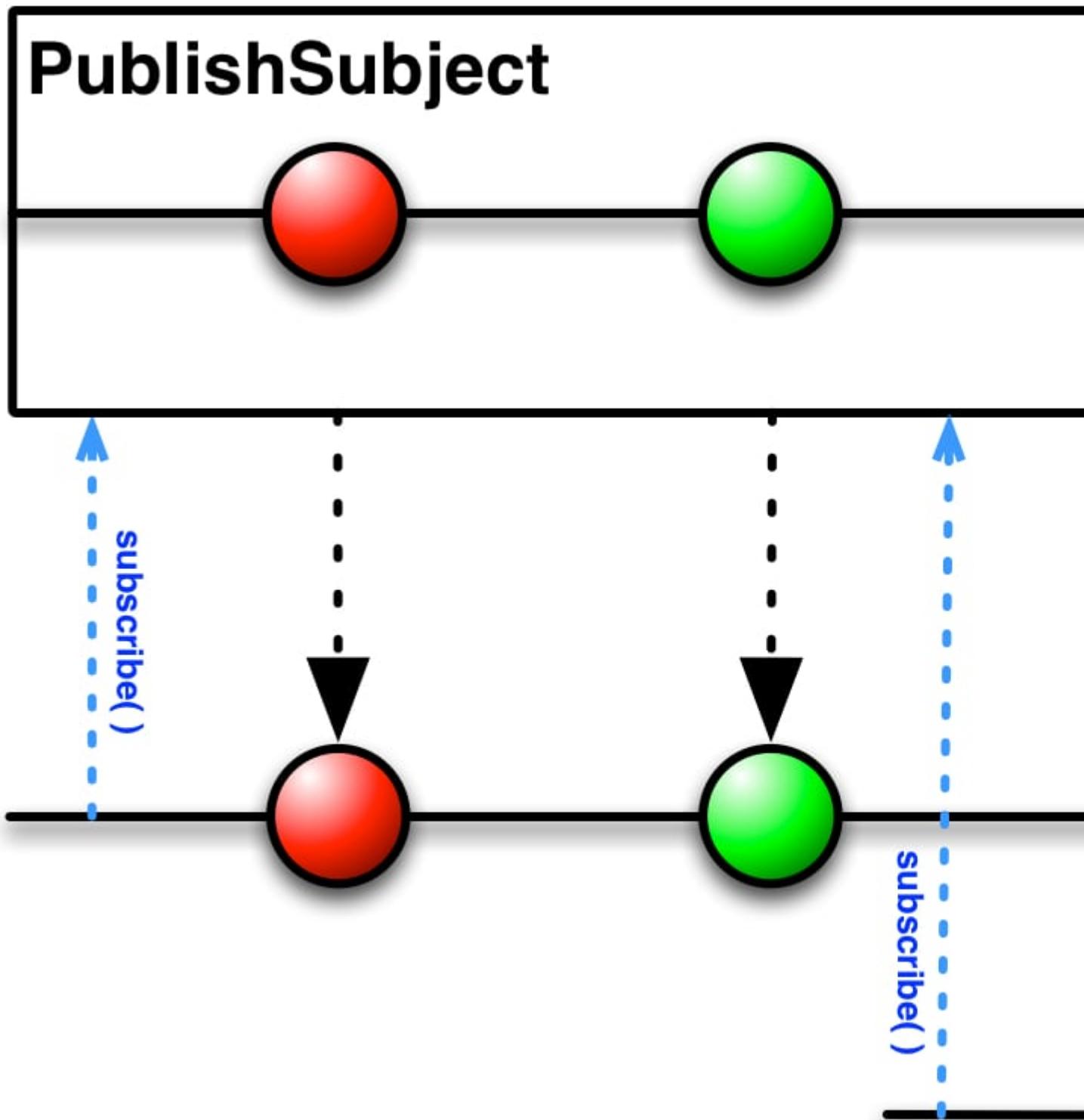
sub1andsub2(); // 6 lines moved for brevity. same as above example
```

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

```
sub110◦ 5◦ ◦ PublishSubjectHot Observable PublishSubject◦
```

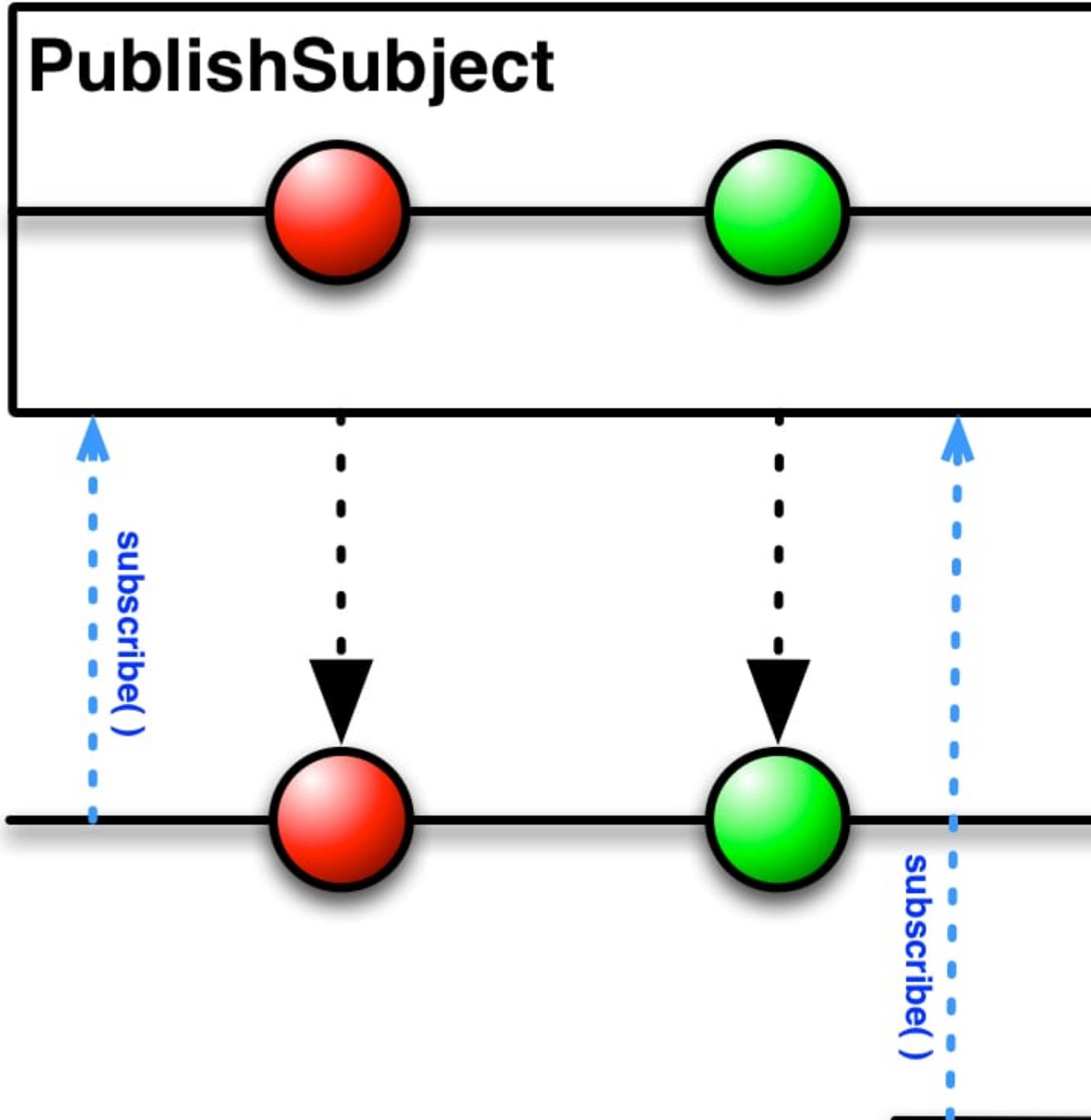
HPublishSubject n◦

PublishSubject



PublishSubjectObservableonCompleted◦

ObservablePublishSubject Observable。



◦

```
/* 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();
```

PublishSubject`watcher`。

- PublishSubject [javadocs](#)
- Thomas Nield

<https://riptutorial.com/zh-CN/rx-java/topic/3287/>

SchedulerJVMRxJava。 TestScheduler。 Schedulers。

Examples

TestSubscriber

TestSubscribersAction <> Observable。

1,2,34onNextObservable。

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

assertValues。

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

assertValueequals。

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

Observable。

```
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
```

assertValue.

o

```
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();
```

getOnNextEvents getOn*Events

```
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

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

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

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

```
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

TestSchedulersObservables. . .

TestSchedulersRxJava

```
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

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

TestScheduler.

TestScheduler。。RxJavaSchedulers.io/ computation/ etc。 RxJavaHooks。 Scheduler。

```
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;  
    }  
}
```

◦◦◦ TestScheduler◦◦◦

```
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)
```

RxJava

<https://riptutorial.com/zh-CN/rx-java/topic/5207/>

6:

Examples

Observable

RxJavaObservable◦ Observable.create◦ ◦ ◦

Observable.just◦

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

Observable.fromCallable◦

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

ObservablelongComputation()◦ ◦

Observable.deferObservable.fromCallableObservable◦ ◦

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

HotCold◦

Cold ObservableHot Observable◦

```
/* 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
```

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

```
sub2, 2
```

```
sub2。 Cold Observable。。
```

Hot observables。。

```
publishCold ObservableHot Observable。
```

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

```
publishConnectableObservable observable。
```

```
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));
```

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

```
sub2sub1。
```

```
SubscriptionObservable。
```

```
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();
```

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

Observable”。

Hot ObservableEventBus ◦ EventBuses◦ RxBus◦

<https://riptutorial.com/zh-CN/rx-java/topic/1418/>

7: RxJava

Examples

RetrofitRxJava

Retrofit2RxJava

RxJavaRetrofit RxJava

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

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

API Observable

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

SingleObservable

RxJava flatMap

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

zip

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

RxJava <https://riptutorial.com/zh-CN/rx-java/topic/2950/rxjava>

8:

Examples

Observable。

```
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);
```

100。 compute(int) Observable。 foronNext。

◦ Rx.NETRxJava100。 101001000OutOfMemoryErrorGC。

onErrorXXXonBackpressureXXX。

PublishSubject。 interval。

RxJavaobserveOn MissingBackpressureException。。

MissingBackpressureException。

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

Thread.sleep(10_000);
```

◦ “” observeOnrange observeOn。

◦ rangeProducerSubscriber setProducerobserveOn。 observeOnProducer.request(n) range onNext。 observeOnrequest。

```
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!");
    }
});

});
```

onStartrange onNextonNext 。 compute(int)rangeo rangeonNext StackOverflowError 。

◦ rangeonNext () request(1)onNext ()onNext () 。

```

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

    compute(v);
}
```

onStart。 ObservableSubscriberrequest(1)。 onNextrequest(1)

```

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
});
```

onStartNullPointerException 。 request(1)onNextnameonNextonStartrequest 。

onStartrequest() 。 request()。

onBackpressureXXX

```
MissingBackpressureException observeOn PublishSubject timer() interval() create()。
```

◦

◦ **observeOn Android16.**

RxJava◦ bufferSize prefetchCapacityHint ◦ observeOn◦

```
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);
}
```

◦ ◦

/

/MissingBackpressureException◦

```
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);
}
```

Observable throttleFirst throttleLast throttleWithTimeout◦

```
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);
}
```

MissingBackpressureException◦

onBackpressureBuffer

- JVM◦

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

observeOnMissingBackpressureExceptiononBackpressureBuffer100observeOn◦

onBackpressureBuffer◦ range◦

onBackpressureBuffer4

onBackpressureBufferint capacity

BufferOverflowError◦

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

◦ “”onBackpressureBuffer◦

onBackpressureBufferint capacityAction0 onOverflow

◦ ◦

onBackpressureBufferint capacityAction0 onOverflow BackpressureOverflow.Strategy strategy

◦ BackpressureOverflow.StrategyBackpressureOverflow4

- ON_OVERFLOW_ERROR BufferOverflowException
- ON_OVERFLOW_DEFAULT ON_OVERFLOW_ERROR
- ON_OVERFLOW_DROP_LATEST ◦
- ON_OVERFLOW_DROP_OLDEST ◦

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

◦ BufferOverflowException◦

onBackpressureDrop

element。①onBackpressureBufferON_OVERFLOW_DROP_LATEST。

GPS。

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

interval()。

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

onBackpressureDrop(Action1<? super T> onDrop)。。

onBackpressureLatest

◦ onBackpressureBuffer②ON_OVERFLOW_DROP_OLDEST◦

onBackpressureDrop◦

◦

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

onBackpressureDrop◦

Observable^③/_④◦

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
})
```

```
onStart。 just。
```

```
justSubscriber S
```

```
int counter;  
  
int computeValue() {  
    return ++counter;  
}  
  
Observable<Integer> o = Observable.just(computeValue());  
  
o.subscribe(System.out::println);  
o.subscribe(System.out::println);
```

112。

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

```
computeValue。
```

fromCallable

```
fromCallable
```

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

```
computeValue 12. fromCallable。 justmap
```

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

```
justcomputeValue。
```

```
Iterablefrom
```

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

210 justfrom。

```
from(Iterable)。 。 。
```

IterableObservable CJava yield return yield break。 Google Guava AbstractIterable IxJava
Ix.generate() Ix.forloop()。 Iterable

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

```

    }

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

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

```

for iterator。 Observable 5。 Observable。

SyncOnSubscribe

getread。 Iterable。

RxJava SyncOnSubscribe。 lambda。

```

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);

```

SyncOnSubscribe3。

```

FileInputStream;

Observer onXXX。 。 onNextonErroronCompleted。 onCompleted() IOException onError。

;。 SyncOnSubscribe。

```

JVM try-catch es。

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

current0**lambda**current1。

AsyncOnSubscribeSyncOnSubscribeObservable。 Observable。

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

◦ ◦

Observable**API**。

RxJava**create(emitter)**。

- Emitter<T>
- Emitter.BackpressureMode。 onBackpressureXXX MissingBackpressureException。

◦ NONEObservableonBackpressureXXXonBackpressureXXX。

GUI。 API**addListener / removeListener**

```
Observable.create(emitter -> {  
    ActionListener al = e -> {  
        emitter.onNext(e);  
    };  
  
    button.addActionListener(al);  
  
    emitter.setCancellation(() ->  
        button.removeListener(al));  
  
, BackpressureMode.BUFFER);
```

Emitter; onNext onErroronCompleted。 API**setCancellation setSubscription Subscription**
setSubscription onError /。 EmitteronCompleted。

◦ CompositeSubscription CompositeSubscription

```
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);

```

API Observable。

```

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);

```

◦ API previous ◦ LATEST;BUFFER128◦

<https://riptutorial.com/zh-CN/rx-java/topic/2341/>

Examples

RxJava · Executor ·

Scheduler

- FIFO
-

Scheduler delay subscribeOn / observeOn

Scheduler · delay · Scheduler ·

Observable subscribeOn · Scheduler ·

Observable observeOn · Scheduler observeOn · observeOn ·

subscribeOn

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

Scheduler

```
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()));
```

Scheduler ·

```
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);
```

```
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);
```

```
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);
```

Web Socket Observable

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

<https://riptutorial.com/zh-CN/rx-java/topic/2321/>

10:

◦

Examples

ObservableSubscriber◦

```
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);
```

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

mapInteger observableString observable◦

chained◦

```
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
```

ObservableSubscriber。

flatMap

flatMapObservable。

Observable

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

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

flatMap perform performconcatMap。

Observable flatMap。 **Observable**

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

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

Reactivex.io <http://reactivex.io/documentation/operators/flatmap.html>

filter。

ObserverSubscriber filterfalse。

```
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
    });
});
```

```
0.0  
4.0  
16.0  
36.0  
64.0
```

```
mapmap o o
```

```
.map () o
```

```
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  
    });
```

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

Observable List<Integer> List<String> .subscribeString

doOnNext

```
doOnNext Observable doOnNext o ...
```

```
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();
```

```
doOnNext Observable Observable.empty() onCompleted o
```

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

```
repeat Observable o
```

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

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

o

repeat。

```
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")
    );
```

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

Observable repeatObservable。 Observable.create。

```
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")
    );
```

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

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

repeat。

```
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")
    );
}
```

```
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
```

repeatObservablemaprepeat。

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

```
Observable.<Integer>create(subscriber -> {
    //...
})
    .repeat(3)
    .map(value -> value * 2) //First chain operator
```

```
.map(value -> "modified " + value) //Second chain operator
.subscribe(
    /*....*/
);

```

<https://riptutorial.com/zh-CN/rx-java/topic/2316/>

S. No		Contributors
1	rx-java	Buttink , Community , dimsuz , Dmitry Avtonomov , Hans Wurst , hello_world , Omar Al Halabi , Saulius Next , Sneh Pandya , svarog , Tom
2	AndroidRxJava	akarnokd , Athafoud , Daniele Segato , Eugen Martynov , Geng Jiawen , Sneh Pandya
3	RxJava2 Flowable Subscriber	P.J.Meisch
4		hello_world , mavHarsha
5		Buttink , Sir Celsius
6		Aki K , dwursteisen , hello_world , JonesV
7	RxJava	LordRaydenMK
8		akarnokd , Bartek Lipinski , Chris A , Cristian , dwursteisen , Niklas , Sebas LG
9		dwursteisen , Gal Dreiman
10		dwursteisen , hello_world , svarog , Vadeg